NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

510.7808
C28.2
78-151
c.2

CMU-CS-78-151

THE IMPLEMENTATION AND EVALUATION OF PARALLEL ALGORITHMS ON C.MMP

P.N. Oleinick

Computer Science Department
Carnegie-Mellon University
November 1978

Keywords: performance evaluation, multiprocessors, synchronization, parallel algorithms, cooperating processes.

The research described here was supported by the Defense Advanced Research Projects Agency (Contract: F44620-73-C-0074, monitored by the Air Force Office of Scientific Research), and in part by the Office of Naval Research (Contract: N00014-77-C-0500).

University Libraries
Carnegie Mellon University
Pittsburgh PA 15213-3890

ABSTRACT

This dissertation demonstrates the implementation and evaluation of parallel algorithms on C.mmp, a multiprocessor computer system. Initial attempts to demonstrate the performance of a simple parallel algorithm yielded unexpectedly large performance degradations from the theoretical calculations. This unexpected result spawned a study of the C.mmp system to discover and measure the major sources that perturbed the performance of the parallel algorithm. The performance study was conducted at several levels:

- Basic hardware measurements
- Runtime performance of Hydra, C.mmp's operating system
- Overall performance of a particular application: a parallel rootfinding algorithm.

The results of this study identified six major sources of performance pertubation. The six sources, in order of importance, were:

- Variations in the compute time to perform the repetitive calculation
- Memory contention caused by finite memory bandwidth
- The operating system's scheduling processes can become a bottleneck
- Variations in the individual processor speeds
- Interrupts associated with I/O device service routines
- Variations in the individual memory bank speeds.

The effects that synchronization can have on the performance of a parallel algorithm were examined apart from the sources mentioned above. Several alternative synchronization primitives were studied. For each, the speed in performing the basic semaphore operations as well as the effect on the performance of the rootfinding algorithm were measured. The type of semaphore primitive selected to perform the synchronization of the rootfinding processes drastically affected the performance of the algorithm. A threshold for the practical application of each semaphore was determined from the measurements of the rootfinding algorithm.

This insight into the C.mmp environment was applied toward a more complex application--the HARPY speech recognition system. Parallelism was incorporated into the algorithm by

decomposing the large task into a sequence of computationally smaller sub-tasks. Each sub-task was implemented as a collection of indentical cooperating processes.

Inefficient allocation of work to processes, and synchronization between sub-tasks resulted in under utilization of the processors. Performance of the algorithm was improved in three subsequent refinements to the initial implementation. The contribution to performance from each enhancement was disscused and measured separately.

The final implementation of HARPY on C.mmp was compared to a version of the algorithm developed for a DEC PDP-KL10 uniprocessor. At maximum parallelism, eight processes, the C.mmp implementation performed the speech recognition task 30% faster than the uniprocessor.

Acknowledgement

I would like to thank Sam Fuller, my advisor, for his constant encouragement during the development of this work. His guidance and support were invaluable. I also want to thank Anita Jones for her insight and especially for her blue pencil. I am grateful for the time she gave to polish my prose and to augment my arguments by playing devil's advocate. The comments from the other members of my committee, Bill Wulf and Don Thomas, helped me define the organization of this thesis. Judy Rosenberg's assitance let me go home early.

I am especially grateful to my best friend Barbara McKissock who read this thesis eight times and didn't complain once.

Table of Contents

1. Introduction

The purpose of this research is to demonstrate how to write parallel programs that effectively use the multiple computers in a multiprocessor. Developing strategies for incorporating parallelism into algorithms has been an area of intense interest for quite some time, e.g., [Avriel and Wilde 66], [Karp and Miranker 68], [Rosenfeld and Driscoll 69], [Heller 76], [Thompson and Kung 76], [Baudet, Brent and Kung 77] and [Baudet 78]. However, until very recently, only simulation and analysis techniques were available for demonstrating the effectiveness of a parallel algorithm.

With the emergence of multiprocessor computer systems that provide users with the facilities for constructing parallel algorithms, CM* and C.mmp[1], the verification of an algorithm's performance is in its implementation. Initial attempts to demonstrate the performance of a simple parallel algorithm [Fuller and Oleinick 76] yielded unexpectedly large degradations in the algorithm's performance. These degradations were not the result of an error or inefficiency in decomposing the problem into cooperating processes. Rather, several non-algorithmic sources were determined to be the source of the degradations. This result indicates that in order to develop effective parallel algorithms for multiprocessors, it is necessary to be aware of the target machine's performance characteristics.

Presently, the task of writing effective parallel software is an ad-hoc procedure of constructing code for a unique machine. Since multiprocessors are almost as different from one another as they are from uniprocessors, it is difficult to apply insight gained from writing parallel software for one multiprocessor to another machine. However, by documenting the performance of various implementations of several algorithms on one machine, we can demonstrate the effectiveness of various strategies at capturing parallelism.

One style of parallel programming for multiprocessors involves tightly coupled cooperating processes. Several decomposition strategies exist that use this approach, among them *pipelining* and *partitioning* [Jones 78]. In both cases, simultaneously executing processes must interact frequently. Since interprocess communication constitutes an overhead, tightly coupled systems exhibit performance degradations proportional to the amount of process interaction among the processes. Thus, in order to maintain high performance, one must reduce both the overheads of interprocess communication and the amount of process interaction.

---

[1] C.mmp and CM* are multiprocessors developed at Carnegie-Mellon University. [Jones 78], [Fuller 78], [Wulf and Bell 72], [Swan, Fuller, and Siewiorek 77]

The user has little power to reduce the overhead of interprocess communication. Since processes are created and maintained by the operating system, interprocess communication is permitted in only a few well defined ways. The user is given a selection of primitives provided by the operating system with which he can build his own communication mechanisms. However, the performance of the communication mechanism is directly influenced by the performance of the operating system's primitive.

Moreover, writing effective parallel software requires an awareness of more than just the overheads involved in interprocess communication. We adopted the following two phase strategy for uncovering the major influences on performance:

1. Develop a simple parallel algorithm as a vehicle for conducting a performance study on C.mmp.

2. Use this test program to measure the effects on performance stemming from both the hardware and the operating system.

A brief introduction to the C.mmp environment, both hardware and operating system is contained in chapter two. In addition, chapter two contains the development and theoretical performance calculations for the simple parallel algorithm.

The investigation into the sources of performance perturbation is presented in chapter three.

Since synchronization is a fundamental parallel programming issue, chapter four is devoted entirely to studying the effects of synchronization on performance. The performance of various synchronizaton primitives is conducted at two levels: the speed in performing the basic synchronization operations and the impact each primitive has on the performance of the rootfinding algorithm.

In chapter five, we apply the insights gained from the initial investigation toward developing complex tightly coupled systems. By decomposing a complex task into a sequence of simpler sub-tasks, and then implementing these sub-tasks as *task forces*[Jones 78] of cooperating processes, we efficiently focus compute power to speed up the execution of the task. To demonstrate the effectiveness of this approach we use it to implement a parallel version of the Harpy speech recognition system[Lowerre 76].

An initial decomposition of the algorithm is successively refined in three implementations. In each iteration, some aspect of performance is improved. This incremental enhancement of the algorithm enables us to measure the performance improvement contributed by each enhancement.

Chapter six contains a summary of the measurements and results of this investigation. The initial measurements of the multiprocessor and the results to come out of the rootfinder study are summarized. The performance of the task force approach to parallel programming is evaluated based on the results of the various implementations of the Harpy algorithm. Finally, areas for further research are discussed.

2. An Introduction to C.mmp and The Rootfinding Algorithm

2.1 An Introduction to C.mmp

The basic structure of C.mmp, as shown in the PMS diagram of figure 1, is that of the canonical multiprocessor. A detailed description of C.mmp is provided in the original article on C.mmp by Wulf and Bell [1972], but the following description should provide a sufficient background for this investigation.

C.mmp is organized as a system of 16 central processors (Pc's) that share a centrally located large primary memory that presently consists of 2.5 Megabytes. The 16 Pc's are completely asynchronous computing elements: 5 are PDP-11/20's and the remaining 11 are PDP-11/40's. They are connected to the shared primary memory through a 16 x 16 crosspoint switch. The operation of the switch is similar to a 16 port memory in that up to 16 memory transactions can be performed simultaneously. I/O devices, unlike memory, are associated with an individual processor. Thus, for example, an I/O request to a device on Pc[0], perhaps a disk, is performed by the requesting Pc by sending an interprocessor interrupt to Pc[0] causing initiation of the appropriate I/O interrupt service routine on Pc[0].

Hydra is C.mmp's general-purpose multiprogramming operating system [Wulf *et al.*, 1974; Levin *et al.*, 1975]. It is a collection of basic or *kernel* mechanisms such as memory management, process dispatching, and message passing. Upon this core, an arbitrary number of systems created from these mechanisms can co-exist simultaneously. Hydra is organized as a set of re-enterant procedures that can be executed by any of the processors. In fact, several processors can simultaneously execute the same procedure. This concurrency is accomplished by placing *locks* around the operating system's critical data structures. These locks maintain mutual exclusion where necessary.

2.2 Description of the Rootfinding Algorithm

The purpose of this study is to present quantitative performance results for implementing parallel algorithms on a multiprocessor. Rather than attempting to measure a broad spectrum of problems, we have chosen to study various implementations of a single problem in order to observe and measure in depth the performance tradeoffs in the implementation process.

Two criteria that our case study problem had to meet were: the problem must be complex enough to have interesting implementation tradeoffs and simple enough to permit the focus of attention on implementation issues rather than algorithm issues. The candidate problem we

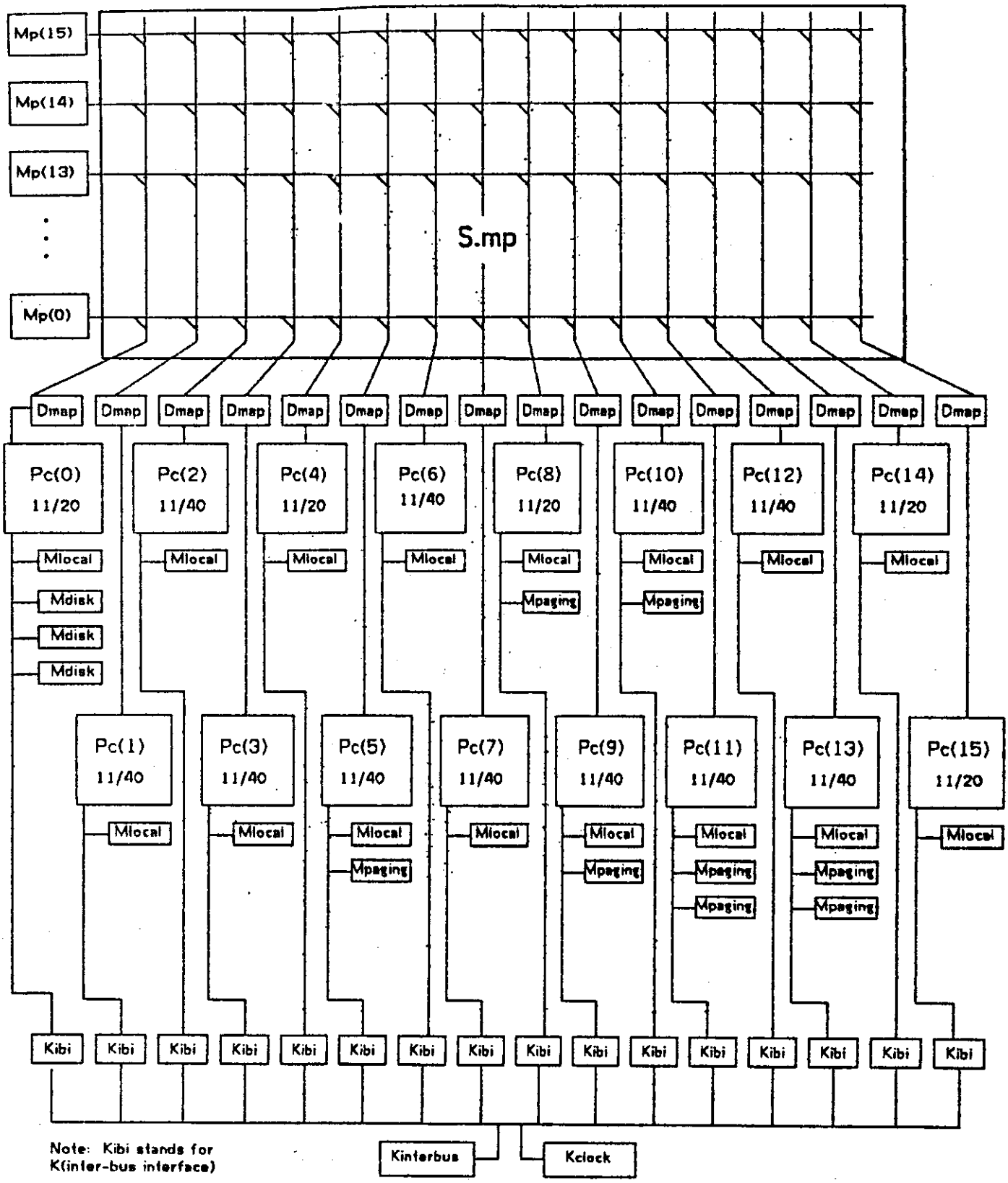


Figure 1 PMS Diagram of C.mmp (1977)

finally selected is the rootfinding task.

We have chosen to consider this problem not because it particularly well-suited for parallel solution, but rather because it is a relatively straight forward task that requires a significant amount of inter-process communication. According to Stone[1973], algorithms like the rootfinding algorithm that exhibit speed-up gains proportional to the logarithm of the number of processes fall into a class of problems at best considered poor candidates for parallel processing. However, the underlying control structure present in this procedure, that of the synchronous parallel algorithm, is representative of many parallel decompositions of otherwise serial algorithms. For this reason, it is worthwhile to understand the nature of the control structure and to study the influences on its performance.

Specifically, we will consider the problem of finding the root of a monotonically increasing function in a bounded region. If we assume no special information about the behavior of the function, the best procedure for a uniprocessor under these circumstances is a binary search. An obvious decomposition of the binary search into n parallel processes on a multiprocessor is to evaluate the function simultaneously at n equidistant points within the bounded region.

The optimal placement of the n processes on the interval is known [Kung 1976], but to minimize the complexity of the algorithm in order to focus on the synchronous control structure, we will use the less than ideal, but good, technique illustrated in figure 2. The n parallel processes perform function evaluations at the n points that divide the interval into $n+1$ equal subintervals. Since our function, F(x), is a monotonic function, the sub-interval that contains the root is the sub-interval with opposite signs for F(x) at its end points. The other sub-intervals are discarded and the procedure repeats this basic iteration until one of the function evaluations is within ϵ, i.e. an acceptably small interval close to zero, of the zero-crossing.

For the measurements presented here, the function we are evaluating is the normal integral:

$$F(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty \exp(-1/2t^2)\ dt \quad - \quad h \tag{2.1}$$

For $x < 2.32$ the following truncated power series was used to evaluate F(x):

$$\left(x + \frac{x^3}{3} + \frac{x^5}{3*5} + \frac{x^7}{3*5*7} + \frac{x^9}{3*5*7*9} + \ldots \right) \quad - \quad h \tag{2.2}$$

and for larger x we used the continued fraction:

$$1/(\ x+1/(\ x+2/(\ x+3/(\ x+\ldots)))) \quad - \quad h \tag{2.3}$$

We selected this normal integral because it is an important transcendental function that exhibits two characteristics important to our measurement studies: it requires an extensive amount of computation, and the type and length of computation are data dependent.

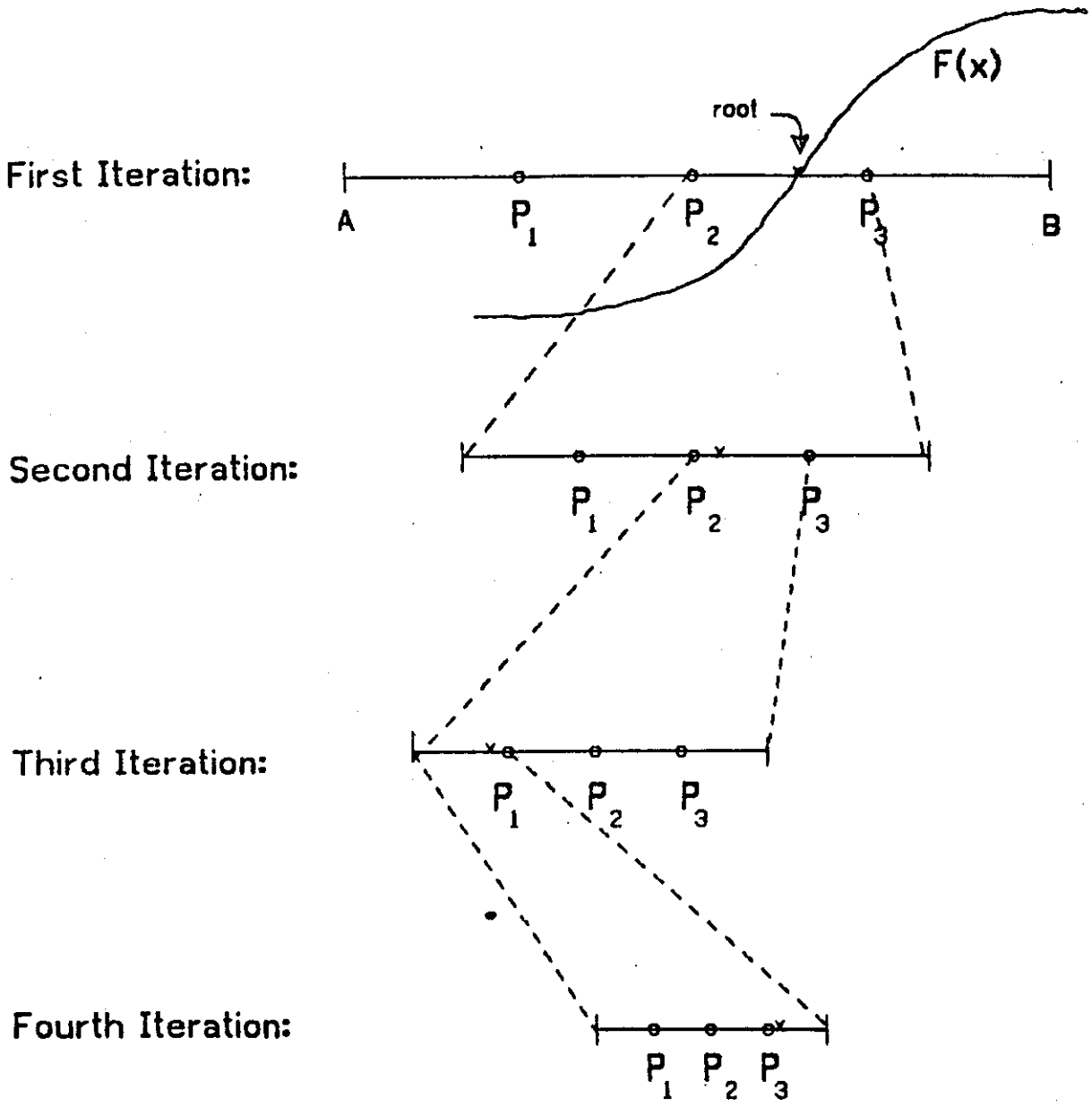


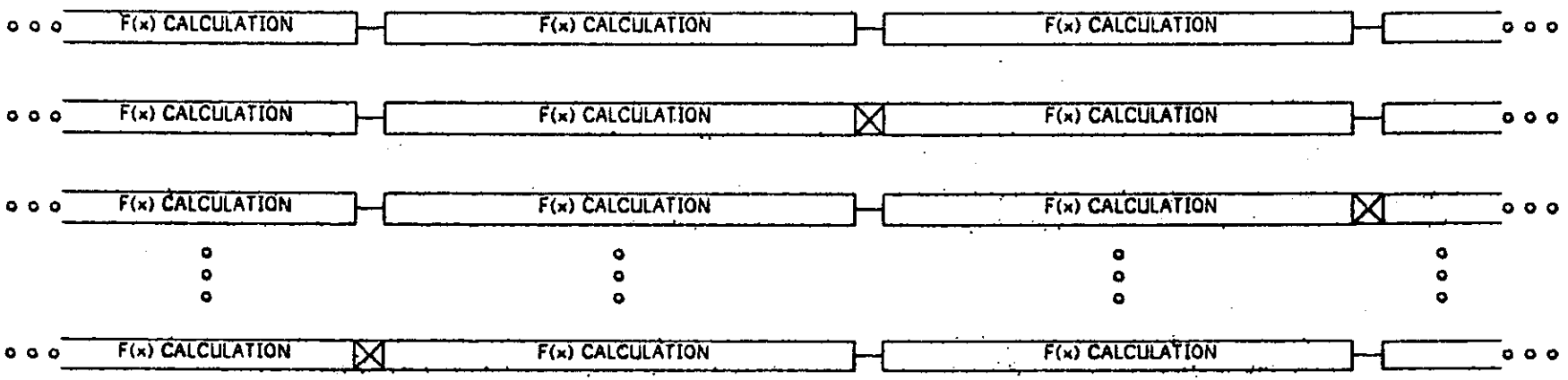
Figure 2 Rootfinding Program Using three Processors

In order to evaluate the performance of our implementations of the rootfinding algorithm, we first calculate the theoretical, or overhead-free, performance curves.

The basic cycle in the rootfinder is the independent evaluation of the function by each of the cooperating processes and, upon finishing, the communication of each process with the other processes by posting the results of its function evaluation. Notice that the new interval is not located until all of the processes have posted their results[1]. When the last process finishes its function evaluation, it assumes the jobs of finding the new root-containing interval and *waking up* all of the waiting processes. This basic cycle we call a STAGE.

Under ideal conditions the cooperating processes in the rootfinder would exhibit the execution behavior found in figure 3. Each process performs a function evaluation independently. They all finish at the same instant and, after a very brief bookkeeping calculation, perform a new F(x) calculation on an interval reduced by 1/(n+1). In practice, we seldom find this to be the case. The fluctuations in performance stem from sources intrinsic to the multiprocessor as well as the rootfinding program.

[1]The new interval is located as soon as the sub-interval is bounded, but again we have opted for a more straightforward algorithm in order to focus on the implementation issues.



⊠ - LOCATING THE INTERVAL THAT CONTAINS THE ZERO-CROSSING AND REDISPATCHING THE N PROCESSES

Figure 3 Optimal Performance of the Rootfinding Algorithm

3. Sources of Performance Fluctuation

3.1 Introduction

Three distinct sources of performance fluctuation are: the variation in the amount of computation required to perform a function evaluation, the individual hardware elements' performance characteristics, and the operating system. We will identify the nature and measure the magnitude of each of these sources starting with the variation in the F(x) calculation since it is the most straight forward of the three.

3.2 The Variation in the F(x) Calculation

The elapsed time to perform a function evaluation is data dependent. The distribution of the compute time is shaped approximately Normal as shown is figure 4. The mean is about 100 milliseconds with almost an equal number of samples on each side of the mean[1]. Thus, we might model the expected finishing time for a process performing an F(x) calculation to be a random variable with a Normal distribution. As other functions would have other compute time distributions, we derive the theoretical performance for the constant and exponential cases also.

Let the time taken by the i^{th} stage in the rootfinding procedure be the random variable T_i. Since all of the processes are performing the same calculation, each process executes for a random amount of time, t (see figure 5), taken from some distribution. Since all of the processes must finish their function evaluations before the new sub-interval is located

$$T_i = MAX(t_1, t_2, t_3, \ldots , t_n) \tag{3.1}$$

From elementary order statistics the expected value of the largest order statistic in random samples of n from a parent distribution with continuous strictly increasing cumulative distribution function P(x) is

$$E(X_{(n)}) = \int_{-\infty}^{\infty} nx[P(x)]^{n-1} dP(x) \tag{3.2}$$

If we know nothing about the distribution of the t_i other than the mean u and standard deviation s, the expected value of the largest order statistic T_i, reduces to

$$E(T_i) \leq \mu + \frac{n-1}{\sqrt{(2n-1)}} * \sigma \tag{3.3}$$

---

[1] On an 11/20 processor

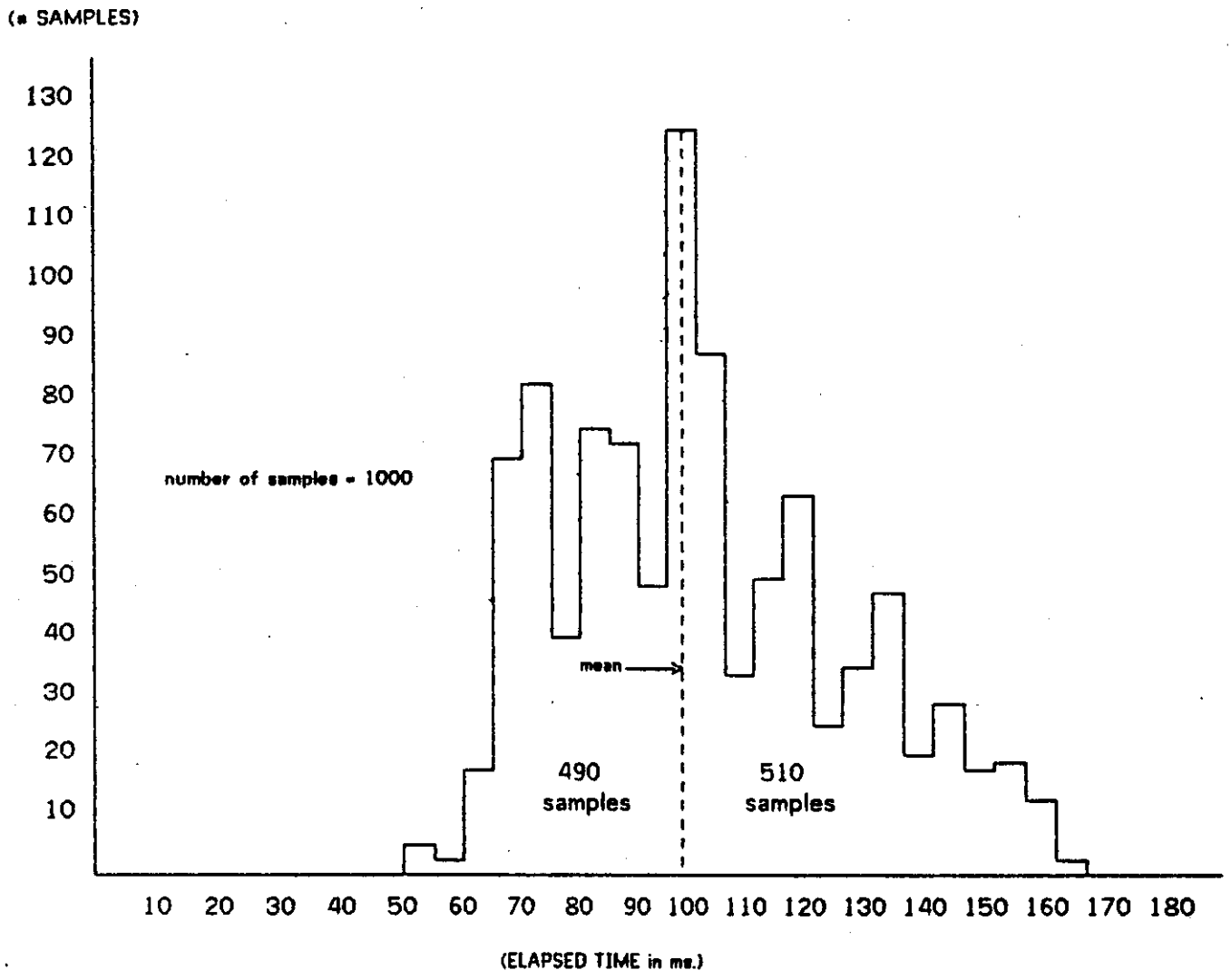
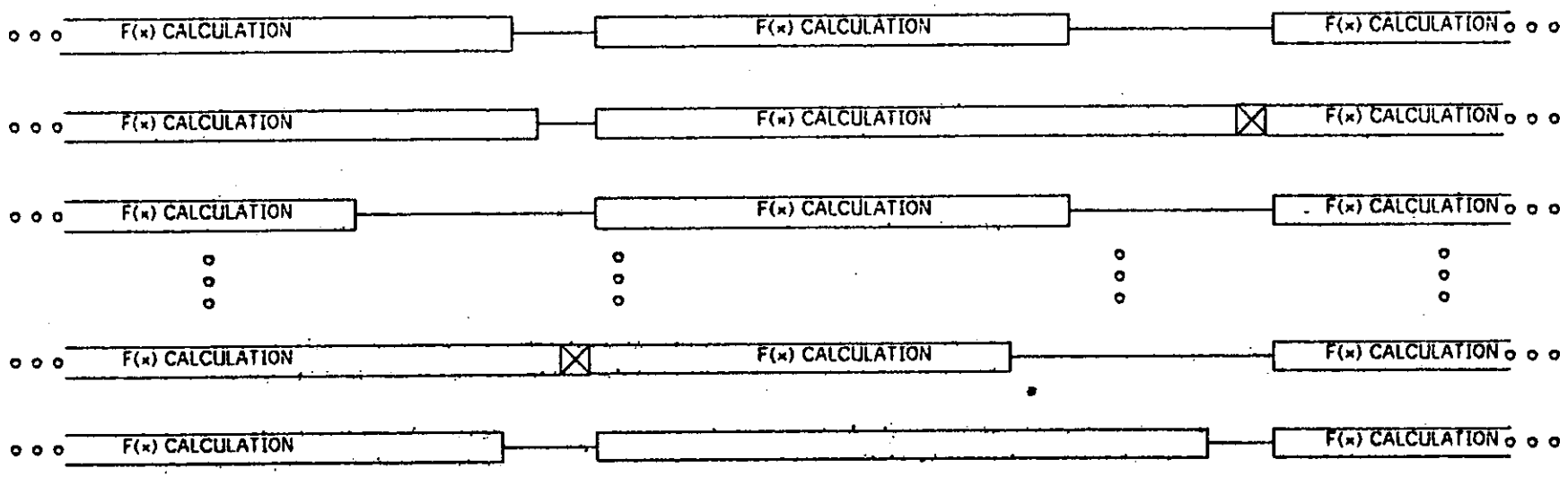


Figure 4 Distribution of the Time to Calculate F(x)



⊠ = LOCATING THE INTERVAL THAT CONTAINS THE ZERO-CROSSING AND REDISPATCHING THE N PROCESSES

Figure 5 Performance Degradation Due to Variation in the F(x) Computation Time

This bound can be replaced in the exponential case by the equality

$$E(T_n) = n\mu \sum_{j=0}^{n-1} \binom{n-1}{j} \frac{(-1)^j}{(j+1)^2} \qquad (3.4)$$

For the Normal case we consult Teichroew's[1956] tables for the expected value of the largest order statistic drawn from the N(0,1) distribution.

When the expected value of the compute time is a constant, equation 3.3 is replaced by the simple equality $E(T_i) = \mu$.

If we are interested in the performance speedups obtained when we put more processes to work finding roots, we need to estimate the average time to locate a root as a function of the number of processes. Since every iteration in the rootfinding procedure reduces the interval of uncertainty, L, by a factor of $n+1$ it takes *Ceiling*$(\text{Log}_{n+1} L)$ iterations to locate the root in a bounded interval of length L. Thus, in our example let R_i denote the number of iterations necessary to arrive within ϵ of the root using i processes. For our choice of ϵ, R={54, 34, 27, 23, 21, 19, 18, 17, 16, 16, 15, 15,...} iterations. It takes the same number of iterations to locate the root using nine and ten or eleven and twelve processes because the number of iterations to locate the root must be an integer. Thus, little is to be gained by incorporating many processes in the procedure. In this study the maximum number of processes we will use is nine.

We can estimate the runtime of the rootfinder to be the following:

$$\text{Runtime}(n) = \sum_{k=1}^{R_n} T_k = R_n * E(T_n) \qquad (3.5)$$

Often we will be interested in the speedup achieved through parallelism. We will use the following formula to calculate speedup:

$$\text{Speed up}(n) = \frac{\text{Runtime}(1)}{\text{Runtime}(n)} \qquad (3.6)$$

Figure 6 is a plot of the speedup *vs.* number of processes for the following three distributions:

Distribution	Mean	Standard Deviation
Constant	1000	0
Normal	1000	278
Exponential	1000	1000

The curves are not smooth because the *Ceiling* function in the equation for the number of iterations to perform yields an integer value.

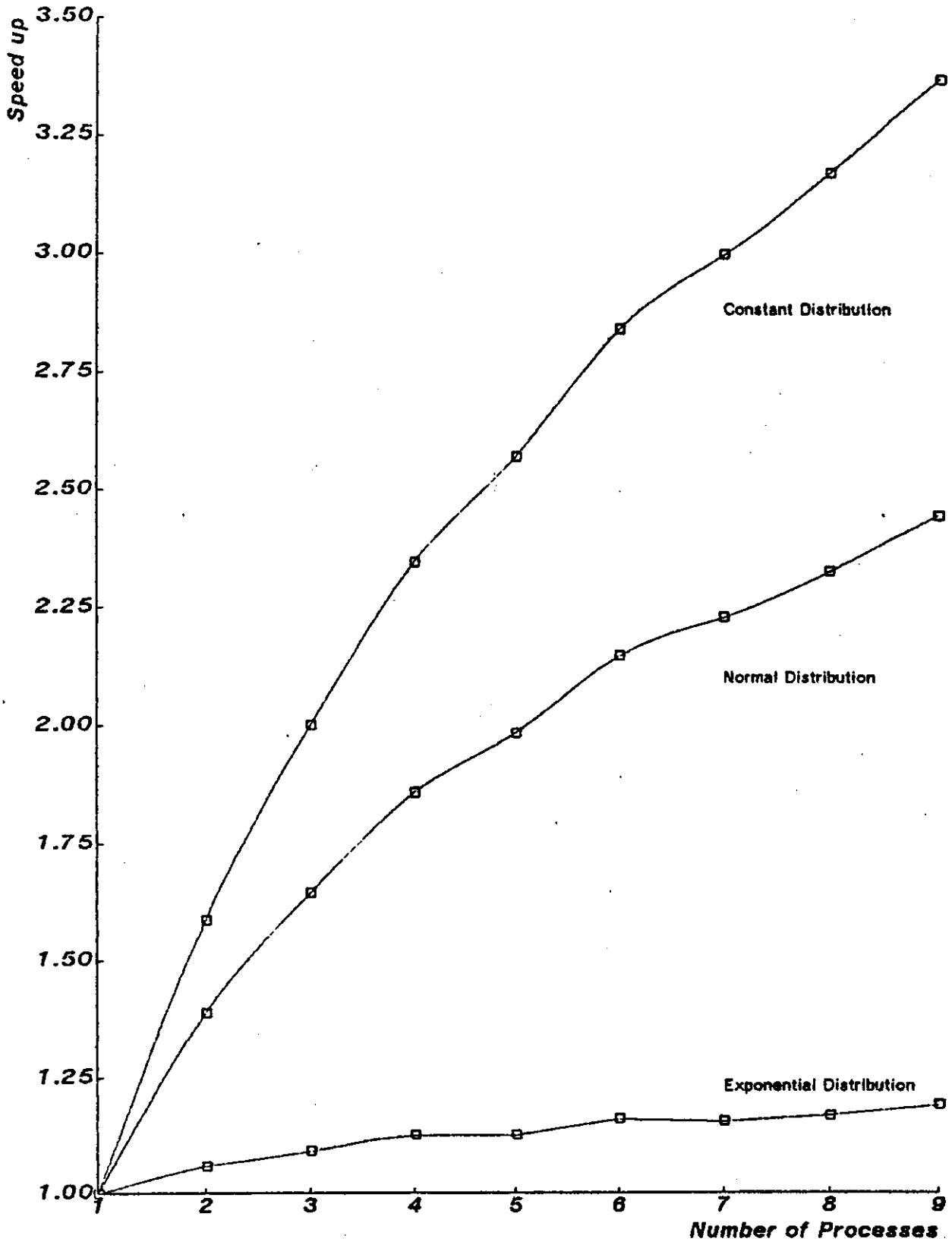


Figure 6 Speed up vs. Number of Processes for Ideal Multiprocessor

This figure contains calculated no-overhead performance curves for three sample F(x) distributions with standard deviations ranging from 0 to μ. The performance range is from negligible speedup when the compute time for the function evaluation is exponentially distributed to more than a factor of 3.3 speedup for nine processes when the distribution of the F(x) calculation is a constant. The Normal curve between these extremes closely approximates the actual F(x) distribution and is included for comparison.

Another way to view this data is to plot speedup for the nine processes case *vs.* the ratio standard deviation/mean as was done in figure 7. This figure clearly shows the impact of the variance on the performance of the rootfinding procedure. When the coefficient of variation is much greater than one, no speedup can be obtained by incorporating multiple processes in the rootfinding task.

Now we compare the calculated no-overhead performance of the rootfinder to measured data observed on the machine. In order to measure performance as a function of the distribution of the F(x) compute time a synthetic rootfinder was developed because we did not want to limit our investigations to particular distributions too early in the experiment. The nature of the calculation was still the real function evaluation; however, the length of time spent computing was adjustable to reflect the distribution under consideration.

Figure 8 graphs performance in terms of elapsed time as a function of the number of processes for three distributions of the F(x) calculation. In each case we compare theoretical performance to measured data. Since the means of the three distributions were not identical, the data points for the single process instantiation do not coincide. Thus, in this graph comparisons across distributions can be only relative approximations. What is important here is how close the measured curves are to their theoretical curves.

For each single process instantiation the measured and theoretical curves are far apart. This discrepancy is because any perturbation the process experiences will be additive and will lengthen the basic cycle time.

As we incorporate more processes the constant distribution diverges the most from the theoretical while the exponential diverges the least. The reason for this behavior is that perturbations experienced by the processes will tend to increase the variance of the underlying distribution. However, a small change in the variance of the constant distribution will be a much larger relative change than a similar change to the exponential distribution.

That the observed data doesn't agree closely with the calculated curves is evidence that other influences on performance exist in addition to the distribution of the compute time. In the following sections we discuss measurements that uncover the other factors influencing

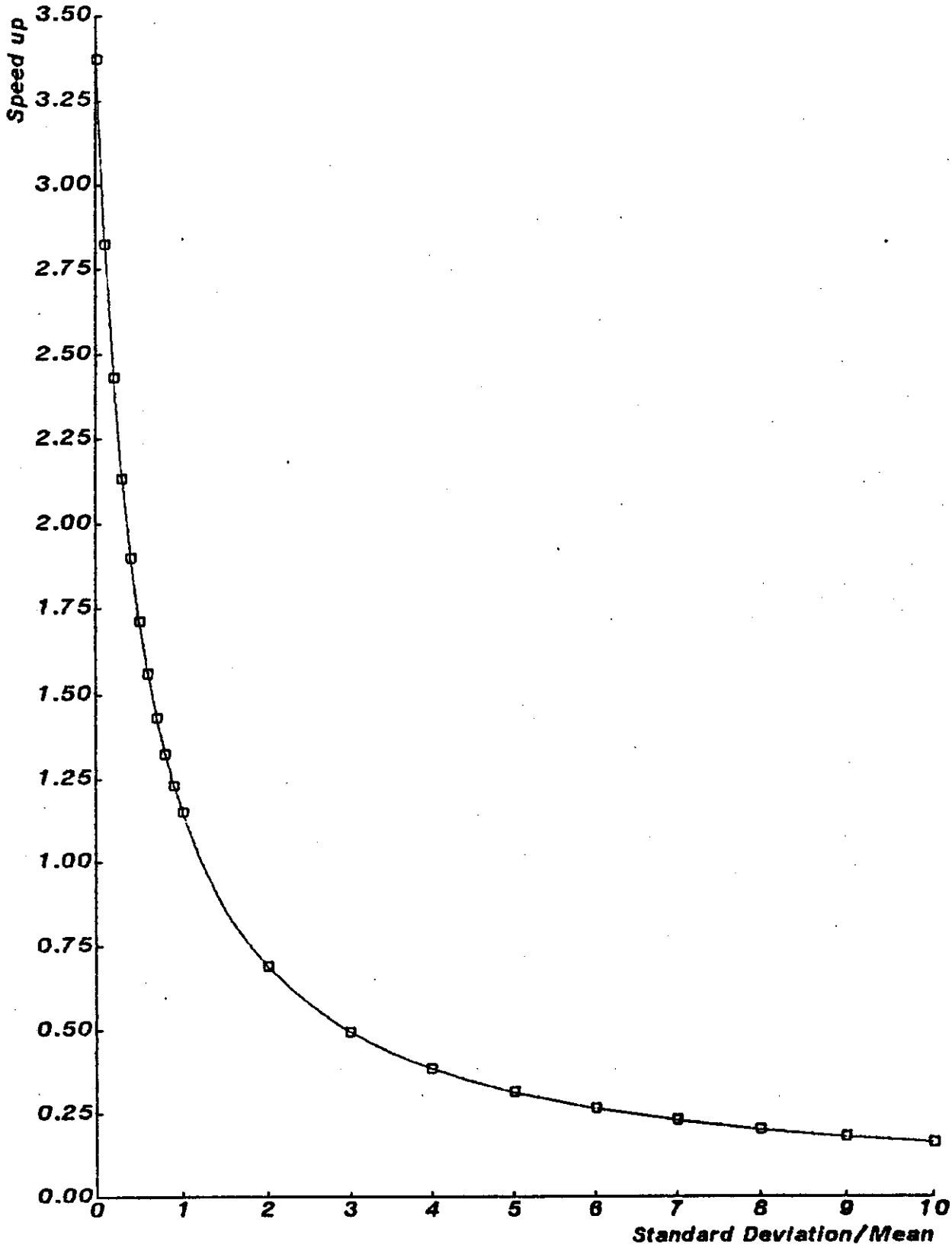


Figure 7 Speed Up vs. Coefficient of Variation for Nine Processes

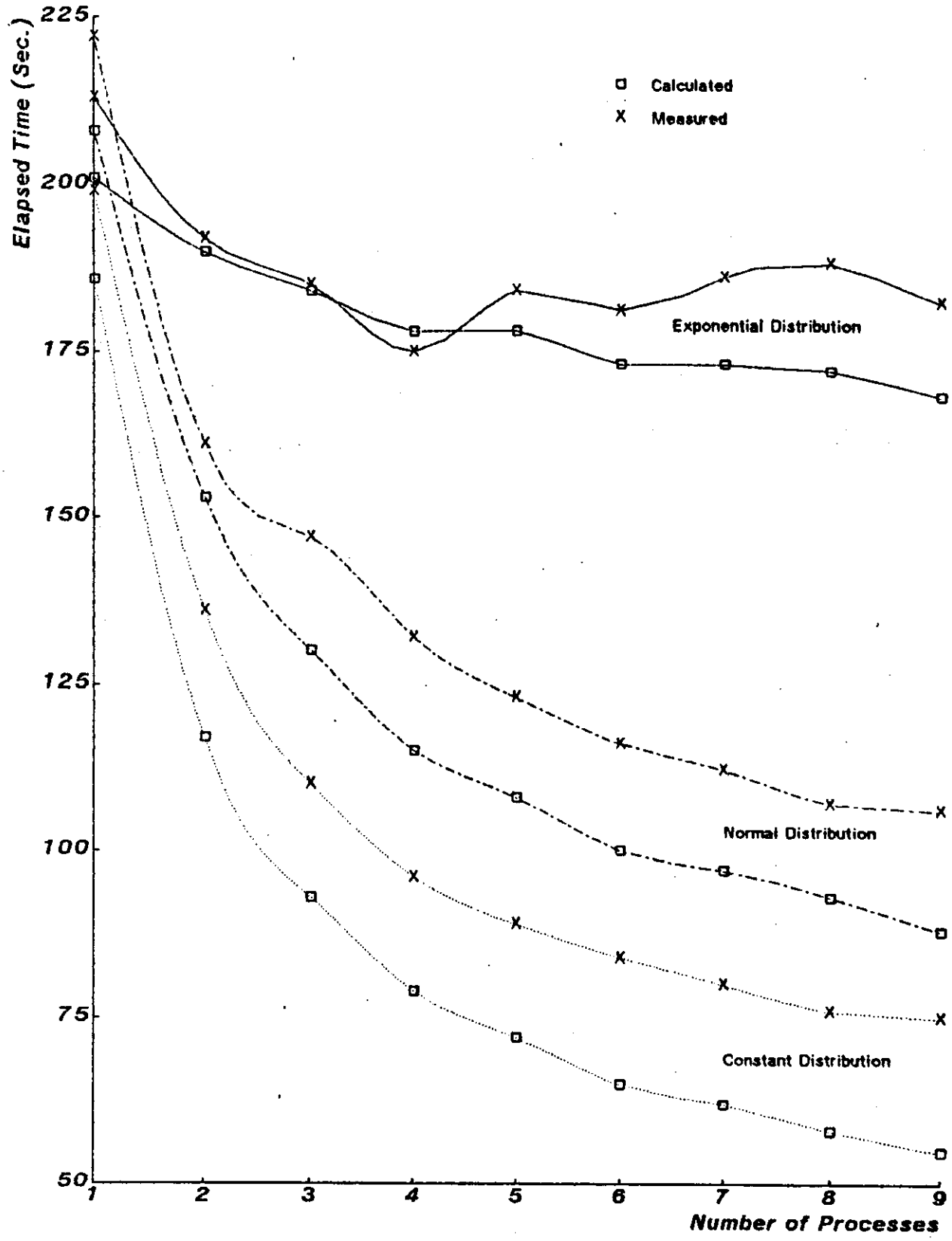


Figure 8 Measured Performance Compared to Calculated Performance

performance.

3.3 The Variation in Performance of Individual Hardware Elements

The fluctuations in performance caused by the hardware will always be present because Hydra allocates C.mmp's resources dynamically. While a user cannot accurately predict the exact performance of his processes, he can estimate the magnitude of the fluctuation in performance by measuring the variation in the performance of the individual hardware elements.

3.3.1 Processor Related Variations

C.mmp is a multiprocessor constructed from PDP-11 model 40 and model 20 minicomputers. In table 1 we have summarized the basic performance difference between the processors by comparing their execution of the F(x) calculation without the presence of Hydra. Each processor performed the calculation 100 times in the same memory port. The number of MSYN's[1] was counted and the elapsed time measured. These figures appear in the first and second columns. The third column of figures is the processor speed relative to Pc[0].

Pc	Model	Elapsed Time (sec.)	kMsyn's/sec	Relative to Pc[0]
0	11/20	15.559	443.3	1.000
1	11/40	10.413	662.4	1.494
2	11/40	9.985	690.8	1.558
3	11/40	9.745	707.8	1.596
4	11/20	16.144	427.2	0.963
5	11/40	10.060	685.7	1.546
6	11/40	10.238	673.7	1.519
7	11/40	9.829	701.8	1.582
8	11/20	14.867	463.9	1.046
9	11/40	10.022	688.3	1.552
10	11/40	10.173	678.0	1.529
11	11/40	10.001	689.7	1.555
12	11/40	10.129	681.0	1.536
13	11/40	10.005	689.4	1.555
14	11/20	14.965	460.9	1.039
15	11/20	14.999	459.9	1.037

Table 1 Speed Variations Among C.mmp's Processors

Naturally, a process on an 11/40 should execute faster than a similar process on an 11/20.

[1] MSYN is the DEC name for the signal that indicates a request is being made for the Unibus™. Since only the processor is making requests the number of MSYNs is the number of memory requests made by the processor.

Notice that even among processor of the same type there can be more than a 5% difference in speed.

Because two types of processors are used in C.mmp, the strategy of dynamically assigning processes to processors is complex. It is not sufficient to schedule a "ready" process to the best processor available. The following scenario demonstrates why.

Suppose the rootfinding processes are performing their function evaluations and are finishing at random times. After several have finished one would expect to find some idle 11/40's and computing 11/20's[1]. A good scheduler should handle its resources better. The idle 11/40's should "steal" the processes computing on the 11/20's. Naturally, this philosophy assumes that a *context swap* can be performed quickly. Process stealing is the scheduling policy on C.mmp.

3.3.2 Memory Related Variations

3.3.2.1 Technology Differences

C.mmp's centrally located primary memory is also a source of fluctuation in performance. The memory is divided into 16 modules, or banks. Each bank can service memory requests independently. However, the relative speeds of the banks are different because they contain different types of memory. At the time of this study, five banks contained semiconductor memory and 11 contained magnetic cores. Table 2 summarizes the speed differences of the memory banks. In this experiment Pc[15] performed the F(x) calculation 100 times in each memory bank. The elapsed times appear in the table.

[1]During the course of our study the number of processors running in the system varied day to day. The processor configuration during the experiment with the synthetic rootfinder was 10 PDP-11/40's and 3 PDP-11/20's. Since we never used more than nine processors to perform the F(x) calculation, all of our processes ran exclusively on the 11/40's. However, the problem is real. If we could have incorporated more than ten processes into the rootfinding procedure we would have had to deal with it. Later experiments in this paper measure the impact of the non homogenous processor configuration as the number of available 11/40's frequently was less than nine.

Mp	Technology	Time (sec.)	kMsyn's/sec	Relative to Mp[0]
0	core	15.243	452.5	1.000
1	core	14.943	461.6	1.020
2	core	15.127	456.0	1.007
3	core	14.999	459.9	1.016
4	core	15.087	457.2	1.010
5	semiconductor	15.950	432.4	0.955
6	core	15.272	451.6	0.998
7	core	15.402	447.8	0.989
8	semiconductor	15.887	434.2	0.959
9	semiconductor	15.858	434.9	0.961
10	semiconductor	15.860	434.9	0.961
11	semiconductor	15.855	435.0	0.961
12	core	15.070	457.7	1.011
13	core	15.155	455.1	1.005
14	core	15.034	458.8	1.013
15	core	15.013	459.4	1.015

Table 2 Speed Variation among C.mmp's Memory Banks

Even among memory banks of the same technology, speed varies. These variations are small however, and are caused primarily by variations in the length of cable connecting a memory bank to the crosspoint switch and in the timing circuitry for individual memory modules.

3.3.2.2 Memory Bandwidth and Memory Interference

The previous experiments show the rates at which individual processors and memories can communicate. Another important characteristic is the maximum bandwidth of a memory bank. This rate will determine how many processors can compete for cycles in a single memory bank before the bank is saturated with requests. In this experiment all of the processors simultaneously executed the tight loop in the same memory bank. Two banks of different types were chosen to be representative of their respective technologies.

The results in table 3 indicate that performance degradation will occur if more than two or three processors are competing for cycles in a memory bank. This result implies that sharing code, a common practice to conserve memory space, will result in slower execution.

Semiconductor	$1.49*10^6$	memory refs/sec.
Core	$1.71*10^6$	memory refs/sec.

Table 3 Maximum Memory Bandwidth

In tables 4 through 6 we illustrate the performance degradation that results from sharing code. All of the measurements were performed on Pc[0]. In each case 100,000 total cycles were sampled. The first column, Memory Cycle Length, is the elapsed time from MSYN to SSYN[1], a complete memory cycle.

Table 4 is the control sample where we monitored the memory accesses while the system was idle. Although the vast majority of cycles were in the 500*ns.* to 1*us.* range there were some cycles that were greater than 14*us.* This difference occurs because a processor that doesn't have a process to execute runs a task called the "idle job". The idle job consists of a WAIT instruction followed by the code that checks to see if a process is available to execute. This piece of code contains a critical section guarded by a mutual exclusion busy-wait loop. Since all of the processors are sharing this code and trying to gain exclusive access to the critical section, a great deal of memory contention occurs and the memory cycle lengths grow longer. We will use this table to compare the performance of the rootfinding processes when they execute from one common code page and when each has a private code page.

Table 5 contains the results for when each of the processes executes from a private code page. Comparing this table to table 4 we make two observations: while the average memory cycle length has increased slightly, relatively little difference exists between the two tables; the one category where a noticeable change does occur is the long (> 5.0 *us.*) cycles. Less than half as many long cycles now occur because the processors are kept busy executing the rootfinding processes.

Compare these two tables to the results in table 6 where all of the processes share one common code page. Again we make two observations: the average memory cycle length has dramatically increased by 300%; more important still is that the percentage of long cycles (> 5.0 *us.*) has increased from .086% in table 4 to 15.6%, over two and one-half orders of magnitude more. This degradation in the basic cycle time will offset and eventually reverse speedup obtained by incorporating multiple processes in the rootfinding procedure.

---

[1]SSYN is the DEC name for the signal that indicates the completion of a bus transfer. It is the signal the memory module uses to tell the processor that the memory access is completed.

MEMORY CYCLE LENGTH	READ	READ-PAUSE	WRITE	WRITE-BYTE
0 - 0.5	0	0	0	0
0.5 - 1.0	65652	7787	14089	902
1.0 - 2.0	9470	1926	8	0
2.0 - 5.0	63	6	2	0
5.0 -14.0	63	6	10	0
14.0-50.0	5	2	0	0
> 50.0	0	0	0	0

Table 4 Histogram for Idle System

MEMORY CYCLE LENGTH	READ	READ-PAUSE	WRITE	WRITE-BYTE
0 - 0.5	0	0	0	0
0.5 - 1.0	65827	7461	11024	822
1.0 - 2.0	12705	1133	38	0
2.0 - 5.0	894	54	10	0
5.0 -14.0	28	3	0	0
14.0-50.0	1	0	0	0
> 50.0	0	0	0	0

Table 5 Histogram with Private Code Pages

MEMORY CYCLE LENGTH	READ	READ-PAUSE	WRITE	WRITE-BYTE
0 - 0.5	0	0	0	0
0.5 - 1.0	52784	6504	9404	761
1.0 - 2.0	10810	689	102	0
2.0 - 5.0	3059	201	84	0
5.0 -14.0	14291	843	287	0
14.0-50.0	174	4	3	0
> 50.0	0	0	0	0

Table 6 Histogram with Common Code Page

Figure 9 captures the impact of the finite memory bandwidth problem on the rootfinding procedure. We have graphed the elapsed time to locate 50 roots *vs.* the number of processes for two implementations of the rootfinding procedure. The dashed curve results when a single copy of the code page is shared. The solid curve is the performance when the cooperating processes each have a copy of the code in a private memory bank.

This graph also can provide some insight into the speed *vs.* space tradeoff. If we compare the speedup over the single process instantiation for both the shared and no-sharing versions of the rootfinder, we find that the no-sharing version has a maximum speedup of 2.60 using nine processes while the shared version's performance peaks at 1.53 using three

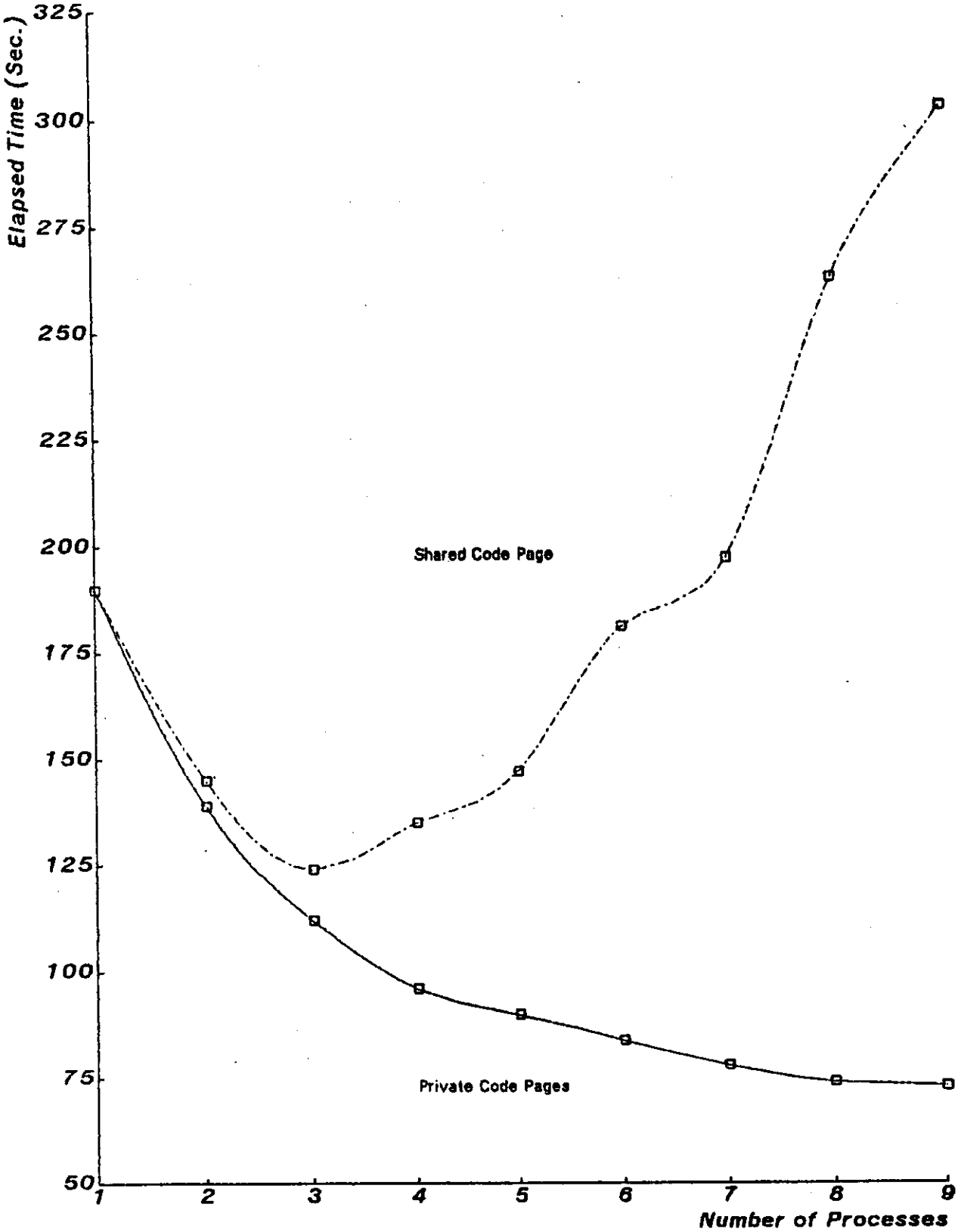


Figure 9 Performance Degradation Due to Finite Memory Bandwidth

processes. Neglecting the single global data page we have a achieved a 170% increase in speed at the cost of a 300% increase in size. In this study memory is plentiful and we squander space for speed.

One solution to the speed *vs.* size tradeoff is to interleave the central memory on the low order bits rather than the high order bits. This solution would tend to scatter memory requests more evenly across the 16 banks. To maintain availability it might be necessary to organize the store as four banks of 4-way interleaved memory. A second solution is to give each processor a cache to work with. This solution is currently being implemented on C.mmp.

3.4 Operating System Related Performance Fluctuations

3.4.1 Introduction

The operating system also perturbs the performance of the rootfinding procedure. Although C.mmp was intended to be a multi-user multi-programming facility, it is possible to use the machine in a dedicated single user mode. In this mode of operation, the user can minimize any interference from Hydra by simply not doing anything that requires operating system assistance. Most of the measurements in this study were made in this way. However, certain functions, i.e. scheduling of processes and I/O interrupts, must be performed by Hydra and cannot be avoided. The contribution to performance fluctuation from these basic operating system functions is measured and discussed in the following sections.

3.4.2 The Kernel Tracer

The Kernel Tracer is a software monitor that can obtain information about significant activity on C.mmp under the Hydra operating system. Since it is a software monitor, the Tracer does perturb the timing data it attempts to measure. However, this perturbation can be compensated for in the post-processor software.

The Tracer can monitor such things as: context swaps which occur when a processor changes from executing one process to executing another, semaphore activity, process starts and stops, operating system requests (Kernel Calls) and a multitude of other events. Events defined by user programs also may be traced.

The data is collected in real time and later processed off-line. Numerous post-processing programs produce various forms of output: process or processor dumps, time-line execution histories, and various statistical analysis packages.

All of the Tracer data that follows is in the form of a processor time-line execution history.

We use various symbols in the trace to encode events in order to compact the data. Table 7 contains these symbols and their meanings. Each row of the trace represents the activity on a processor. The time in seconds appears along the bottom edge. We will discuss in detail the first trace that captures the impact of I/O interrupts on performance.

3.4.3 I/O Devices and Interrupts

Random interrupts from I/O devices and processors contribute to performance fluctuations in the rootfinder processes. Unlike the memory, I/O devices are not centrally located and accessable through an $n \times m$ crosspoint switch. Devices are associated with a particular processor. Thus, for example, a read or write from a disk on Pc[0]'s Unibus must be performed by processor 0 regardless of which processor initiated the request. Since interrupts are serviced by stealing cycles from the currently executing process, large fluctuations in compute times can be found for processes running on processors with I/O devices.

In figure 10, interrupts associated with I/O perturb the performance of the rootfinding processes. C.mmp's processor configuration during this trace was Pc[0, 3, 4, 5, 6, 7, 8, 9, 11, 12, and 13]. The processors appear from left to right as columns of the trace. Pc[0, 4, and 8] are PDP-11/20s and the rest are PDP-11/40s. Processes(35, 43-50) are the nine rootfinding processes. Process 29 and the DAEMON are other processes that happened to be awake at the time. These two processes are doing things that cause a substantial amount of I/O. The following discussion describes how this I/O activity perturbs the rootfinding processes.

A previous iteration finishes at 0.612 seconds into the trace. Process 50, P(50), on Pc[11] was the last to finish its calculation (the activity on Pc[6] is P(29)) and begins to wake its sleeping companions by unlocking their semaphores. One by one the processes wake up and begin to perform the next iteration. P(50) finishes waking up all the processes (P(49) was the last to wake up at .641) and begins its own function evaluation. One by one the processes finish their calculations and post their results, after which they "*P*" their semaphores and wait for the beginning of the next iteration. When they block on the semaphore they are removed from the processor (e.g. CSW for P(45) on Pc[5] at .700). Four of the processors have large chunks of time shaded between brackets. This shading and brackets denotes an interrupt service routine performing I/O to a device on that Pc's Unibus. Interrupt service routines can consume between 1 and 15 milliseconds of time. This causes the rootfinding process on that Pc to arrive at the synchronization point late, thus lengthening the STAGE time.

PROCESS N	: PROCESS #N IS RUNNING
- CSW -	: A CONTEXT SWAP
IOT #X	: SPECIAL TYPE OF KERNEL KALL
KALL #X	: KERNEL KALL #X
RET X	: RETURN VALUE FROM A KERNEL KALL
n [	: START OF AN INTERRUPT AT LEVEL N
I	: INTERRUPT SERVICE ROUTINE EXECUTION
]	: END OF AN INTERRUPT
EVENT X	: USER DEFINED EVENT X OCCURS
P	: P OPERATION ON A SEMAPHORE
V	: V OPERATION ON A SEMAPHORE
DAEMON	: OPERATING SYSTEM PROCESS
|||||	: IDLE TIME

Table 7 Tracer Symbols

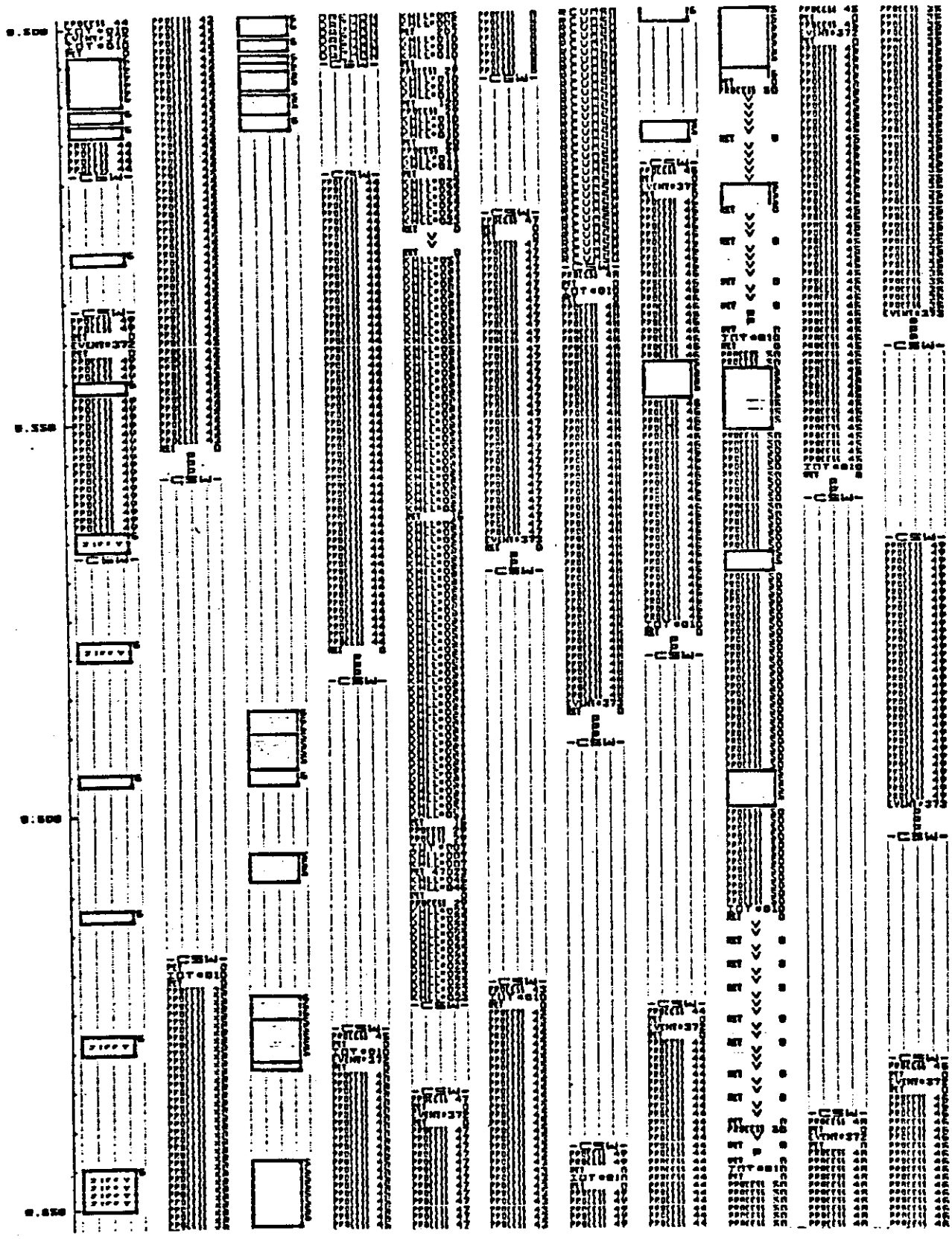


Figure 10a Perturbations from Interrupts

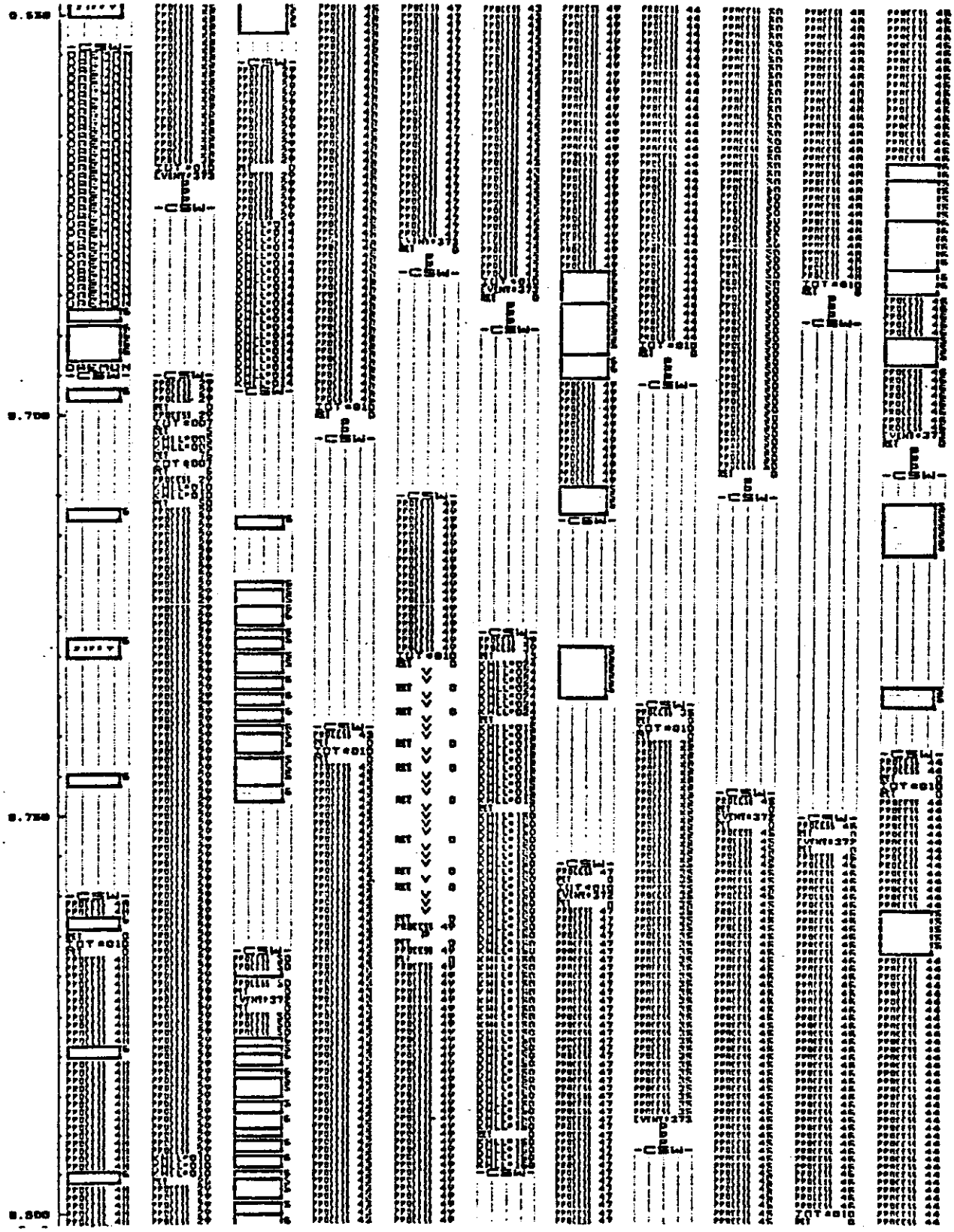


Figure 10b Perturbations from Interrupts

For example, P(49) on Pc[8] is interrupted at .681 for 13 milliseconds and then again at .707 for 4 more milliseconds. Notice however, that P(49) on Pc[8] switches to Pc[6] at .709 and finishes its function evaluation at .728 uninterrupted. Since it is the last process to finish, it assumes the jobs of finding the new root containing sub-interval and dispatching the processes to perform the next iteration.

In this example the interrupted process was delayed enough to become the last process to finish, thus lengthening the STAGE time. However, P(46) on Pc[13] was also interrupted during its function evaluation for approximately 21 milliseconds yet it was not the last to finish and did not cause the STAGE time to lengthen. This is another advantage the multiprocess implementation of the rootfinding procedure has over its uniprocess counterpart. In the single process instantiation the interrupt time is additive and each occurence lengthens the iteration. In the multiprocess version, only the interrupt time associated with the last process to finish is additive.

3.4.4 Kernel Processes and Special Functions

Operating system requests are frequently handled by special high priority Kernel processes and as such perturb the cooperating rootfinder processes by stealing processors. Of particular interest are the processes that perform scheduling. Because synchronization of communicating processes can involve rescheduling the processes, the special scheduler processes can become bottlenecks causing performance degradations.

During the trace of figure 11, C.mmp's processor configuration was Pc[0, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, and 13]. Of these, 4 and 8 are 11/20's (so is Pc[0]) and are the third and seventh blank columns with no execution history. Since enough processors of the prefered (11/40) type were available the 11/20's were never used. Similarly Pc[12] was not needed.

In this trace processes 18, 19, 20, 21, 22 are rootfinding processes. Processes 1 and 2 are Kernel scheduling processes, and process 14 is the Tracer process.

P(22) on Pc[10], the last process to finish the previous function evaluation, initializes the necessary parameters for the next iteration. At 285 ms. into the trace (.285) it begins to *V* its sleeping companion processes, and at .309 it begins its own function evaluation (event #372).

Meanwhile P(2) on Pc[6], scheduling process, wakes up CSW at .293 and begins to perform the task of actually waking up the processes that process 22 has just *V-ed*. It is a relatively painful task involving several semaphore operations and several Kernel calls per process.

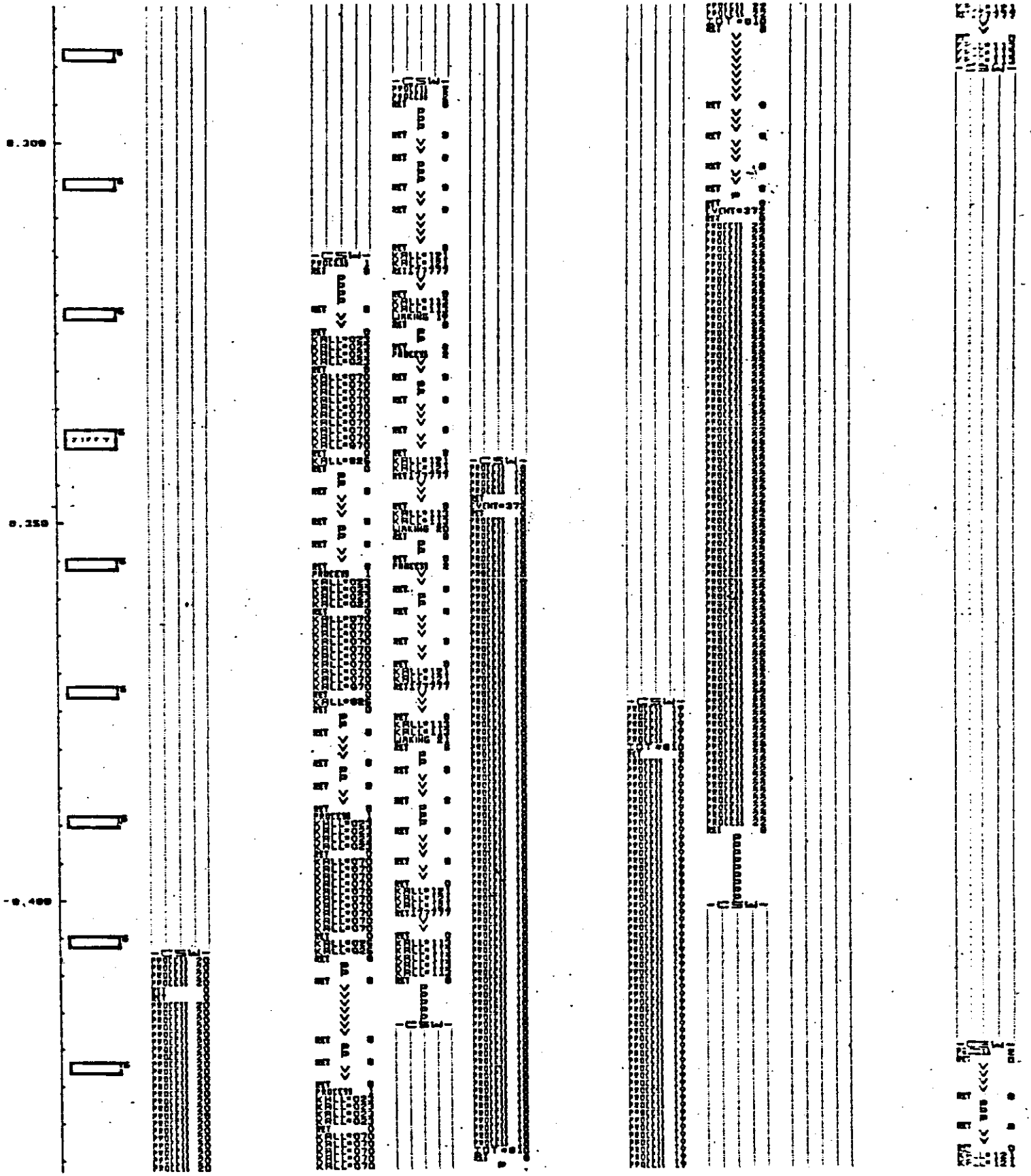


Figure 11a Perturbations Induced by Operating System Processes

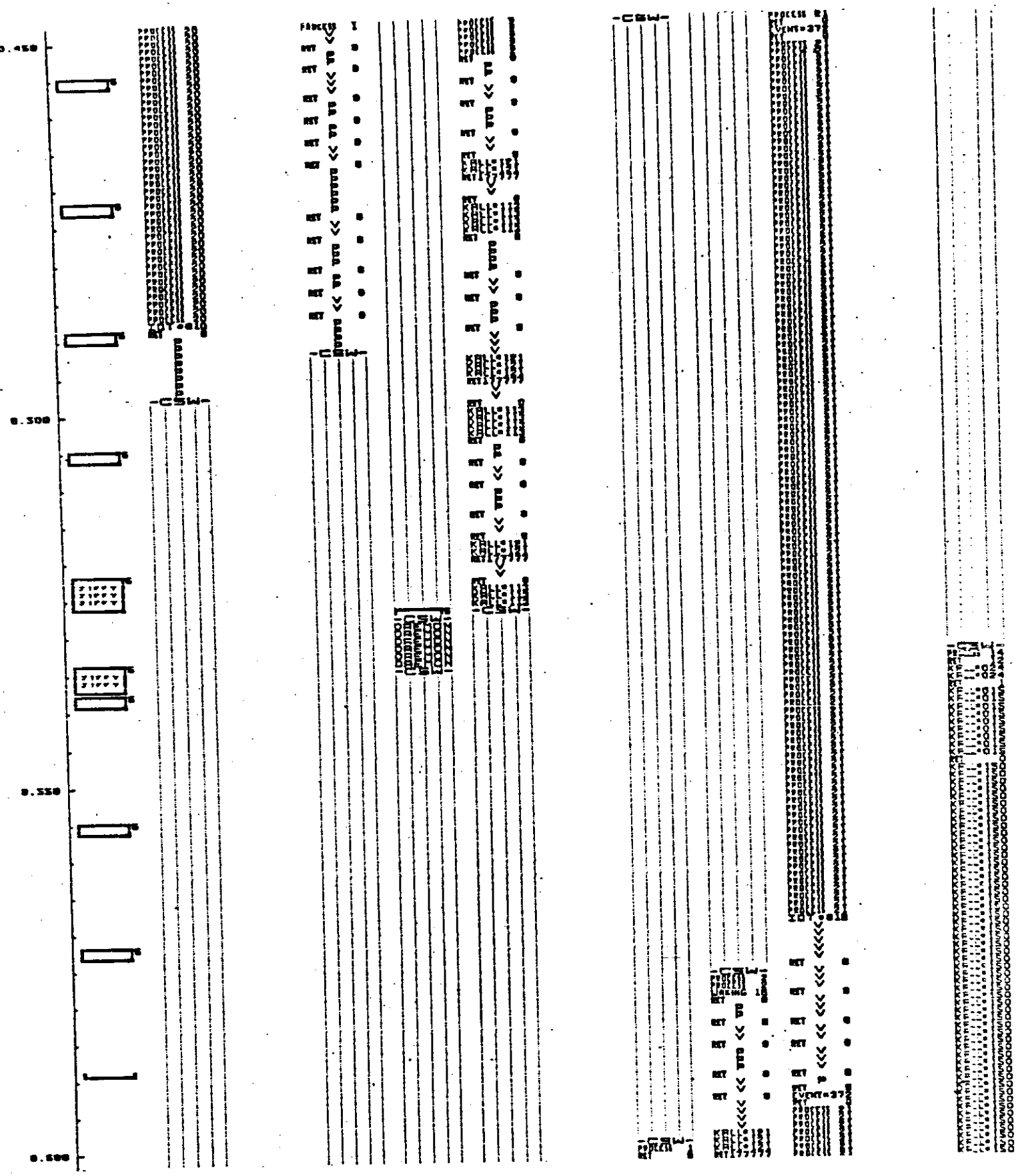


Figure 11b Perturbations Induced by Operating System Processes

Finally process 18 (the first to be *V-ed*) wakes up and begins its function evaluation at .348, approximately 60 ms. after process 22 performed the *V* operation.

To expedite the costly wake up procedure, processes 1 and 2, the scheduling processes, cooperate to start and to stop the rootfinding processes. Moreover, by the time they get around to starting process 21, the last process that is to wake up, three of the other rootfinding processes have already finished their function evaluations and have gone back to sleep (*P* followed by CSW). A full 130 ms. have transpired since process 22 performed the *V* to wake process 21.

Another side effect related to the operating system that can affect the performance of cooperating processes is the round-robin scheduling of processes under Hydra. This traditional policy is implemented using the notion of "time-sliced" intervals of execution to provide equal service to all tasks. Occasionally a process exhausts its time slices and must ask for more. This request can take more than 150 milliseconds to execute. Whether or not the time-slice end anomaly will perturb the performance of the cooperating processes depends upon the average duration of the function evaluation and the frequency of the time-slice end condition. In this study a process must consume 10 one half second slices before encountering the time-slice end condition.

Figure 12 is the distribution of the elapsed time to perform an F(x) calculation in the presence of Hydra. The long tail in the distribution is a result of the time-slice end condition occurring for the process performing the function evaluation.

3.5 Summary

The sources of performance fluctuation we have discussed can be classified into one of three types-- application, hardware, or operating system related. In the table below we rank the sources of perturbation by their potential for causing performance fluctuations. Each source is measured and the observed range calculated by dividing the maximum measurement by the minimum observed value. The larger the range, the more potential for performance fluctuation.

In the next section we eliminate several sources of perturbation in order to measure the performance of various synchronization primitives. We do this by carefully selecting processors and memory banks to execute the rootfinding program.

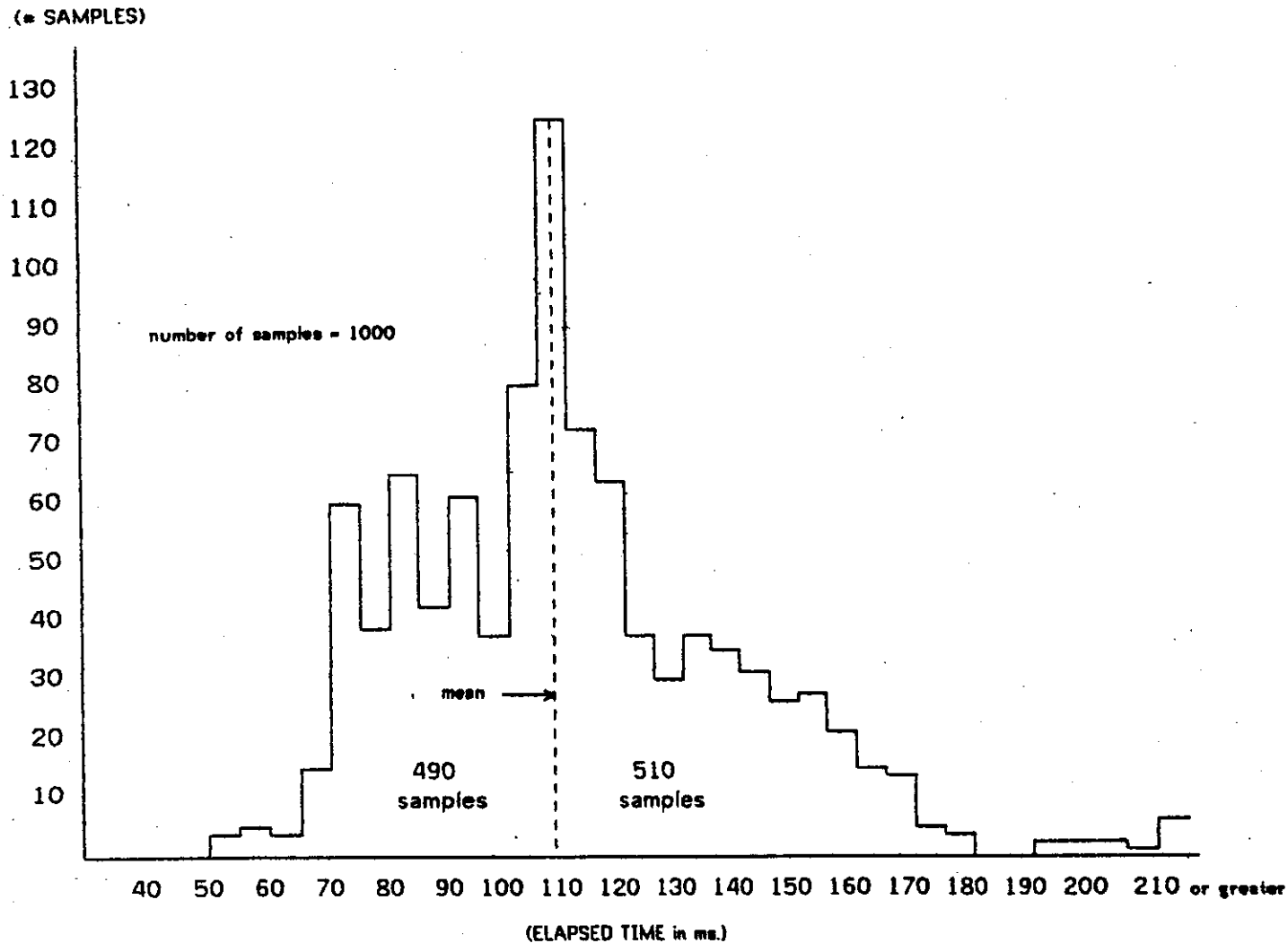


Figure 12 Distribution of the Time to Calculate F(x) in the Presence of HYDRA

Rank	Type	Source	Measurement	Range
1	Application	F(x) Calculation	Function Evaluation	1 : 3.4
2	Hardware	Memory Contention	Average Cycle Length	1 : 3.0
3	Operating System	Kernel Processes	Bottlenecking of Scheduling Processes	1 : 2.8
4	Hardware	Processors	Speed	1 : 1.6
5	Operating System	I/O Devices and Interrputs	F(x) Calculation Degradation	1 : 1.3
6	Hardware	Memories	Speed	1 : 1.07

Table 8 The Sources of Performance Perturbation

4. The Effect of Sychronization on Performance

4.1 Introduction

Newell and Robertson[1975] identified seven programming issues for multiprocessor computer systems. One of these, synchronization, is a fundamental problem with cooperating processes in any environment. Since it has great impact on the performance of a parallel algorithm, we will measure the perfoqmance and discuss the tradeoffs of the various synchronization mechanisms available to the C.mmp user.

Until now, we have used a very simple form of "busy-waiting" loop to synchronize the cooperating processes. Although synchronization using this method is extremely fast, undesirable side effects can cause serious performance problems. We will discuss several alternative synchronization mechanisms, describe their operation and side effects, compare their performance in the context of the rootfinding algorithm, and present the range of usefulness for each.

4.2 Description of Synchronization Primitives

We first examine the nature of the synchronization problem for the rootfinding processes. In figure 13 we present a more detailed view of the STAGE time and in particular focus on the mechanics of synchronization. The segment labeled FIND is the time spent locating the new root containing sub-interval. The *V(i)'s* correspond to waking up each of the rootfinding processes. One quickly notices that the overhead of synchronization can be a significant part of the STAGE time in certain instances. Because we have used a *spin lock*, a form of busy waiting, to synchronize the processes, the overhead of synchronization has been negligible. However, it is not always desirable to implement synchronization with this mechanism.

4.2.1 The Spin Lock

Of the three synchronization primitives considered in this study, the spin lock is the most rudimentary. This primitive is actually implemented independently of any Hydra support and is only a tight loop in which the process continually tests a semaphore until it can set it successfully. The *P* and *V* operations are the following PDP-11 code sequences:

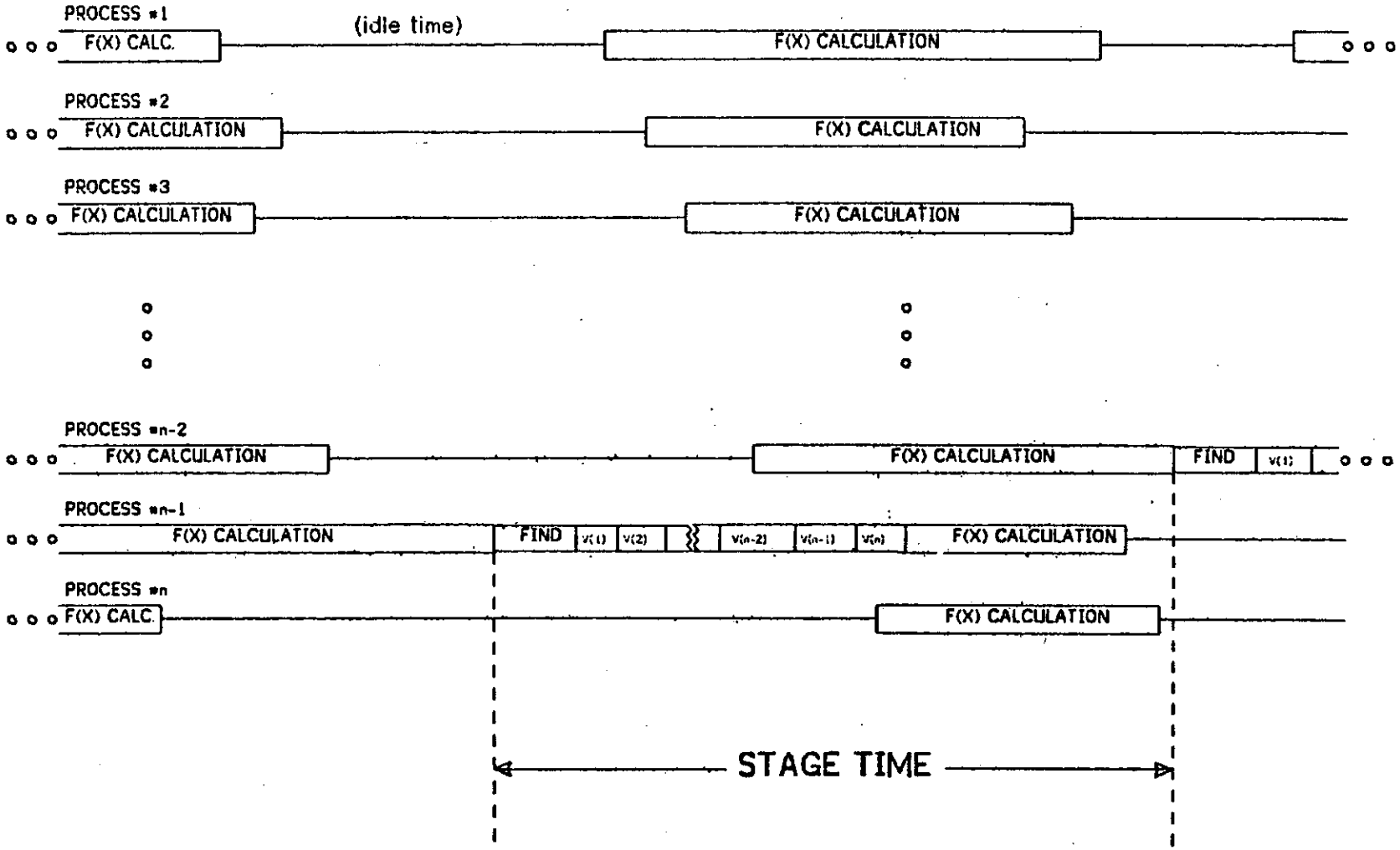


Figure 13 A Detailed View of the STAGE Time

```
P:      CMP SEMAPHORE, #1       ;SEMAPHORE = 1 ?
        BNE P                   ;loop until it is = 1
        DEC SEMAPHORE           ;decrement SEMAPHORE
        BNE P                   ;if SEMAPHORE neq 0 then go to P

V:      MOV #1, SEMAPHORE       ;reset SEMAPHORE = 1
```

The repeated polling of the semaphore, although extremely fast, has two very nasty characteristics.

The first is that when the process completes its function evaluation and starts to poll the semaphore while waiting for its counterparts for finish, the processor is not free to perform useful work.

The second major drawback is that the polling process consumes many cycles in the memory bank that contains the semaphore. As more processes finish their function evaluations and begin to poll the semaphore, the bandwidth of the memory bank is quickly consumed. The process that has its code page located in the bank with the semaphore will be competing for cycles with many busy processors. This second problem can be circumvented by inserting a tiny delay loop in the semaphore code, i.e., decrement a register to zero before checking the semaphore. This delay will decrease the frequency of memory requests in the semaphore memory bank, but not slow the sychronization primitive appreciably. However, the primary problem still remains: a "spinning" process prevents a processor from doing useful work.

4.2.2 The Kernel Semaphore

The Kernel semaphore (K-SEM) is implemented by the Hydra operating system. It is the low level synchronization mechanism used by system processes. When a process blocks or wakes up, a state change for that process is made inside the Kernel. Because it is implemented within the domain of the Kernel, the user evokes operations on the semaphore (*P* and *V*) by issuing Kernel calls. If the process blocks while trying to *P* the semaphore, the Kernel swaps the process from the processor and places the process in the semaphore's blocked-queue, where it remains until another process *V*'s the semaphore. When the process can proceed again, it is swapped back onto an available processor and continues execution from the point where it was blocked. The important attributes of the Kernel semaphore are:

- A blocked process is swapped from a processor.

- When a process blocks, its pages are kept in primary memory. Keeping the pages in primary memory ensures that the process will quickly resume execution when it is swapped back onto a processor.

- The Kernel semaphore is approximately two orders of magnitude slower than the spin lock.

4.2.3 The Policy Module Semaphore

The policy module semaphore (P-SEM) is implemented by the scheduling subsystem called the *Policy Module* (PM). This primitive is intended as the user's primary mechanism for performing synchronization.

Since the synchronization is performed within the context of a system process, more flexibility is available in handling a blocking/waking process. The first policy that was adopted to handle blocking/waking processes was the following:

- Two PM processes would cooperate to perform synchronization operations for users; one would start and stop processes and the other would handle communication between the Kernel and user.

- When a process blocked on a semaphore it would be context swapped from the processor.

- Any 'dirty' pages belonging to the process would be updated on secondary storage.

- When a process was to wake up it would be restarted by one of the PM processes after all the swapped out pages belonging to the process were brought back into central memory.

This policy has evolved into a much faster arrangement of multiple processes in the current version of the PM.

One modification to the PM that was found to improve performance substantially was to delay the updating of a process' dirty pages onto secondary storage. Often a process is blocked for very short amounts of time and will quickly resume execution after only several milliseconds of waiting for a certain condition to be true. However, when a page is to be updated onto secondary storage it is written onto one of several fixed head disks that will take at least 32 milliseconds per page. The swapping disks revolve once every 16.67 milliseconds. It takes two revolutions to update a page: one to write it out and the second to perform a read-check operation to validate the copy. Thus, it is quite possible for a process to spend most of its time blocking and unblocking if the inter-synchronization interval

is small enough. The problem would be even more severe if there were a task force of cooperating processes, e.g. the rootfinding processes, blocking and unblocking every few milliseconds.

The current version of the PM initializes the delay time parameter, ϵ, to 300 milliseconds. Table 9 is a summary of the time it takes to perform the basic semaphore operations on the various primitives.

Measurement	Spin Lock	K-SEM	PM0	PM1(ϵ=0)	PM1(ϵ=300)
Time for a process to do a V (*us.*)	30	3000	6000	5000	5000
Time till a process wakes up from a V (*us.*)	30	5000	55000	50000	13000
Time from P to CSW (*us.*)	na	3000	9000	6000	6000
Time spent in PM while waking a process (*us.*)	na	na	62000	20000	0

Table 9 Comparison of Execution Times for Semaphore Primitive Operations

4.3 The Impact of Synchronization on Performance

4.3.1 Introduction

Now that we have described the functionality and presented the individual performance statistics for the basic primitive operations, we can observe the impact of synchronization on the performance of the rootfinder. We have eliminated most of the overheads associated with synchronization by using the spin lock primitive. The remainder of the paper examines the rootfinder's performance as we employ the alternative synchronization primitives.

4.3.2 Comparison of Primitives When Compute Time ~ Synchronization Time

The first graph, figure 14, compares the performance of the various implementations of the rootfinder using different primitives to perform the process synchronization. We have plotted the elapsed time to find 50 roots as a function of the number of processes. This data was generated by the authentic, not synthetic, rootfinder. The distribution of the F(x)

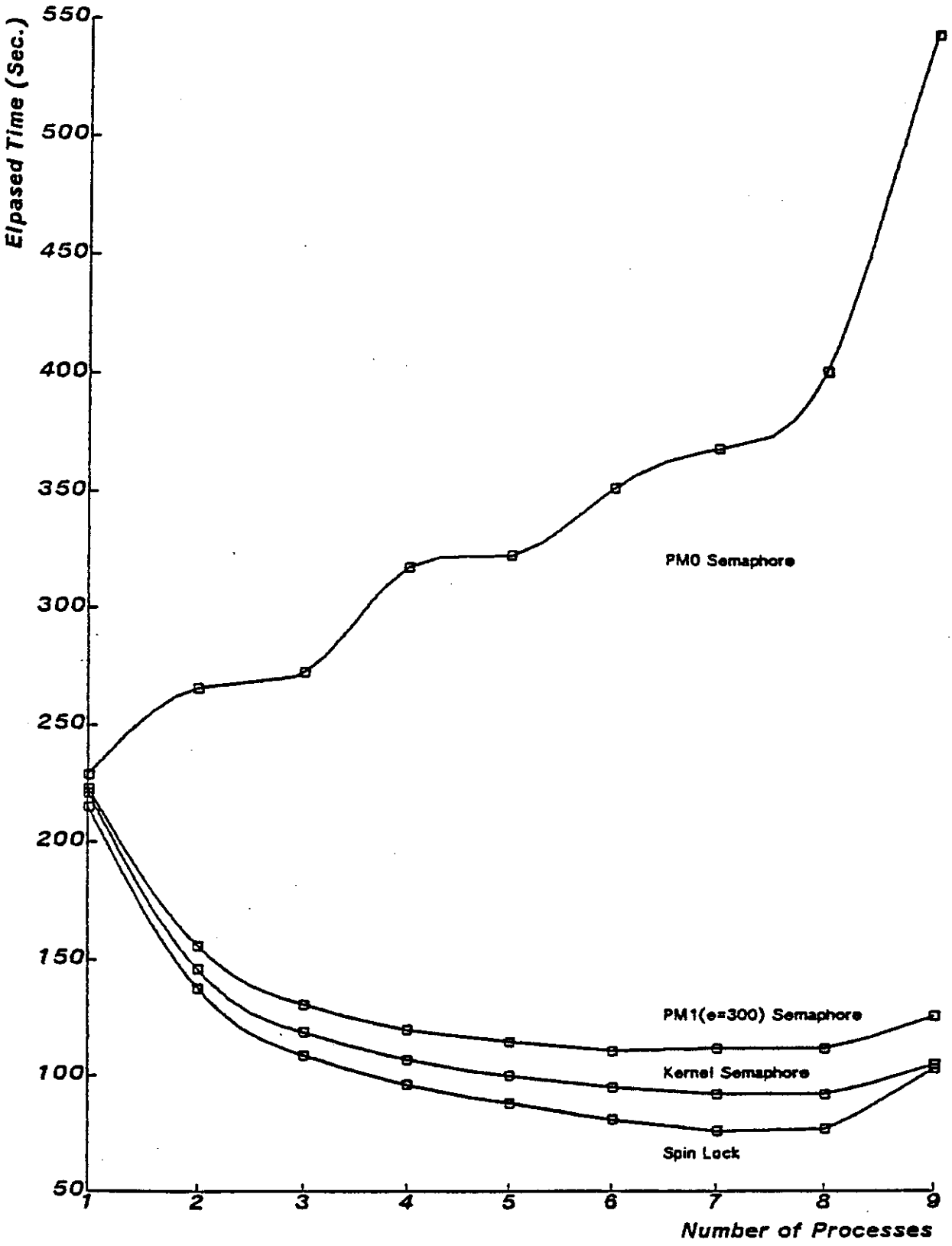


Figure 14 A Performance Comparison of Synchronization Primitives

computation is approximately Normal with mean 72 milliseconds and standard deviation 18 milliseconds[1]. We compare the performance of four alternative synchronization primitives: spin lock, K-SEM, PM1(ϵ=300), and PM0 semaphores.

The curve for the PM0 semaphore implementation exhibits degradation as we increase parallelism. The reason for this behavior is that the overhead of synchronization is greater than the average compute time. A process spends more time synchronizing than computing. In this instance we would be better off using a single process.

The curve for the PM1(ϵ=300) semaphore implementation depicts substantially better performance than its predecessor. Performance reaches a maximum speedup of 2.00 at six processes. No additional speedup is gained by employing more processes. Moreover, a noticeable degradation occurs at nine processes. This sudden degradation occurs because of the non-homogenous processor configuration (NHPC). During this experiment C.mmp's processor configuration was eight 11/40's and one 11/20. Thus, when we incorporated the ninth process, it ran on the slower 11/20 type processor. The STAGE time lengthed, thus yielding an overall slower performance.

The K-SEM implementation has its peak performance of 2.4 at eight processes. It too is affected by the NHPC problem and performance degrades slightly at nine processes. The overall performance of the K-SEM implementation is about midway between the PM1(ϵ=300) and the spin lock versions.

The spin lock implementation has by far the best speed up maximum of about 2.8 for eight processes. The NHPC problem causes a much more severe performance degradation for this semaphore than for the others[1]. The reason is that the processes blocked on the spin lock semaphore remain on their processors, whereas the other implementations free the faster 11/40 type processors to steal the process that is still running on the slower 11/20 processor.

4.3.3 Comparison when Compute Time is Much Greater Than Synchronization Time

In the previous experiment the overhead of synchronization was in some cases a considerable fraction of the STAGE time. If we make the compute time for the function evaluation much larger, thus reducing the percentage of time spent synchronizing, the

---

[1] On an 11/40 processor

[1] The PM0 implementation performance curve has a greater degradation than the spin lock version. However, the reason is not merely the NHPC problem. The primary reason is that the two PM processes that perform the semaphore operations are almost constantly running.

performance differences between the various implementations is also reduced. Figure 15 graphs performance in terms of speed up as a function of the number of processes. We used the synthetic rootfinder again to generate F(x) computations that take 375 milliseconds to compute with the distribution a constant. The dashed curve is the performance obtained using the PM0 semaphore and the solid curve the performance obtained using the spin lock.

We expected the curves to be closer together, yet the spin lock version outperforms the PM0 semaphore 2.8 to 2.1 at maximum speed up. The reason for the large difference is that the PM processes must perform the semaphore operations *serially*, each V operation taking about 55 milliseconds. Thus the n^{th} rootfinder process is not started until 55*n milliseconds into the STAGE time. In this manner the ninth rootfinder process does not complete its function evaluation until 870 milliseconds have past. Similarly, when the rootfinder processes complete their F(x) calculations, the PM processes again *serially* perform the P operations on the semaphores causing still further performance degradations.

The severe performance degradation that occurs at eight and at nine processes for the spin-lock implementation is another instance of the NHPC problem. This time, with only seven 11/40 type processors, performance peaks at seven processes, declines slightly at eight, and then plummets from a speed up of more than 2.7 to slightly more than 2.0. The performance of the two implementations is nearly identical at nine processes.

However, in figure 16, where the distribution is exponential, relatively little difference exists between the performances of the two implementations. Because the distribution of the compute phase causes the processes to arrive at random times, the PM does not become a bottleneck when the processes finish their work. When they are restarted, the last one to be started is still delayed by 55*n milliseconds. However, since the distribution is exponential, the process that must compute the function evaluation with a compute time that lies in the long tail of the distribution always finishes last. Thus the overhead of synchronization is again hidden by the MAX function that governs the STAGE time.

4.4 Summary of Results: The Useful Range for Various Semaphores

In figure 17 we have summarized the results of this investigation by graphing the useful range for each of the synchronization primitives. We have graphed the performance of the rootfinder using each primitive as we vary the size of the computation phase between synchronization points. For each point, five cooperating processes performed 1000 total function evaluations to find 50 roots. The distribution of the function evaluation was a constant and ranged in size from 2 milliseconds to 375 milliseconds.

The NO-OVERHEAD curve is the ideal performance we would see if no degradation occured

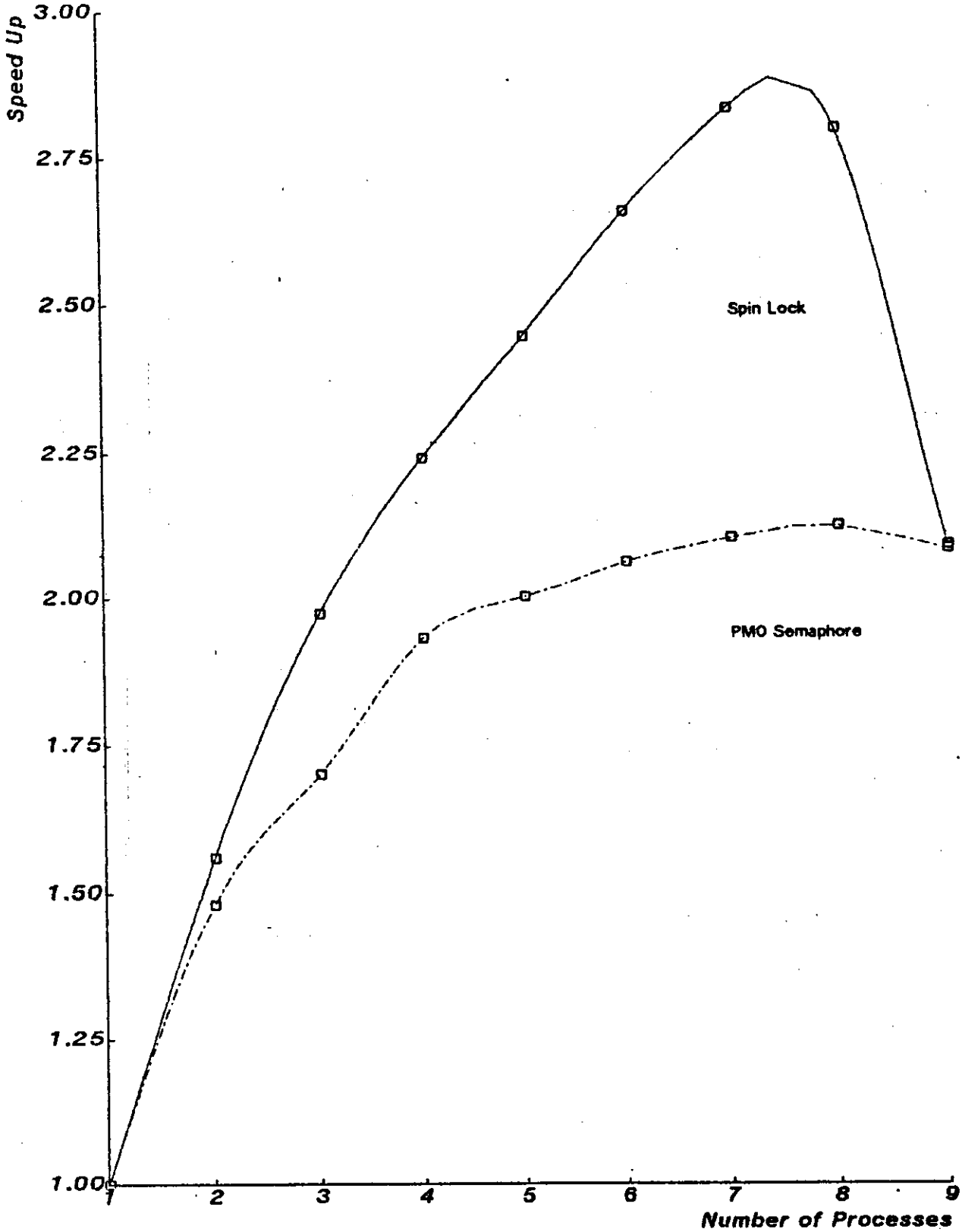


Figure 15 Comparison of Two Synchronization Primitives

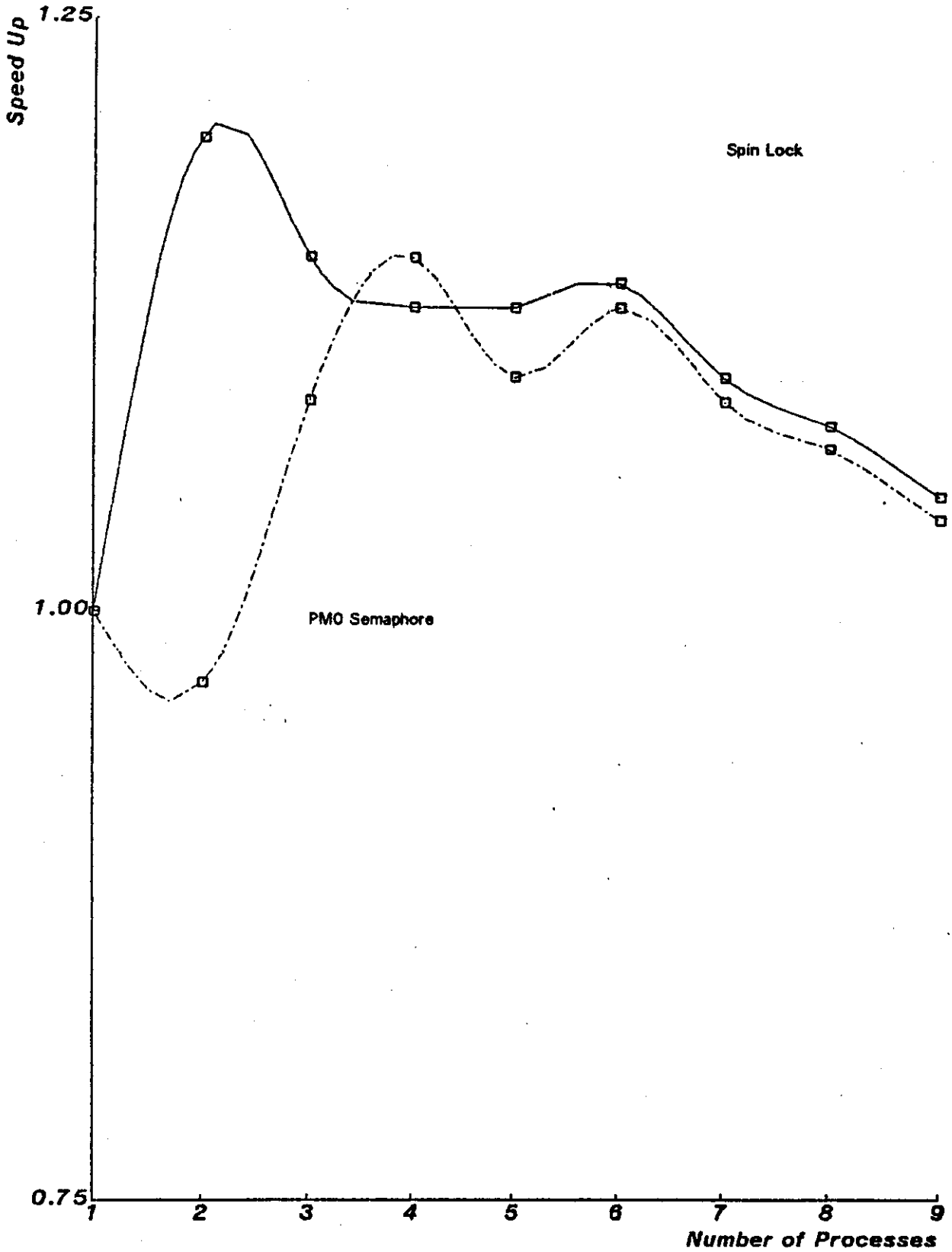


Figure 16 Comprison of Two Synchronization Primitives

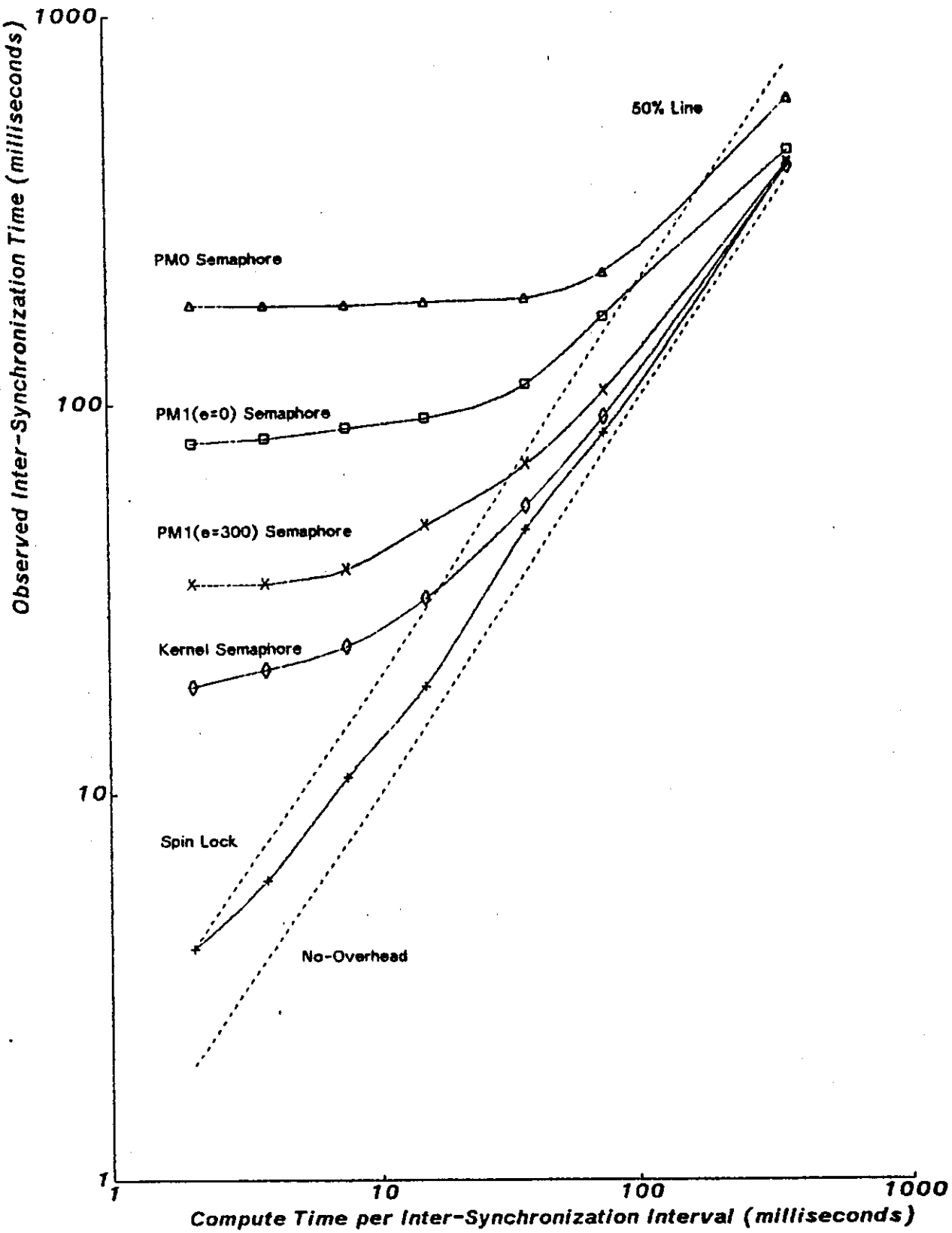


Figure 17 The Range of Usefulness for the Various Semaphores

due to hardware, operating system or synchronization overheads.

The 50% line represents our threshold for adequate performance. It parallels the NO-OVERHEAD curve but represents exactly half of the performance that would be achieved in the best case. The point at which a performance curve crosses the 50% line is the threshold of usability for that synchronization primitive.

From these results we see that the spin lock is the only primitive that performs adequately when the length of the compute phase is less than 15 ms. At the other extreme, all of the primitives with the exception of the initial version of the policy-module semaphore, become indistinguishable beyond 400 ms. In the region between these two endpoints the user can select the appropriate primitive to match the length of the computation phase. The cross-over points for the various semaphores appear in the table below.

Semaphore Type	Cross-over Point (msecs.)
Spin Lock	2
K-Sem	18
PM1(ϵ=300)	33
PM1(ϵ=0)	80
PM0	200

Table 10 Cross-over Points for the Various Semaphores

5. An Example Implementation

One technique to decompose a task for parallel execution is to portion the work into independent *partitions* for simultaneous processing. This method is applicable to problems involving the repeated evaluation of a sequence of functions on a stream of data, e.g. integer programming and matrix manipulations. The parallelism results from simultaneously performing the function evaluations on different data elements in the stream.

Two overheads are associated with decomposing an algorithm into parallel processes using the partition approach: 1) the cost of partitioning the data and 2) the cost of synchronizing the processes. To successfully capture parallelism using this approach, these two overheads must be minimized. Thus, problems involving minimum communication between the processes and a long data stream composed of independent data elements are favored as good candidates for decomposition using the partition approach.

However, not all tasks that exhibit potential parallelism are simply the repeated application of a function on a stream of data. Connected speech recognition systems exhibit a great deal of parallelism [Lesser 75], but have complex control structures that can constitute a large synchronization overhead. In order to efficiently implement algorithms of this type, it is necessary to restructure the algorithm so that the overhead of process synchronization has only minor impact on the algorithm's performance.

Often, minimizing the overhead of synchronization can be accomplished by decomposing a large, complex task into a series of smaller, simpler sub-tasks. While this introduces new synchronization points into the algorithm, it also increases the potential for parallelism if the sub-tasks can be performed simultaneously.

To demonstate the effectiveness of the partition approach we have chosen a complex task, the Harpy speech recognition system developed at Carnegie-Mellon University [Lowerre 76], for decomposition into cooperating processes. This chapter describes the algorithm, demonstrates a series of implementations, and discusses the performance that results from each refinement to the algorithm.

5.1 A Brief Description of the HARPY Speech Recognition System

HARPY is a speech recognition system that can recognize phrases and sentences from many speakers based on a finite vocabulary within a constraining task [Lowerre and Reddy 77]. Two important features of any speech recognition system are its representation of knowledge and the search and match techniques that convert the passive knowledge into an

active process for understanding the spoken utterance.

5.1.1 Representation of Knowledge

HARPY represents all legal sentences within a task in a finite state graph structure. Figure 18 is a graph of a simple grammar. The knowledge is organized as a network of nodes where each node holds a word in the vocabulary. The nodes are interconnected such that any path through the word network constitutes an acceptable sentence.

Many words have more than one pronunciation. Alternative pronunciations can be represented as a separate network of phonemes[1]. Each path through this type of network represents an acceptable pronunciation of a word. For example, the southern pronunciation of the word "tell" is an optional path through the phoneme graph in figure 19.

By replacing every node in the word network with its pronunciation network, we produce a new finite state graph, figure 20, where each path is a pronunciation of an acceptable sentence.

A separate knowledge network is compiled for each task. Pre-compiling the network eliminates the need for dynamic interpretation of knowledge during the search and match phase of the recognition process.

5.1.2 The Recognition Process

HARPY's recognition process consists of three separate phases: the pre-processing of raw speech, the heuristic search through the knowledge network, and the backtrace through the network that yields the connected sentence of speech. The heuristic search is by far the most interesting and computationally intensive phase of the recognition process, but we will include discussion of the other two phases for completeness.

The pre-processing phase starts when the utterance is input to the computer. The utterance is digitized and segmented into acoustical units, figure 21. These segments are analyzed to determine their segmental features and parameters. At this point, an attempt is made to match each segment of speech with one of the possible phonemes. Since an absolute assignment cannot be made reliably, the system calculates a match probability for each phoneme based on the acoustic information in each segment, figure 22.

The goal of the heuristic search phase is to find an optimal sequence of phonemes

[1] Phonemes are the smallest units of speech that distinguish one word from another, e.g., the "m" in mat and the "b" in bat are two English phonemes.

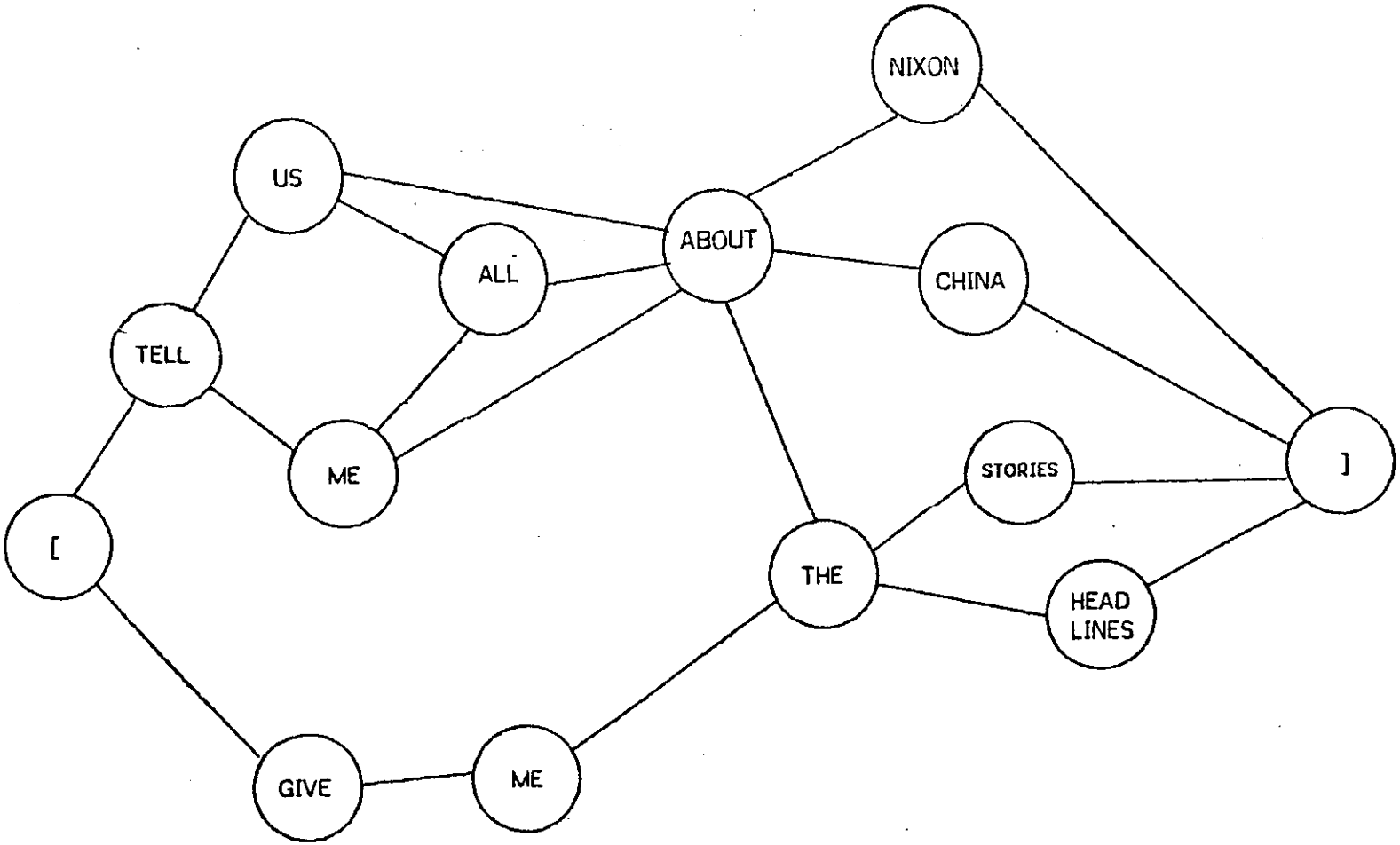


Figure 18 A Word Network for a Simple Grammar

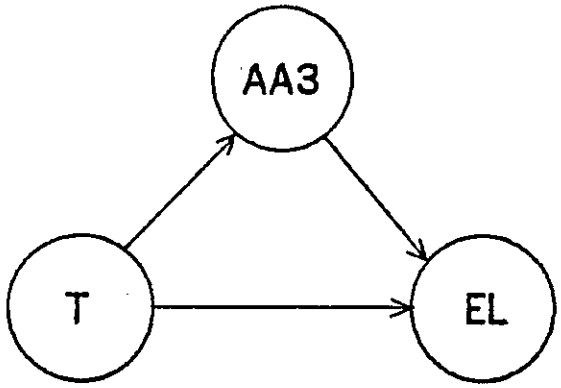


Figure 19 Pronunciation Network for the Word "TELL"

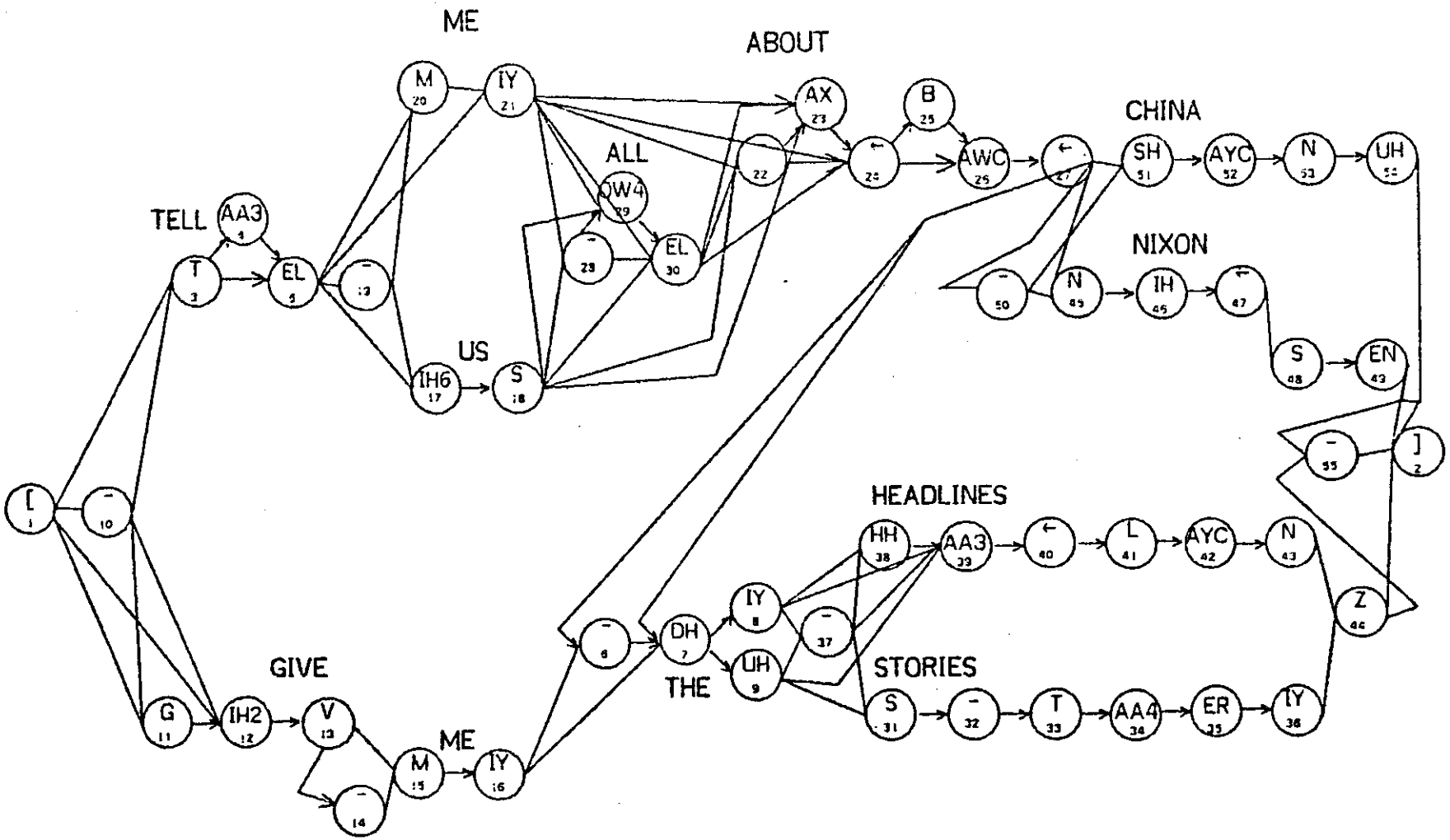


Figure 20 Pronunciation Network Incorporated Into the Word Network

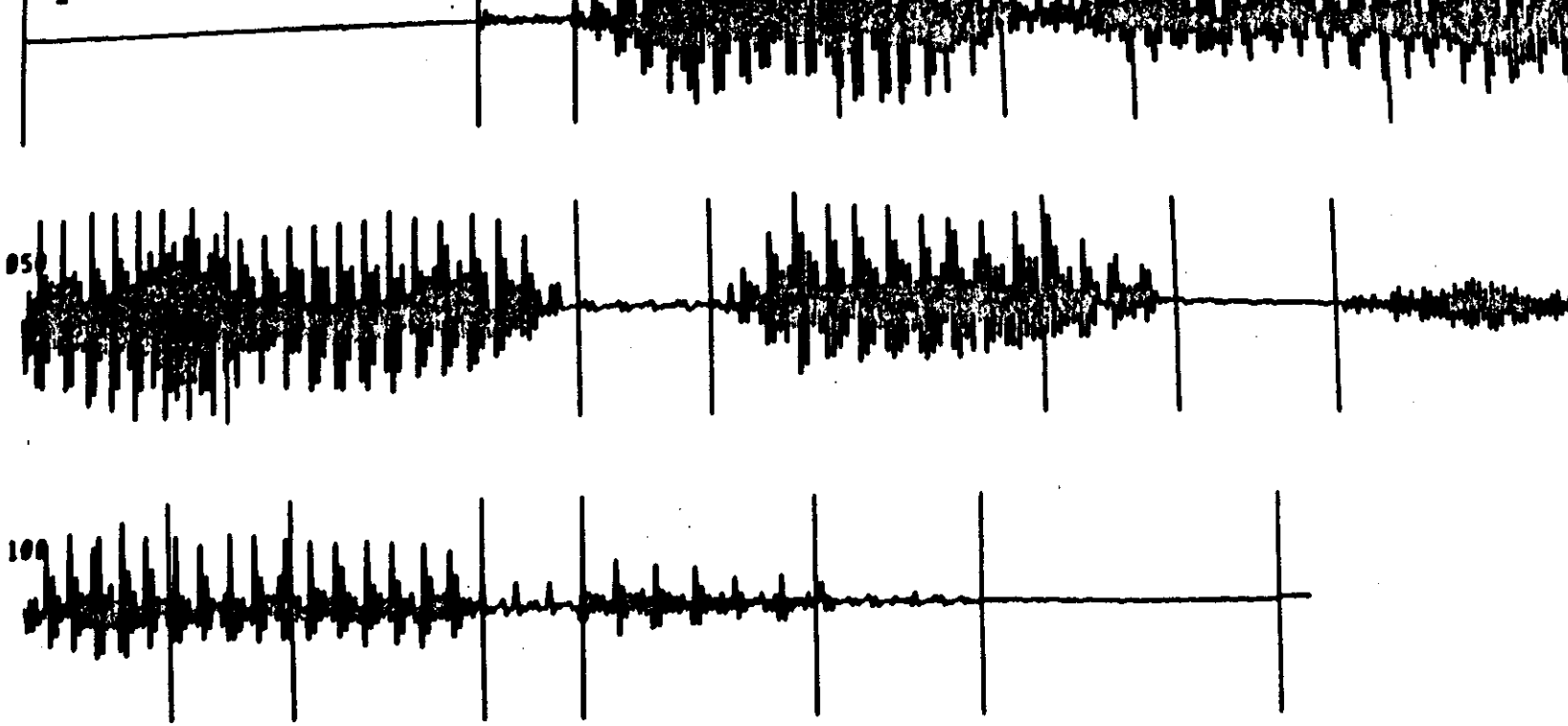
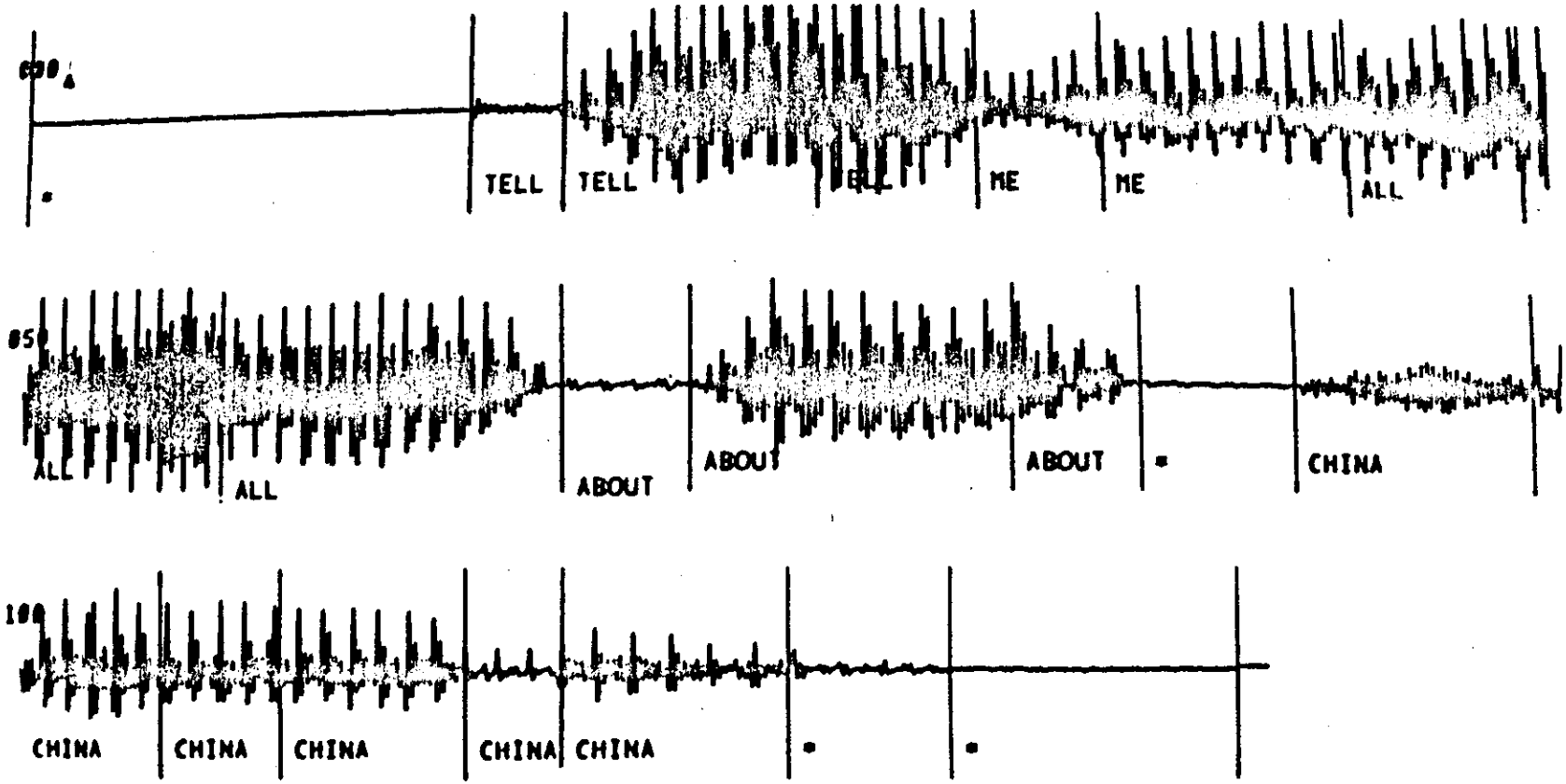


Figure 21 Digitized Speech Segmented into Acoustic Units



I heard "TELL ME ALL ABOUT CHINA"

Figure 22 Words Corresponding to Selected Phonemes

satisfying two criteria: the sequence must represent a legal path through the knowledge network, and the sequence should consist of phonemes with high acoustic match probabilities.

HARPY uses a *beam search* to locate this optimal sequence of phonemes. This technique involves searching a few of the best paths simultaneously, eliminating the need for backtracking.

The search is performed by creating and dynamically pruning a tree structure of phonemes. Each ply in the tree represents one segment of the digitized utterance.

For example, HARPY begins the search by placing all the legal phonemes for the start of a sentence in the recognition tree, figure 23a. Next, a path probability is calculated for each candidate. The path probability is a cumulative probability based on the path probability of the previous node and the acoustic match of the current node, figure 23b. The path with the best probability is determined and the remaining candidates are compared with it. Those that fall below a threshold of acceptability are pruned from the recognition tree, figure 23c.

The surviving candidates are expanded based on the information in the knowledge network and the search continues, figure 24a. The path probabilities are calculated, the best path determined, and unpromising alternatives are pruned, figure 24b. The heuristic search continues, expanding the recognition tree and saving those connections that satisfy the threshold until the end of the utterance is reached.

The final phase of the recognition process is a backtrace through the recognition tree along the path with the highest probability. This backtrace is purely a lookup operation, and does not involve any search. The final output of the backtrace is the sequence of words that correspond to the optimal path.

5.2 The Decomposition of the HARPY Algorithm

The first step in decomposing HARPY's search algorithm is to isolate sub-tasks, independent functions that operate on a data stream. No restriction exists on the number of sub-tasks that, when combined, accomplish the task. Moreover, performance may be improved by decomposing a complex sub-task into a series of simpler tasks.

HARPY is a three phase recognition system; in this study, we will decompose only the heuristic search phase since it is the most complex and computer intensive of the three. We can identify three sub-tasks in HARPY's heuristic search. The three sub-tasks and the names of the routines which perform them appear below and in figure 25.

CHECKNEXT A candidate state is expanded into a list of successor states. Each item

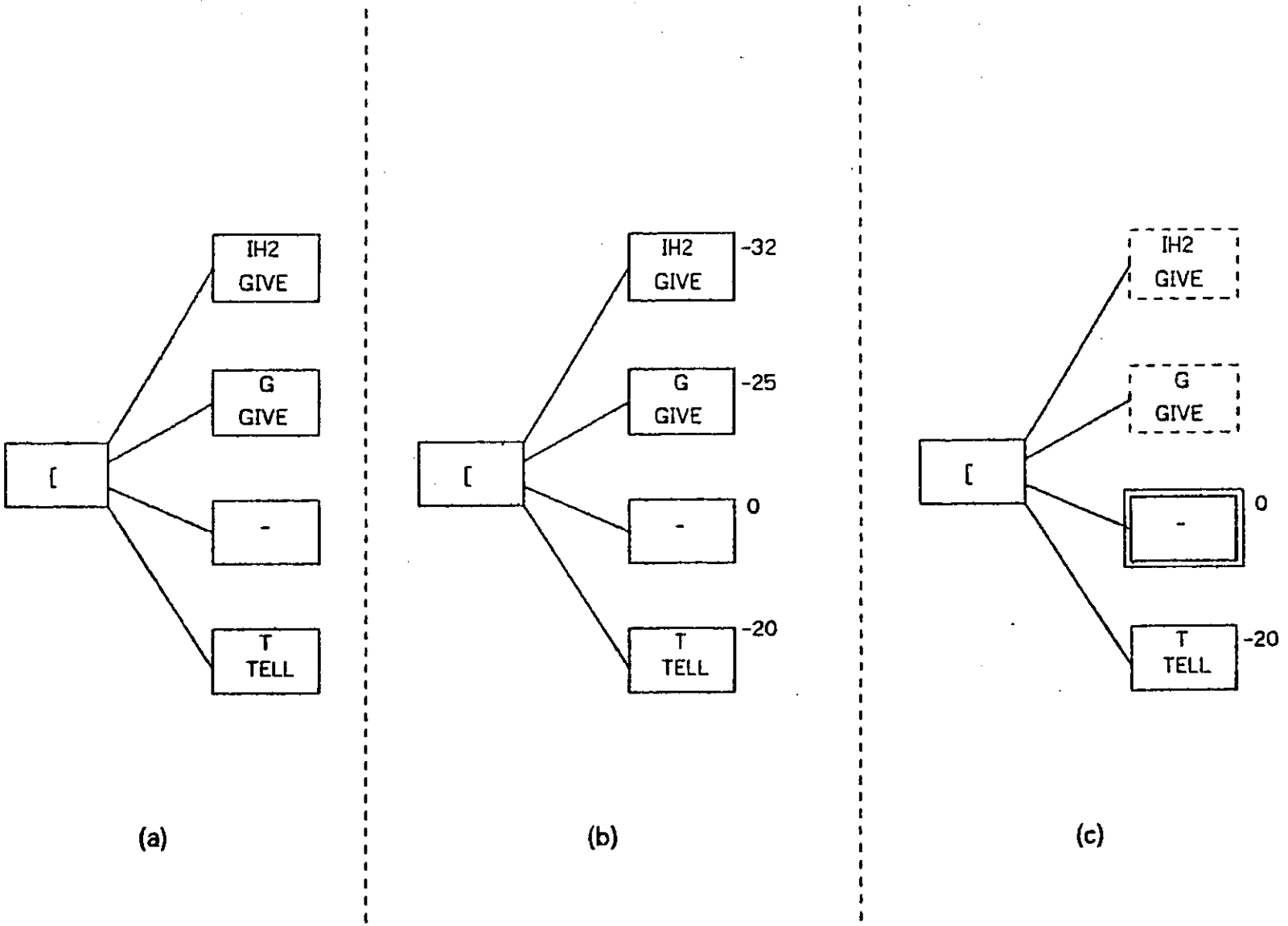


Figure 23a,b,c First Three Steps in the Recognition Sequence

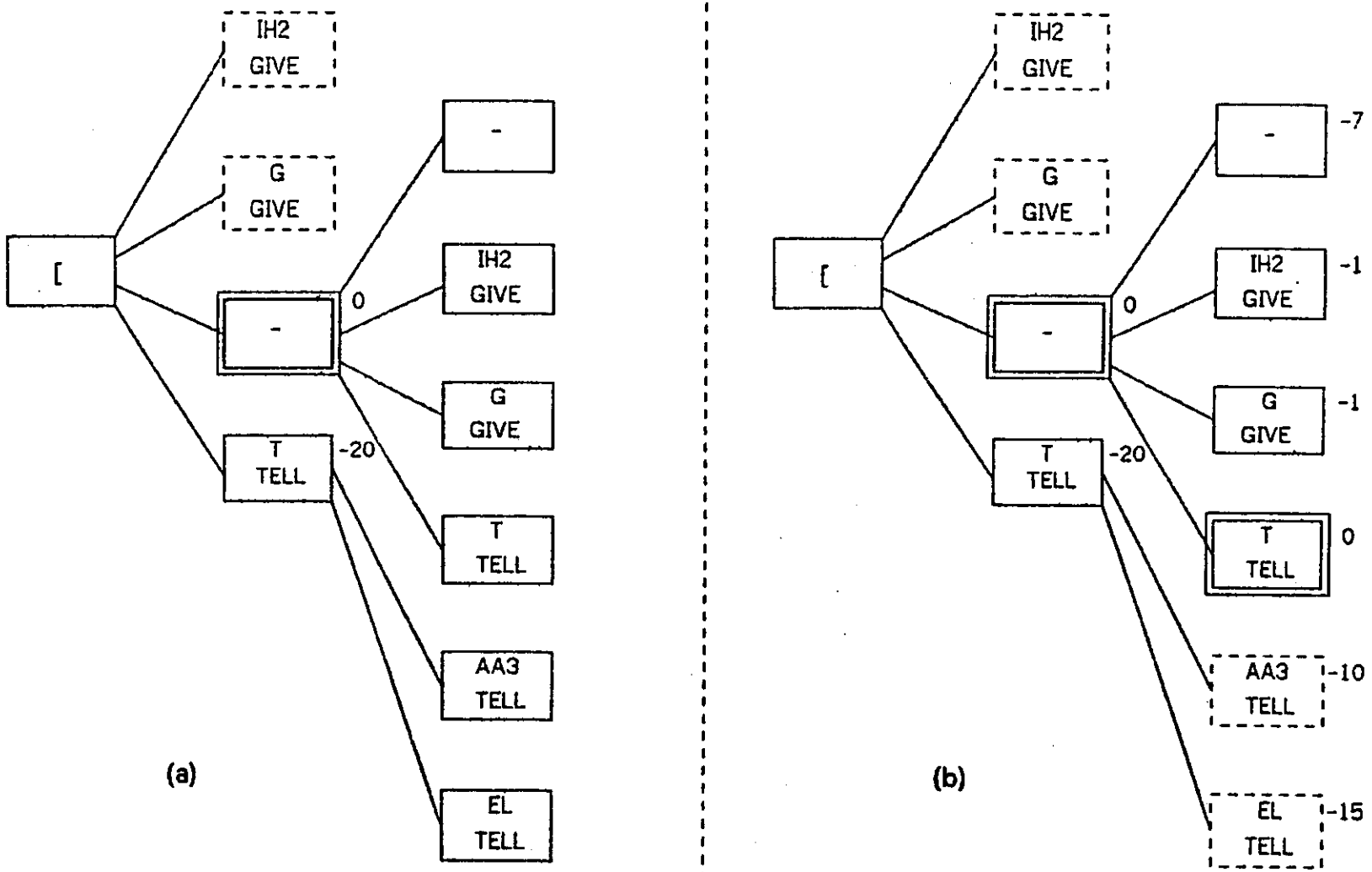


Figure 24a,b Recognition Sequence Steps Four and Five

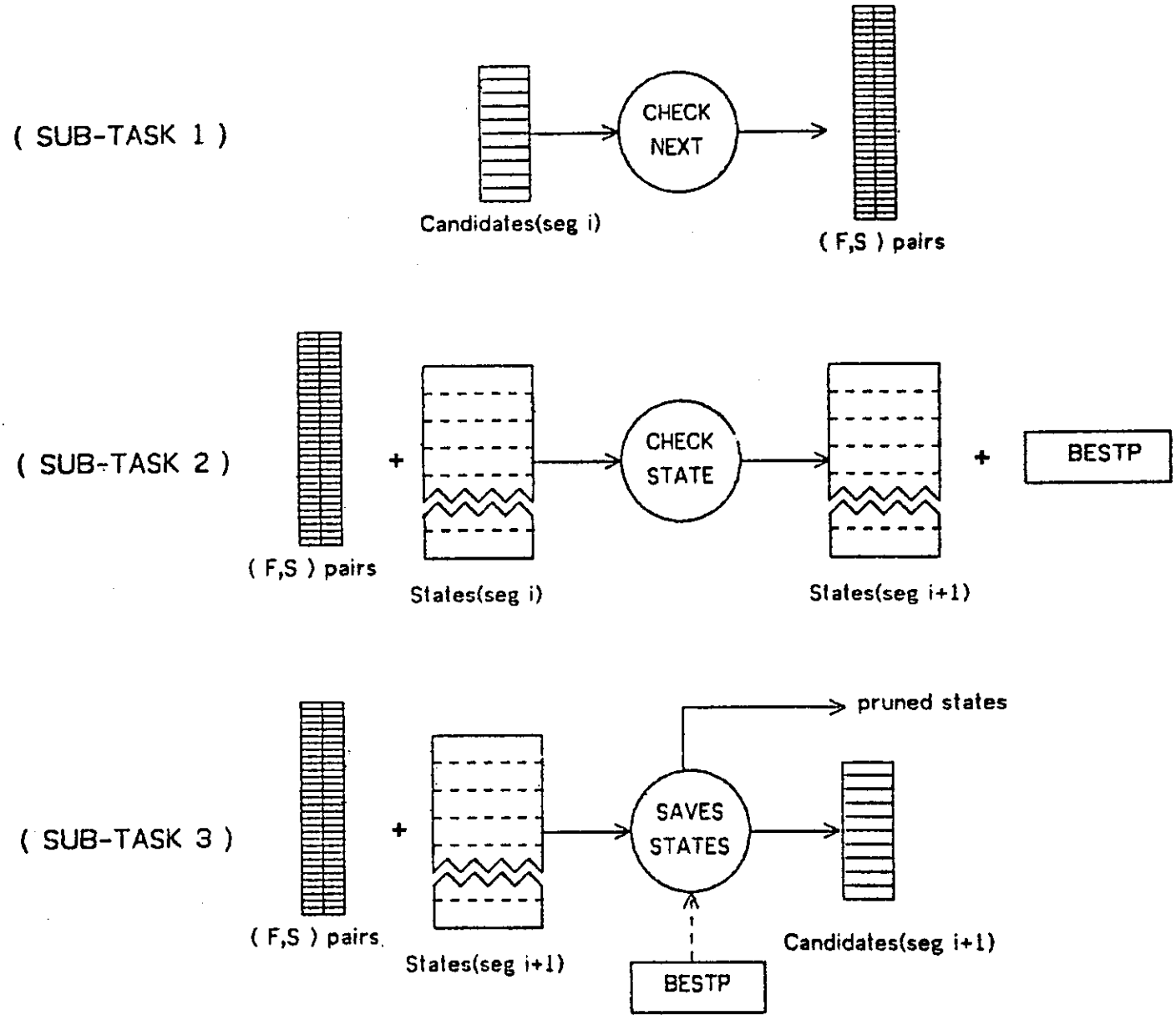


Figure 25 A Flow Diagram of Harpy's Search Algorithm

in the list is a father and son (F,S) pair.

CHECKSTATE — The probability of transitioning from father to son is calculated. If the value is greater than the current probability in the successor state, this value is updated; otherwise it is not. The best probability during this current segment also is saved, (BESTP), and is used later in the pruning phase.

SAVESTATES — A threshold is determined, based on the current best probability. All sons that have transition probabilities higher than the threshold become next generation candidates; all those below the threshold are discarded as unlikely paths.

In the uniprocessor implementation of the algorithm, the total compute time was divided among these routines in these proportions:

CHECKNEXT	21.5%
CHECKSTATE	46.0%
SAVESTATES	28.5%
OTHERS	4.0%
TOTAL	100%

Table 11 Compute Time Proportions

We therefore will attempt to speed up the algorithm by concentrating compute power in the form of task forces of cooperating processes to perform the three major routines--CHECKNEXT, CHECKSTATE, and SAVESTATES.

5.3 The Initial Implementation

5.3.1 Constraints on the Implementation of the HARPY system

In the implementations that follow, we have divided the heuristic search into two phases: the forward step, where the recognition tree is expanded; and the pruning step, where unlikely paths are removed from the tree. Hence, the forward step consists of the two routines, CHECKNEXT and CHECKSTATE, while the pruning step is performed by SAVESTATES.

Two decisions were made prior to the design of the first implementation: the first, because of the nature of the algorithm; and the second, to simplify the data structures.

- Because the pruning step cannot begin until BESTP is found, the forward step must be completed before the pruning step can begin.

- For simplicity, the forward step will not begin until the previous pruning step is completed. This decision was made because the forward step modifies the STATE vector, which is input to the pruning step. To maintain the data's integrity, it would be necessary to add an additional dimension to the STATE vector indicating the speech segment for which the state's path probability is calculated[1].

5.3.2 Control Structures and Data Sharing

The original version of HARPY combined the two sub-tasks, CHECKNEXT and CHECKSTATE, forming the forward step of the algorithm. The initial C.mmp implementation is a parallel version of the uniprocessor algorithm.

Since the pruning step cannot begin until the forward step is completed, we use a *synchronous control structure*[2] to sequence the pruning step after the forward step. Similarly, another synchronous control structure sequences the forward step to process speech segment(i+1) after the pruning step completes processing speech segment(i).

The cooperating processes in the forward step *statically*[3] allocate the candidate states among themselves. Each process is assigned an equal number of candidates to work on: the process first expands a candidate into a list of successor states and then calculates the probability of transitioning to each of these states from the candidate. When a process exhausts its supply of candidate states, it must wait for the other task force member processes to finish before the pruning step can begin.

---

[1] It would be possible to immediately expand candidates into (F,S) pairs as soon as they are created by the pruning step, but this implementation is not discussed here.

[2] If sub-task(j) takes as input the output of sub-task(i), and if sub-task(j) cannot begin until all processing at sub-task(i) is finished, then the control structure to sequence sub-task(j) after sub-task(i) is a synchronous control structure.

[3] Data is allocated statically in a task force if the processes do no compete for the data items. Instead, each process has a private partition of the data.

In the pruning step, work units are *dynamically*[4] allocated from a data stack. A process takes the top element on the stack, a successor state, and performs a calculation to determine if the state's path probability is above the pruning threshold. If it is, the state is saved and becomes a candidate for expansion in the next iteration of the forward step. If the path probability falls below the threshold, the state is discarded.

5.4 Performance of the First Implementation

The performance of this implementation is presented in two parts: the forward step and the pruning step. In both cases, three measurements were performed:

Elapsed time	to process fifteen utterances.[1]
Speedup	relative to the single process instantiation as the number of cooperating processes in the task forces is increased.
Pc utilization	as the number of cooperating processes in the task forces is increased[2].

5.4.0.1 The Performance of the Forward Step

In figure 26, the elapsed time to perform the forward step of the algorithm decreases from 52.89 seconds in the single process instantiation, to 18.14 seconds when eight processes are incorporated into the algorithm. This improvement corresponds to a relative speedup of only 2.914.

In figure 27 we compare the algorithm's relative speedup, as a function of the number of processes, to linear speedup. Theoretically, if n processes cooperate to perform the algorithm's forward step, the elapsed time to perform the task should be reduced by a factor of n. Unfortunately, the speedup exhibited by the algorithm is substantially less than linear.

Figure 28, which graphs process utilization as a function of the number of processes, sheds some light on the reason for less than linear speedup. In this graph, process utilization decreases rapidly as the number of processes increases. At eight processes, only 27.5% of the available processing power is being used. The under-utilization of processing power indicates that allocation of data to the processes is the source of the performance problem.

---

[4]Data is dynamically allocated in a task force if the processes compete, or share, all the data. There is no pre-assignment of data to specific processes.

[1]These utterances came from the Aritificial Intelligence information retrieval task [Lowerre 76], see Appendix.

[2]The processor utilization measurement does not include operating system related effects on utilization such as: context swaps, time-slice end rescheduling, and interrupts from I/O devices.

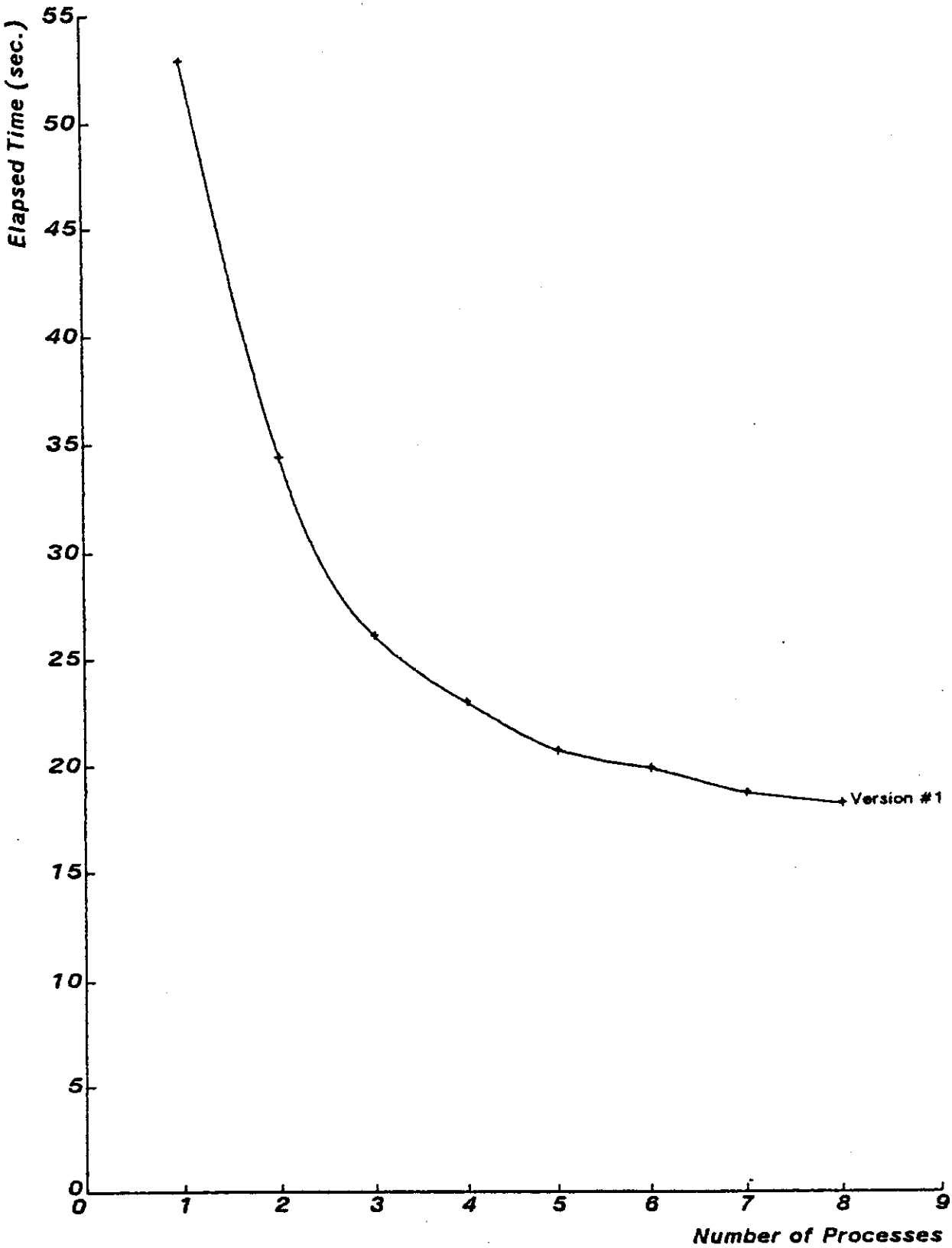


Figure 26 Decomposition of the Forward Step-- Version #1

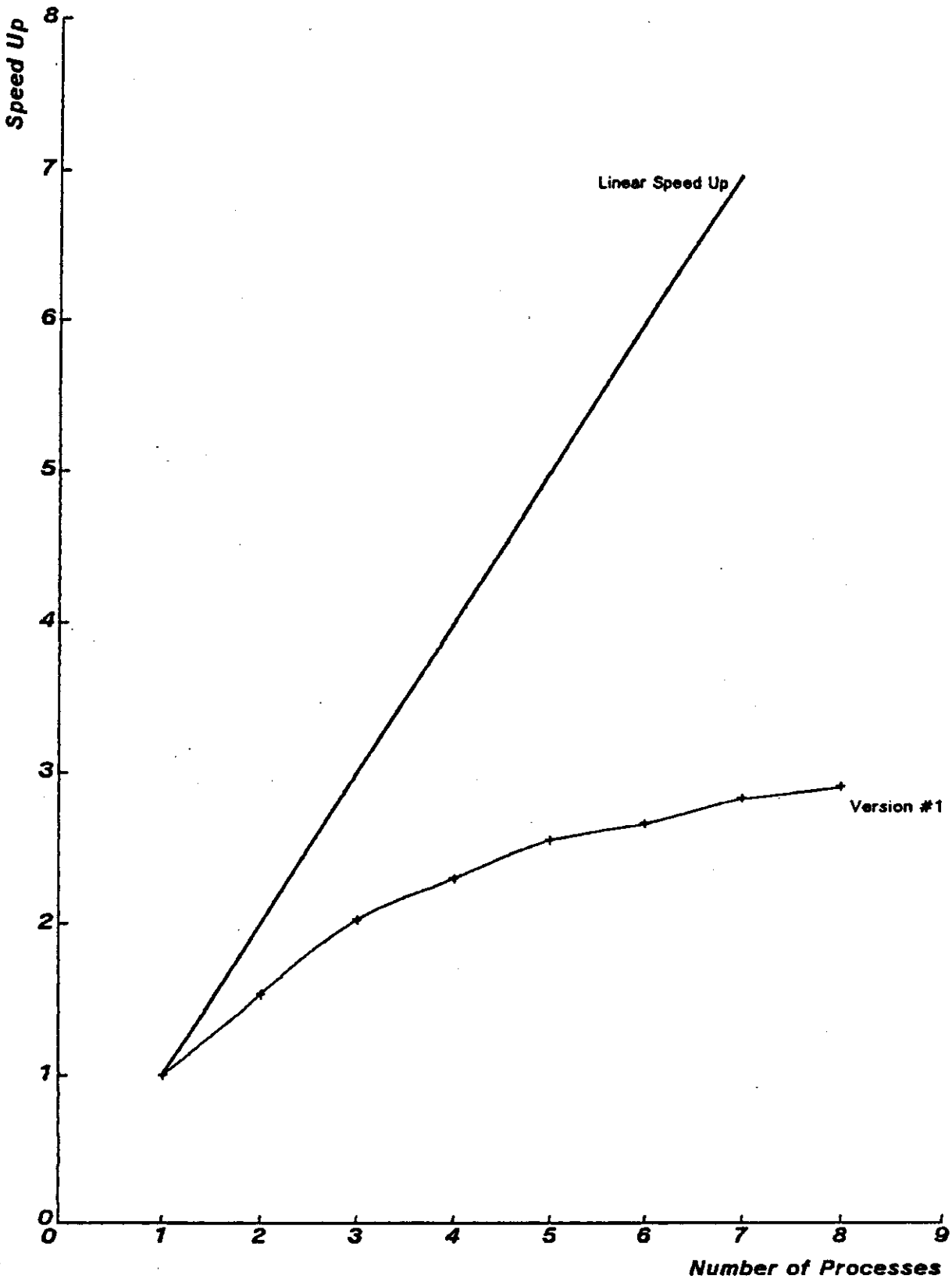


Figure 27 Decomposition of the Forward Step-- Version #1

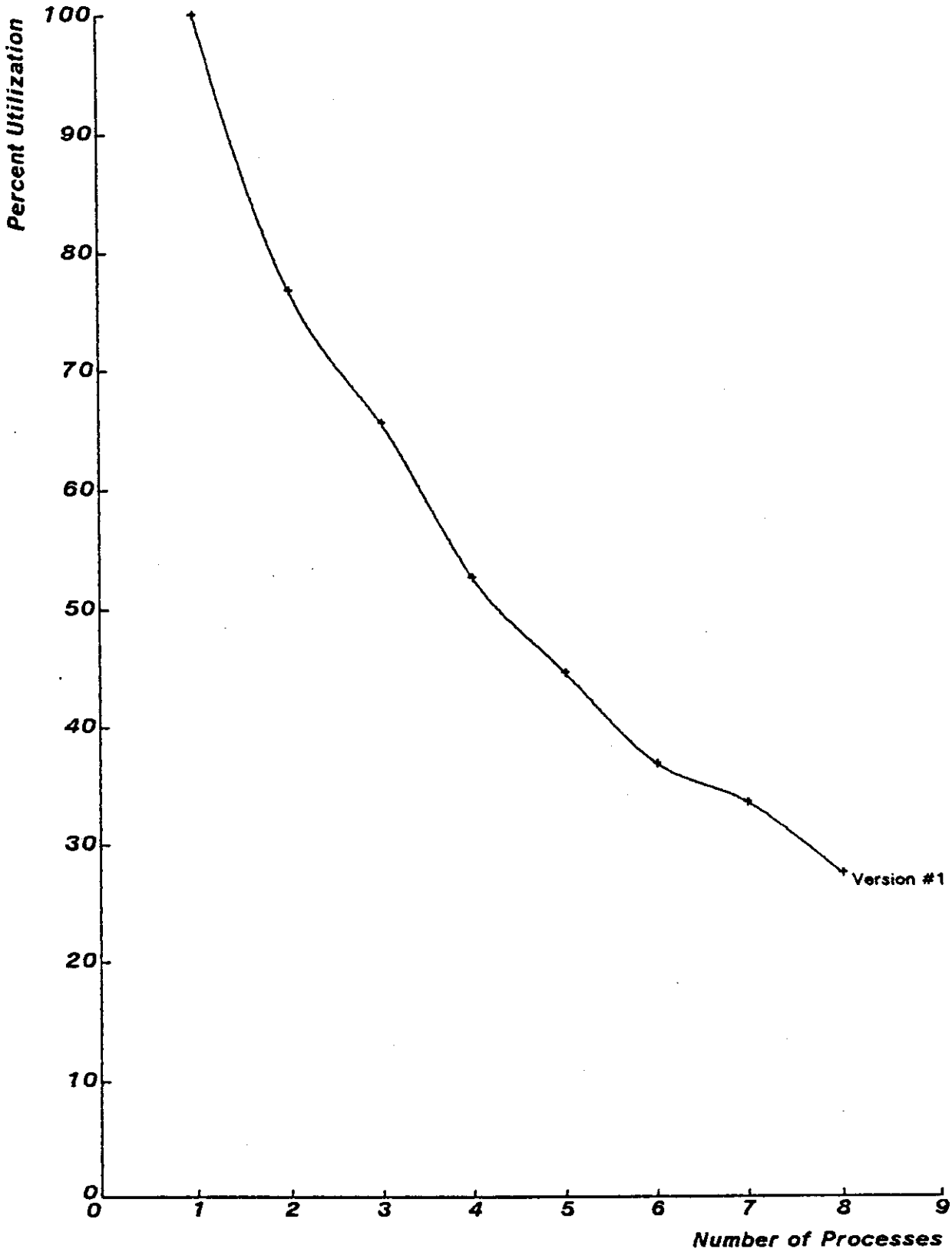


Figure 28 **Decomposition of the Forward Step-- Version #1**

In order to understand the reason for this poor behavior, we must look more closely at the work units and the way they are allocated to the processes.

For the synchronous control structure to perform well, the processes must arrive at the synchronization point together. Any one process that lags behind will cause the entire task force to wait.

Although each process receives an equal number of candidate states to work on, this method does not guarantee that each process will receive an equal share of the total computation. Figure 29, which is a graph of distribution of compute time for expanding a candidate state into the list of successor states, shows that while most candidate states can be expanded in less than five milliseconds, occasionally the expansion can take as much as thirty milliseconds to perform. In addition, the distribution of the time to perform the transition probability calculation is bimodal, figure 30. The first peak, at 800 microseconds, corresponds to performing the transition probability calculation and not updating the STATE vector. The second peak, which is centered at 1200 microseconds, corresponds to performing the calculation and also updating the STATE vector with the new value.

If the number of candidate states each process received were very large, the variation in the compute time would have small impact on the performance of the forward step. Unfortunately, this is not the case. Figure 31, which is the cumulative distribution of the number of candidate states per segment of speech, shows that the average number of candidate states per segment is small; 65% have fewer than ten candidate states, and 24% have but a single candidate.

Thus, although the current method allocates an equal number of candidate states to each process, those processes that receive many 'prolific' states will perform more computations than those processes that receive mostly 'barren' candidate states. The net result is under-utilization of the processes caused by an unequal allocation of work.

5.4.0.2 The Performance of the Pruning Step

The performance of the pruning step is much better than the forward step. In figure 32, the elapsed time to process the fifteen utterances is reduced from about 28 seconds to less than six seconds when eight processes are incorporated into the algorithm. In figure 33, where the speedup of the pruning step is plotted as a function of the number of processes, almost a fivefold improvement is realized when eight processes cooperate to perform the pruning step. Although less than linear speedup is exhibited, the performance of the pruning step is substantially better than the forward step. The successful implementation of the pruning step stems largely from two sources.

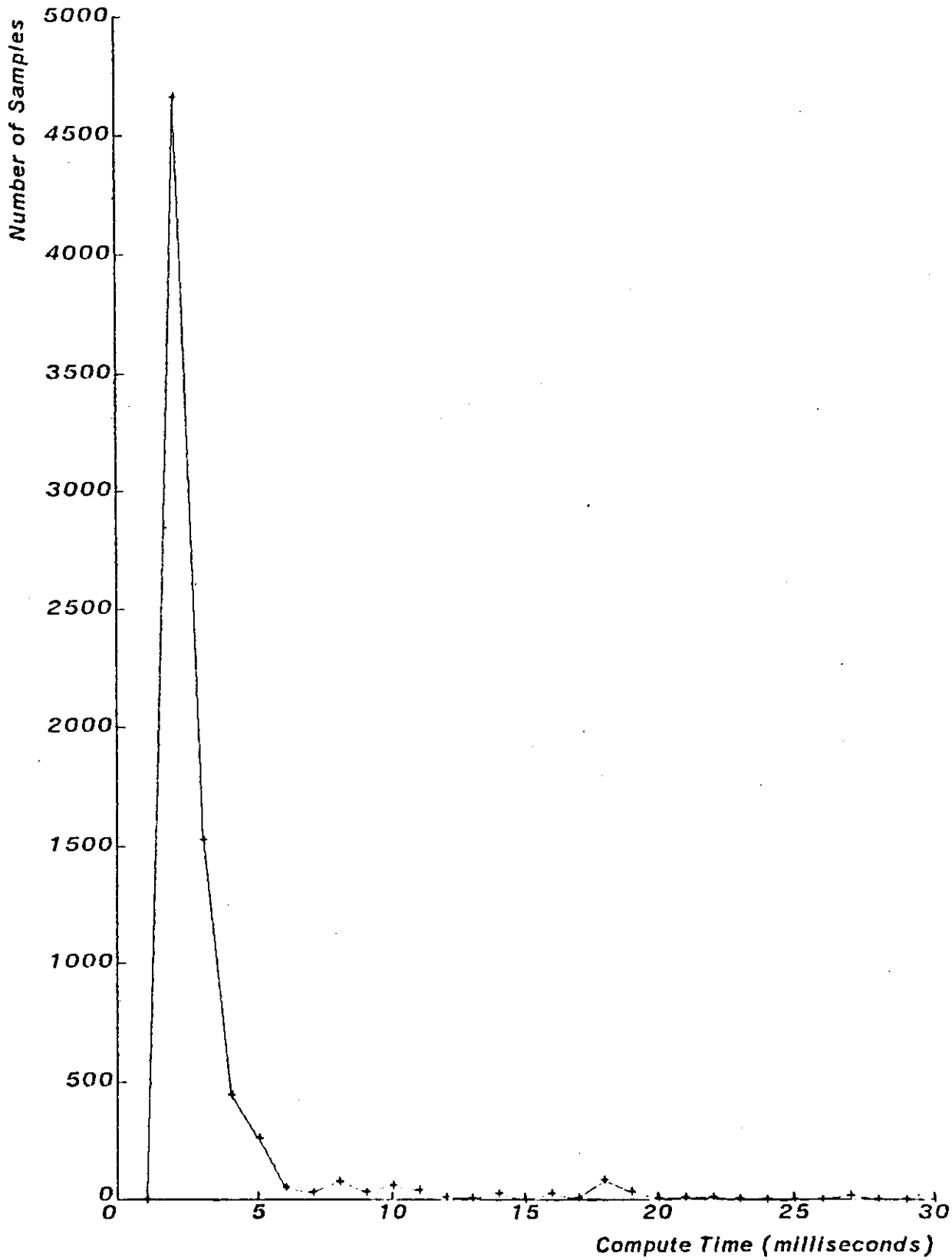


Figure 29 Distribution of the Compute Time to Expand Candidates

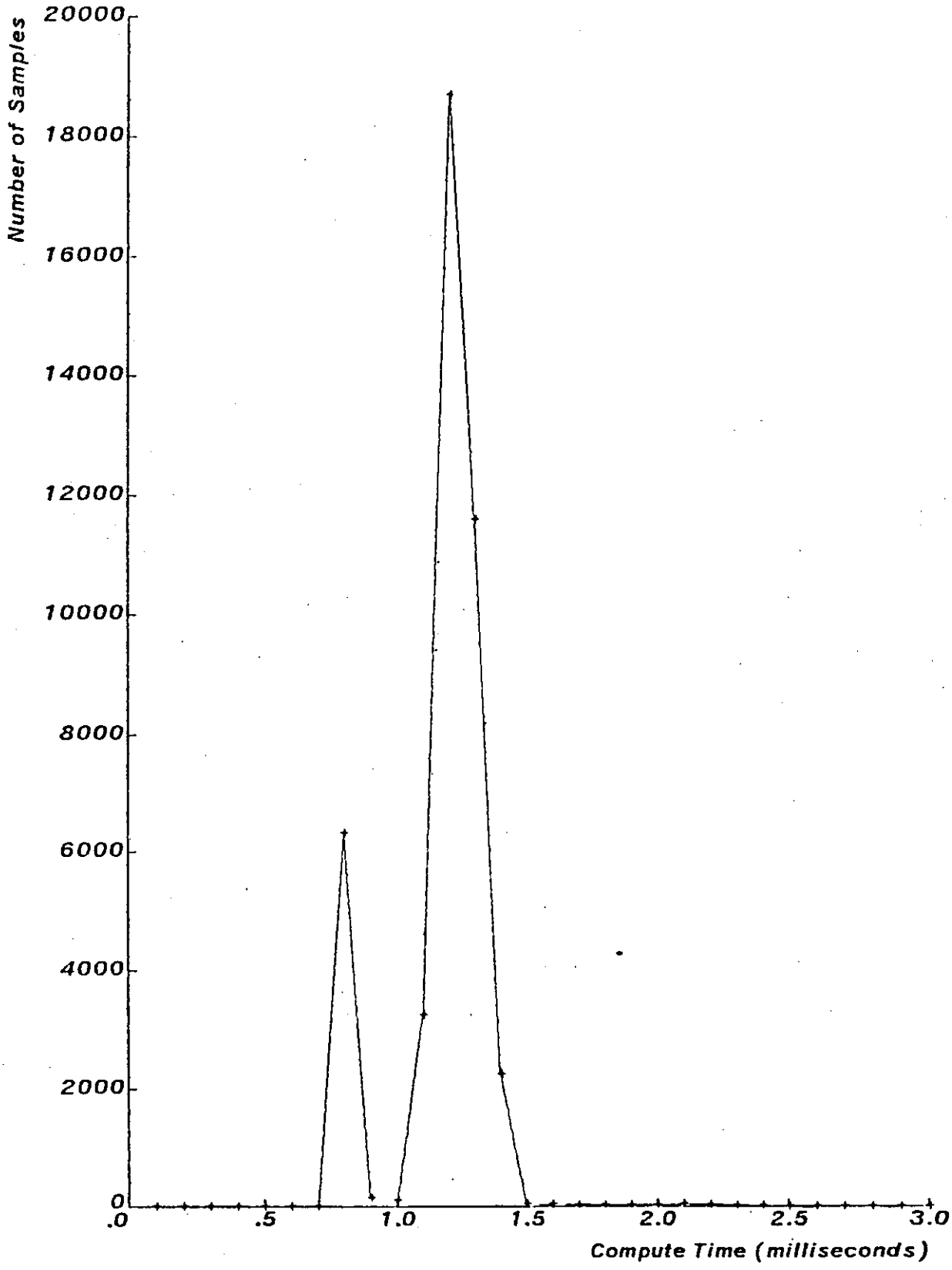


Figure 30 Distribution of Transition Probability Compute Time

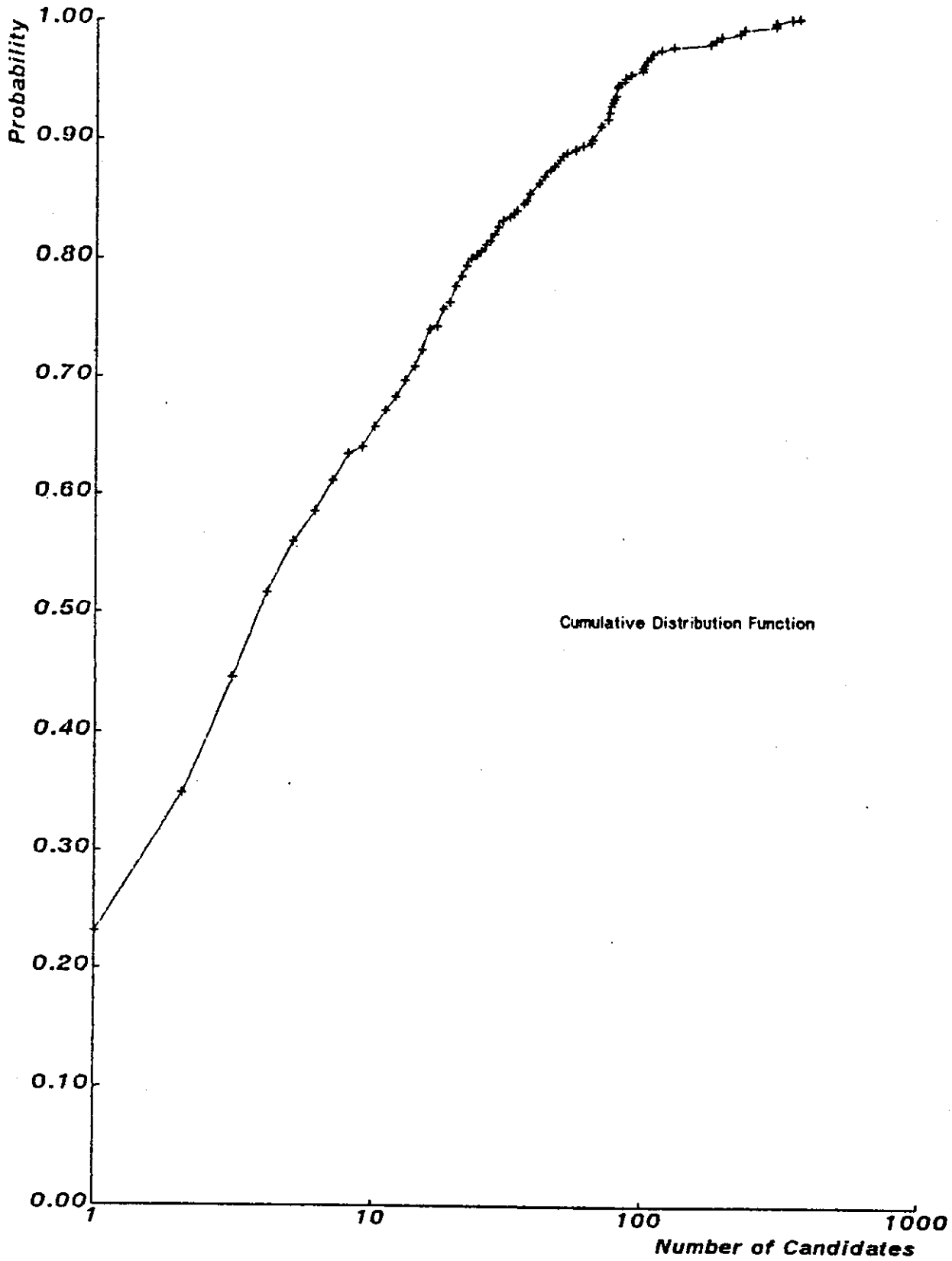


Figure 31 Distribution of Candidates per Segment of Speech

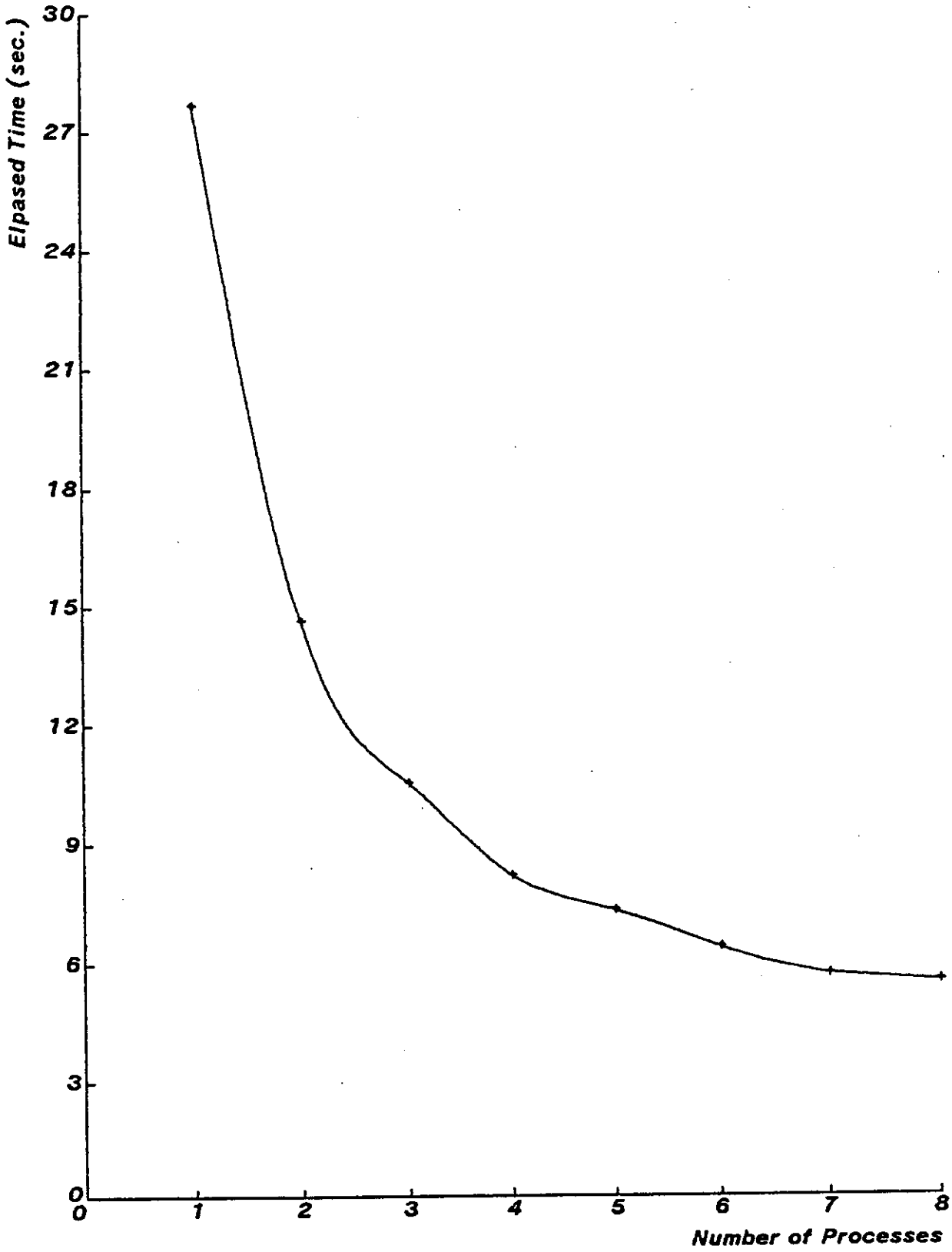


Figure 32 Elpased Time to Perform the Pruning Step

First, processes dynamically acquire work units, father and son (F,S) pairs, from a data stack.[1] Thus, even though the distribution of the compute time to perform the threshold calculation on an (F,S) pair is bimodal, figure 34, the processes arrive at the synchronization point almost simultaneously. Since the last process to finish can hinder the rest of the task force by at most one candidate state's processing time, high process utilization results, figure 35.

Second, the pruning step's data stream contains more work units than the forward step's data stream. The cumulative distribution of the number of (F,S) pairs per segment of speech can be found in figure 36. The table below summarizes the difference between the two data streams.

Work Units per Segment	CANDIDATES	(F,S) PAIRS
1-10	65.7%	38.0%
10-100	30.3%	32.0%
100-1000	4.0%	28.0%
>1000	0%	2%

Unlike the distribution of candidate states, the number of (F,S) pairs per segment of speech spans more than three orders of magnitude. More than 30% of the segments have greater than one hundred (F,S) pairs; less than 40% have only ten or fewer pairs.

In summary, the dynamic allocation of data in the pruning step is the key to its successful implementation. When a synchronous control sturcture is used to synchronize cooperating processes it is imperative, in order to maintain high process utilization, that the processes arrive at the synchronization point together. Dynamic allocation of data ensures that high process utilization will occur, as processes do not develop a backlog of unstarted work units, while other processes are idle due to a lack of work.

5.5 Refinements to the Initial Implementation

In this section, three refinements to the initial implementation are presented. In each, only the implementation of the forward step was enhanced. The performance of the initial implementation will serve as the baseline for measuring the performance improvement each refinement contributes. As it is possible to measure the performance of the forward step separately from the algorithm as a whole, measuring the performance improvement of the

[1] On the average, less than 30 microseconds of overhead is associated with obtaining a work unit from the stack.

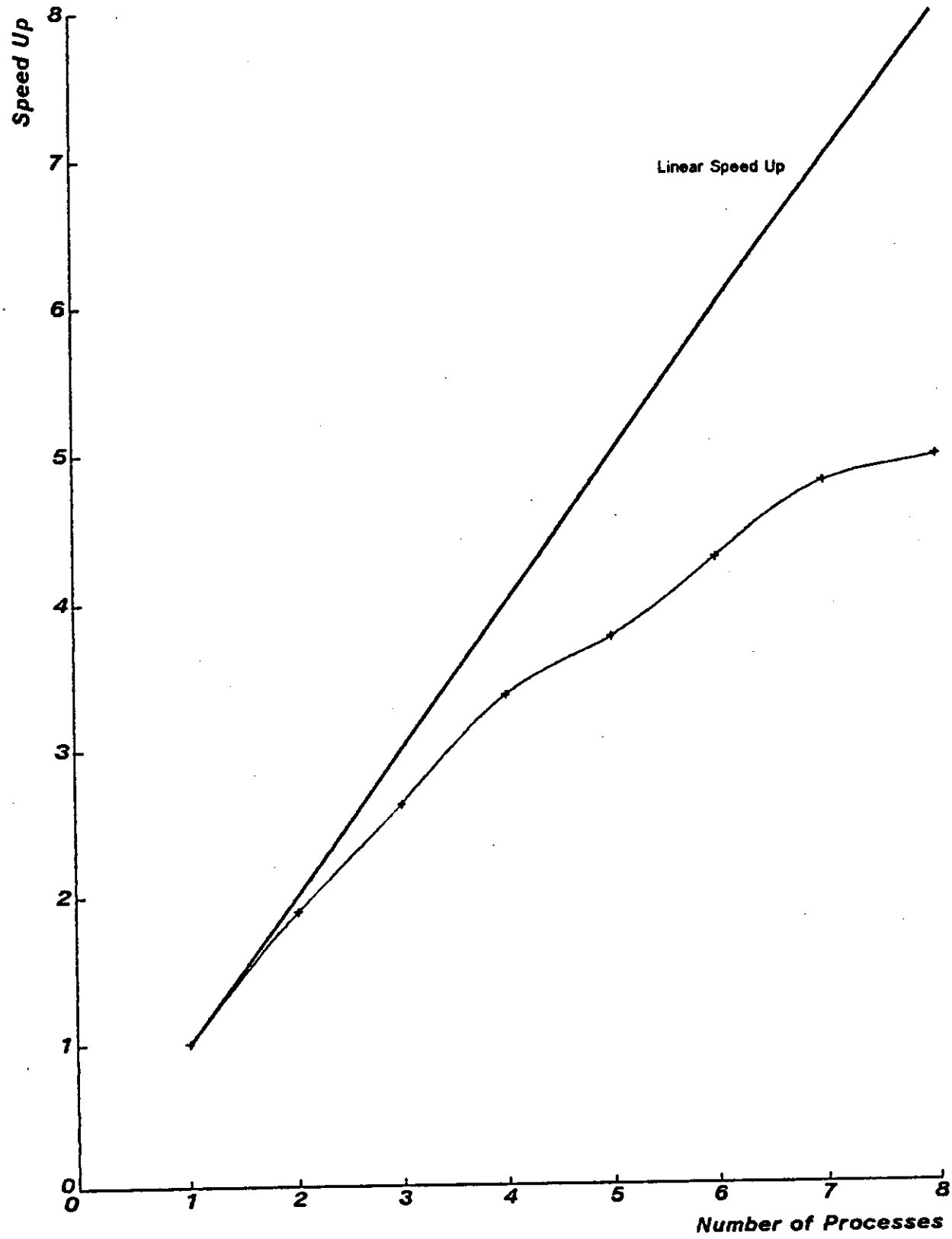


Figure 33 Speed Up During the Pruning Step

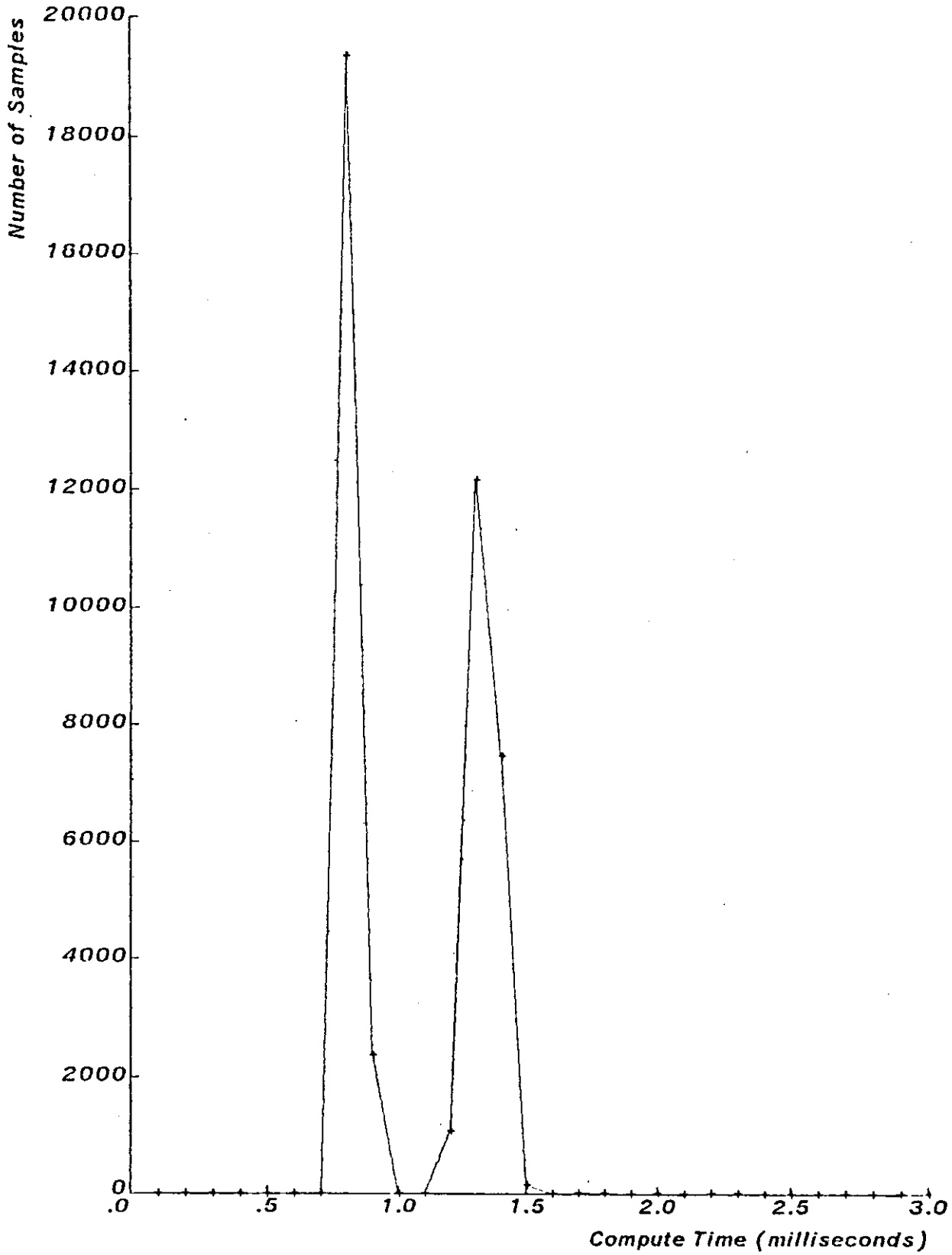


Figure 34 Distribution of Compute Time During Pruning Step

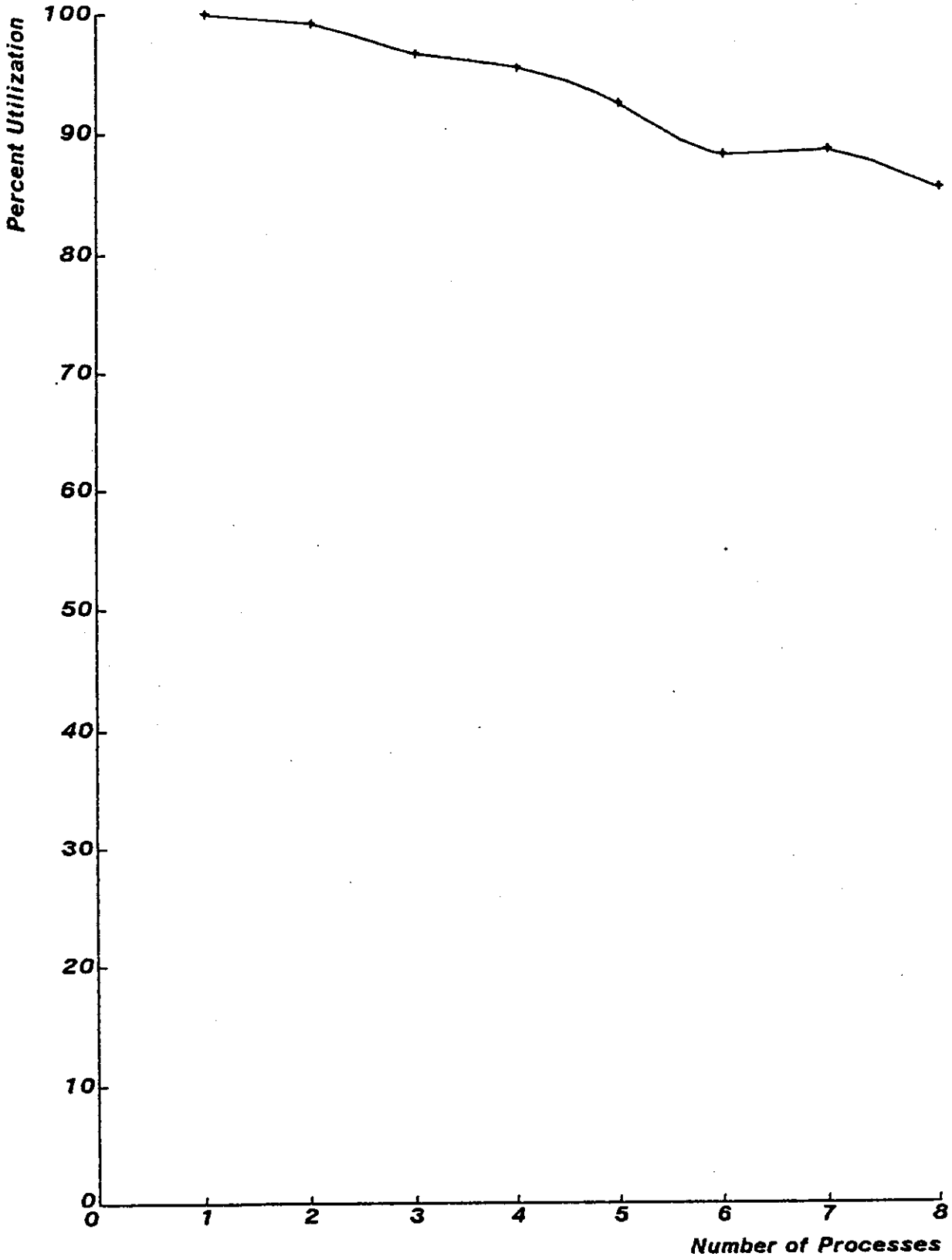


Figure 35 Processor Utilization During the Pruning Step

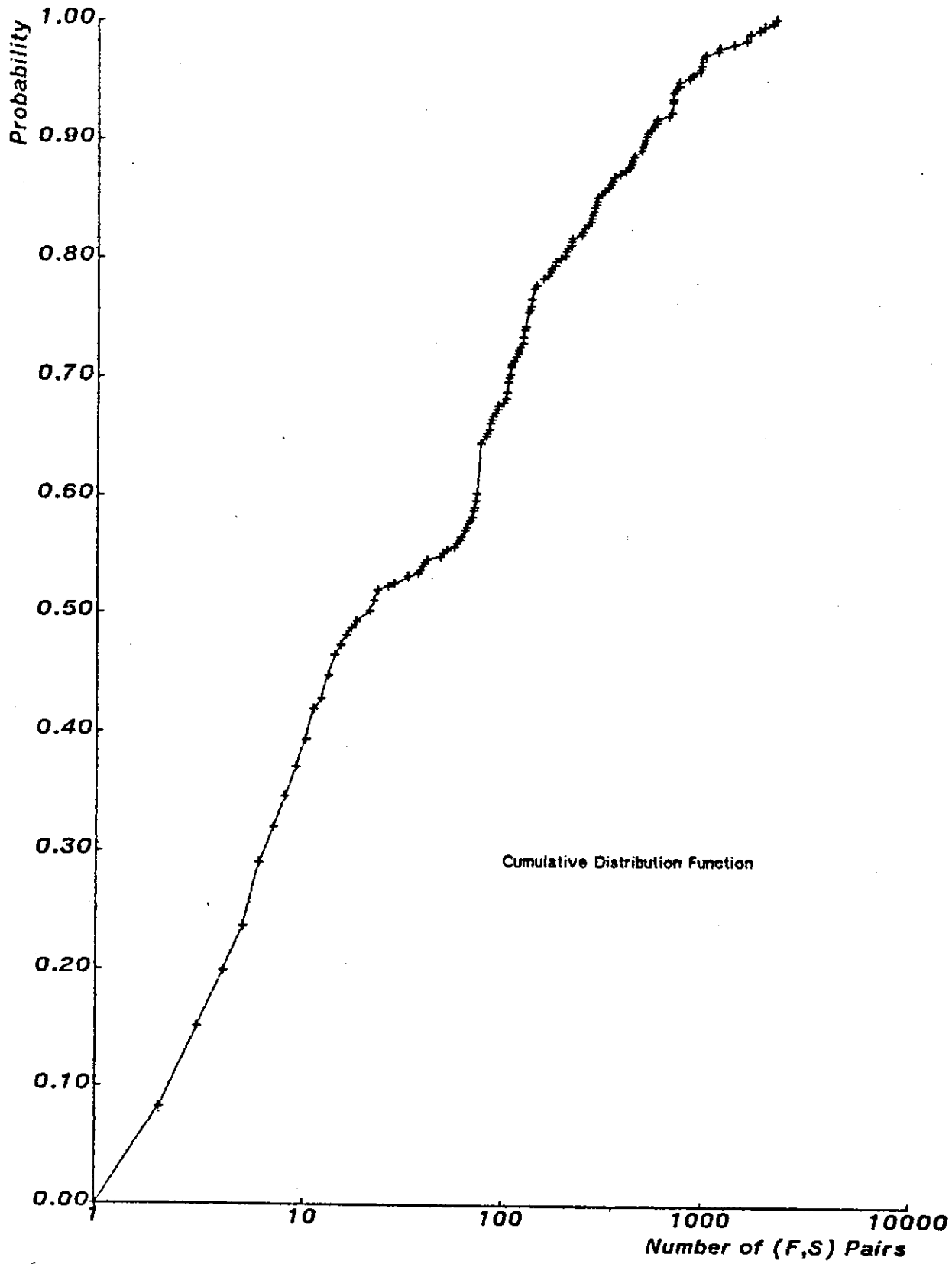


Figure 36 Distribution of (F,S) Pairs per Segment of Speech

forward step is sufficient to evaluate the refinement.

5.5.1 The First Refinement

In the initial implementation, the key to the success of the pruning step and the failure of the forward step was the allocation of work to the processes. During the forward step, the work units were allocated statically, by number instead of by amount of computation. This method resulted in an unequal distribution of work among the processes in the forward step task force.

We will attempt to allocate work units equally by dynamically allocating candidate states to the forward step task force. Now, as in the pruning step, a process takes a work unit from the stack of unstarted work only when it is ready to start processing the new work unit. All other aspects of the algorithm will remain the same.

We compare the elapsed time to perform the forward step under the two work allocation strategies in figure 37. From the two process case on, the dynamic allocation method outperforms the static method. At eight processes, the maximum measured parallelism, a 16% performance improvement results; the elapsed time to perform the task reduced from 18.15 seconds to 15.20 seconds.

Similarly, greater speedup is achieved by dynamically allocating work to the task force. Speed up as a function of the number of processes is graphed in figure 38 for both implementations. In all measurements, dynamic allocation of work yields higher performance than static allocation. For eight processes, a speedup of 3.47 was achieved using the dynamic strategy, compared to only 2.91 for the static method.

An improvement in process utilization also resulted. In figure 39 process utilization under the two allocation strategies is graphed as a function of the number of processes. For the eight process instantiation, a 32.7% utilization was achieved using dynamic work allocation, compared to 27.5% process utilization when the work was statically allocated to the processes. The table below summarizes the comparison of the two implementations for the eight process, maximum measured parallelism, case.

Performance Measure	Version #2	Version #1
Elapsed Time (secs.)	15.206	18.148
Processor Utilization	32.7%	27.46%
Speedup	3.471	2.914

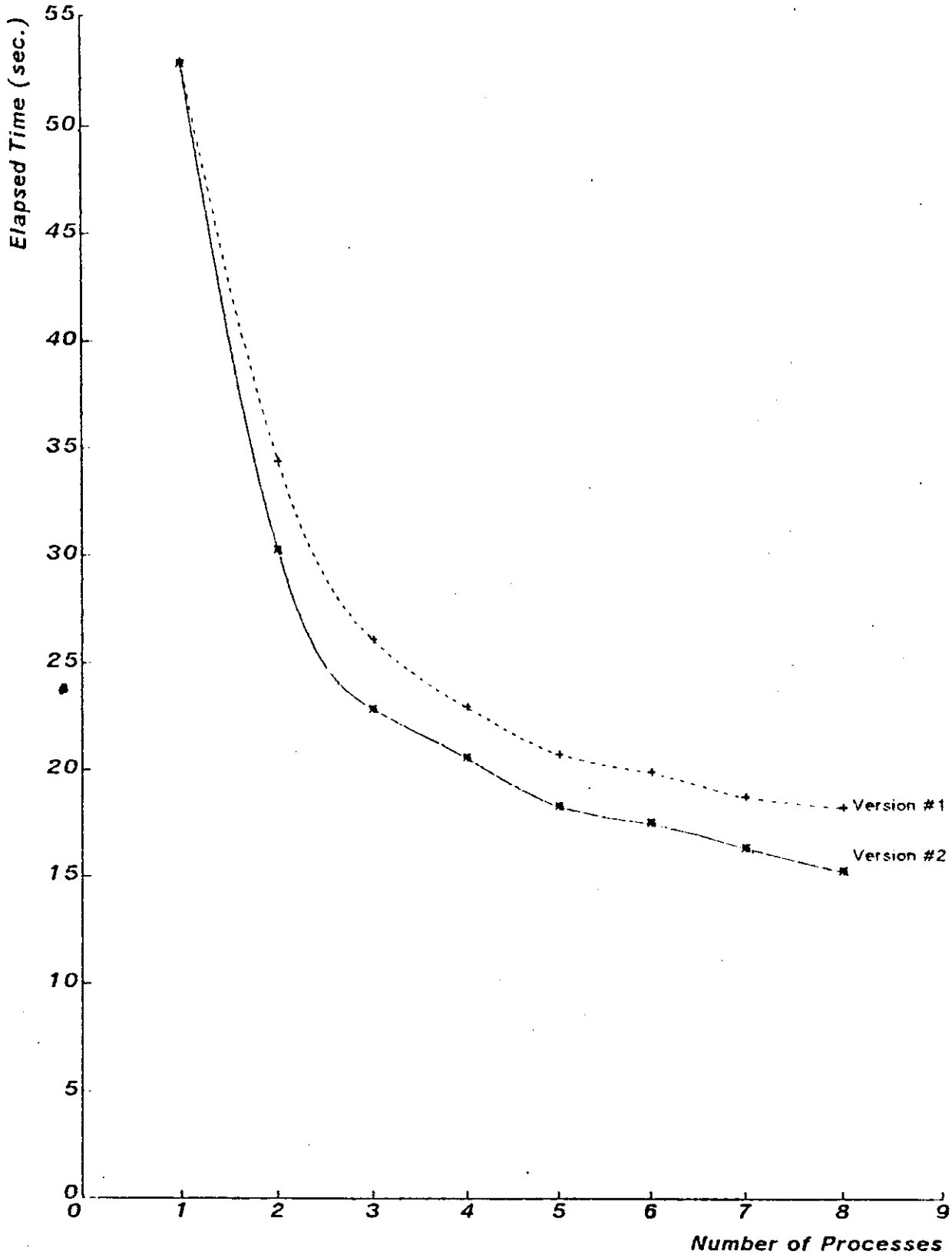


Figure 37 Decomposition of the Forward Step-- Version #2

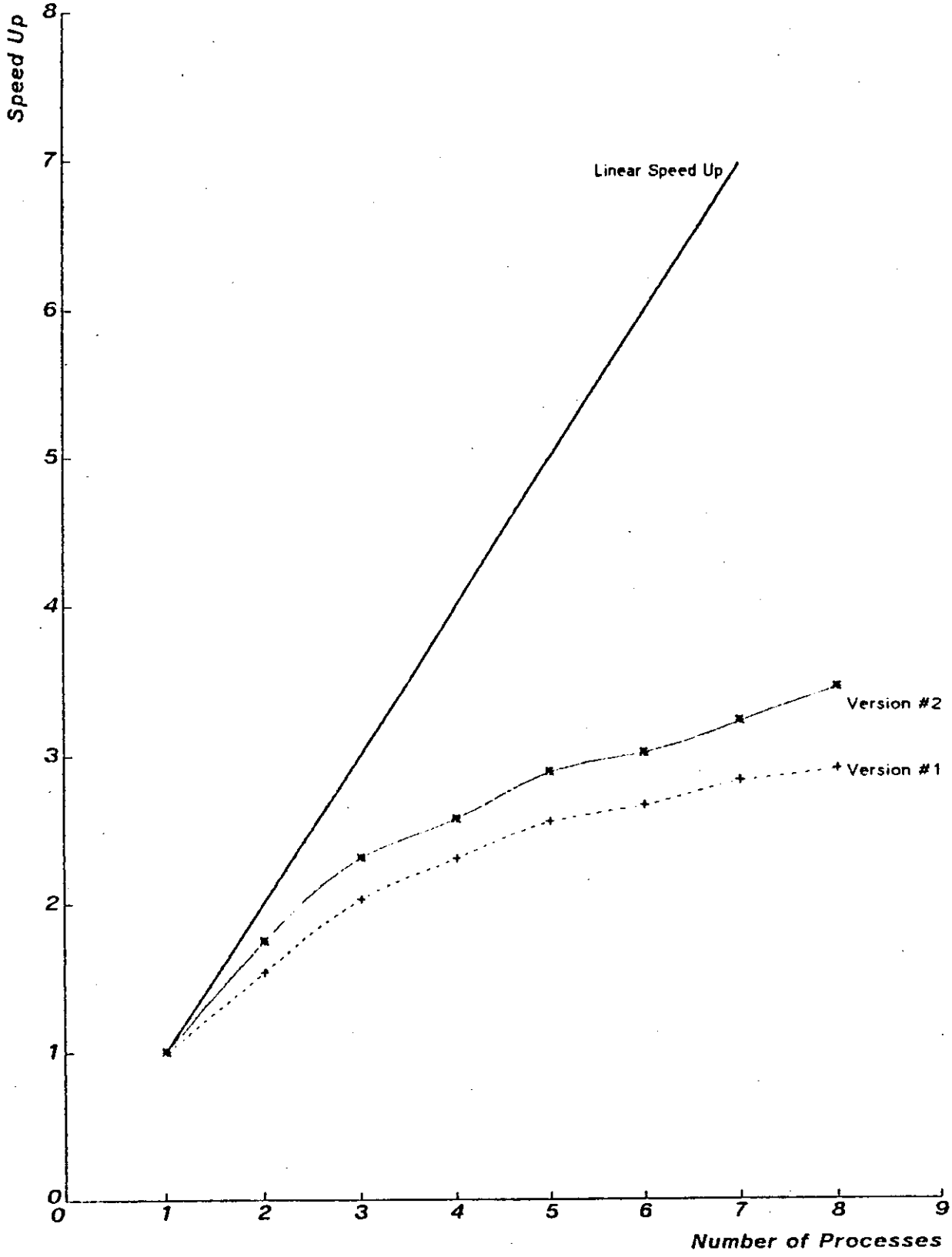


Figure 38 Decomposition of the Forward Step-- Version #2

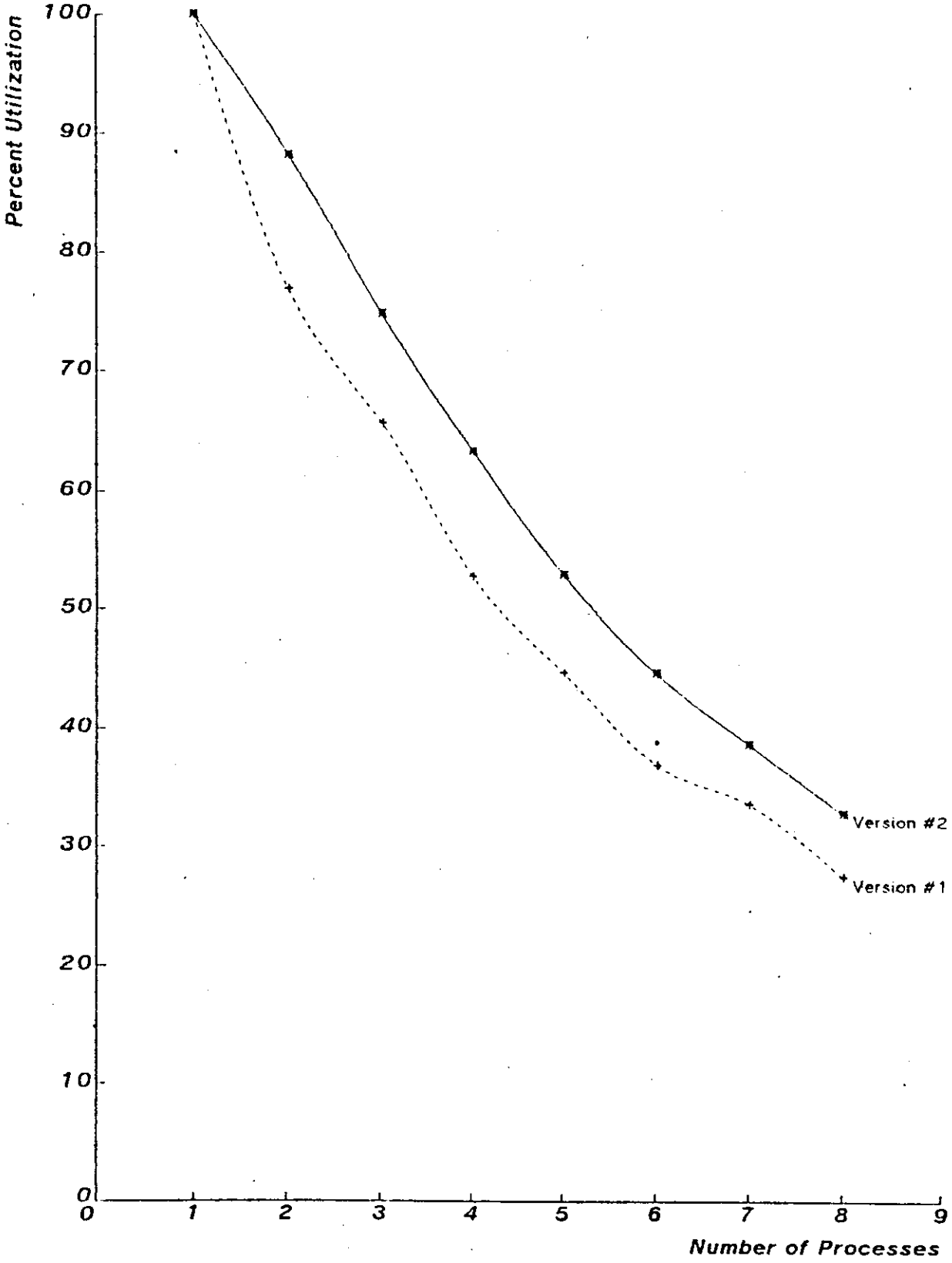


Figure 39 Decomposition of the Forward Step-- Version #2

In summary, switching from static work allocation to dynamic work allocation resulted in a 16% performance improvement. The performance improvement is small because there are not enough work units in the data stream. The small number of work units causes two problems: 1) a large variance in the amount of computation a process performs; and 2) often, processes do not receive even a single work unit. Together, these performance problems cause low process utilization during the forward step.

5.5.2 The Second Refinement

In this version of the forward step we solve the problem of low process utilization by decomposing the forward step into the two sub-tasks CHECKNEXT and CHECKSTATE. By making CHECKSTATE a separate sub-task, any process may now perform the transition probability calculation on an (F,S) pair-- not just the process that created the pair. In this way we have increased the number of work units in the data stream by breaking large units into many smaller ones.

As in the previous implementation, the CHECKNEXT task force begins processing the speech segment by taking candidate states from a stack and expanding them into a list of (F,S) pairs. However, instead of performing the transition probability calculation as each (F,S) pair is produced, the processes place them in another data stack that supplies the CHECKSTATE task force with input.

When all the candidate states have been expanded, the processes synchronize and the CHECKSTATE task force begins to execute. Thus, we initially will use the synchronous control structure to sequence the CHECKSTATE sub-task after the CHECKNEXT sub-task.

The elapsed time to perform the forward step is compared to the two previous versions in figure 40. In the single process instantiation, the latest version of the forward step is more than 20% slower than the two previous versions. This penalty results from the CHECKNEXT sub-task storing, and the CHECKSTATE sub-task retrieving the (F,S) pairs from a data stack. In the previous implementations, the storing and retrieving was unnecessary since the (F,S) pairs were not placed in a common pool; the process that created the pair also performed the transition probability calculation on it.

As the parallelism increases the elapsed time to perform the forward step is reduced from 66.69 seconds to 14.96 seconds. Thus, this version outperforms the initial implementation from the three process case on, and the first refinement from the six process case on, despite incurring the large initial overhead associated with storing and retrieving the (F,S) pairs.

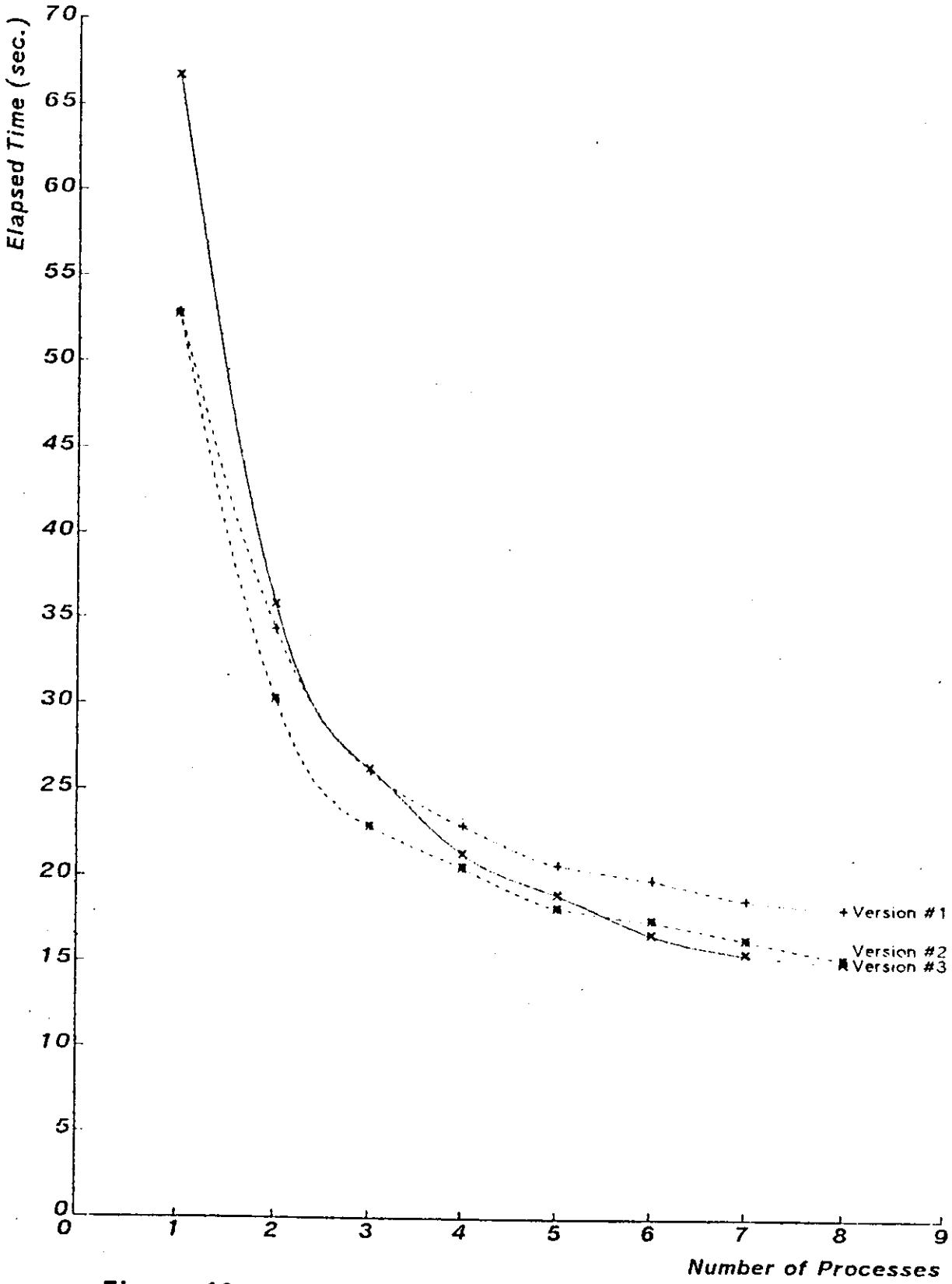


Figure 40 *Decomposition of the Forward Step-- Version #3*

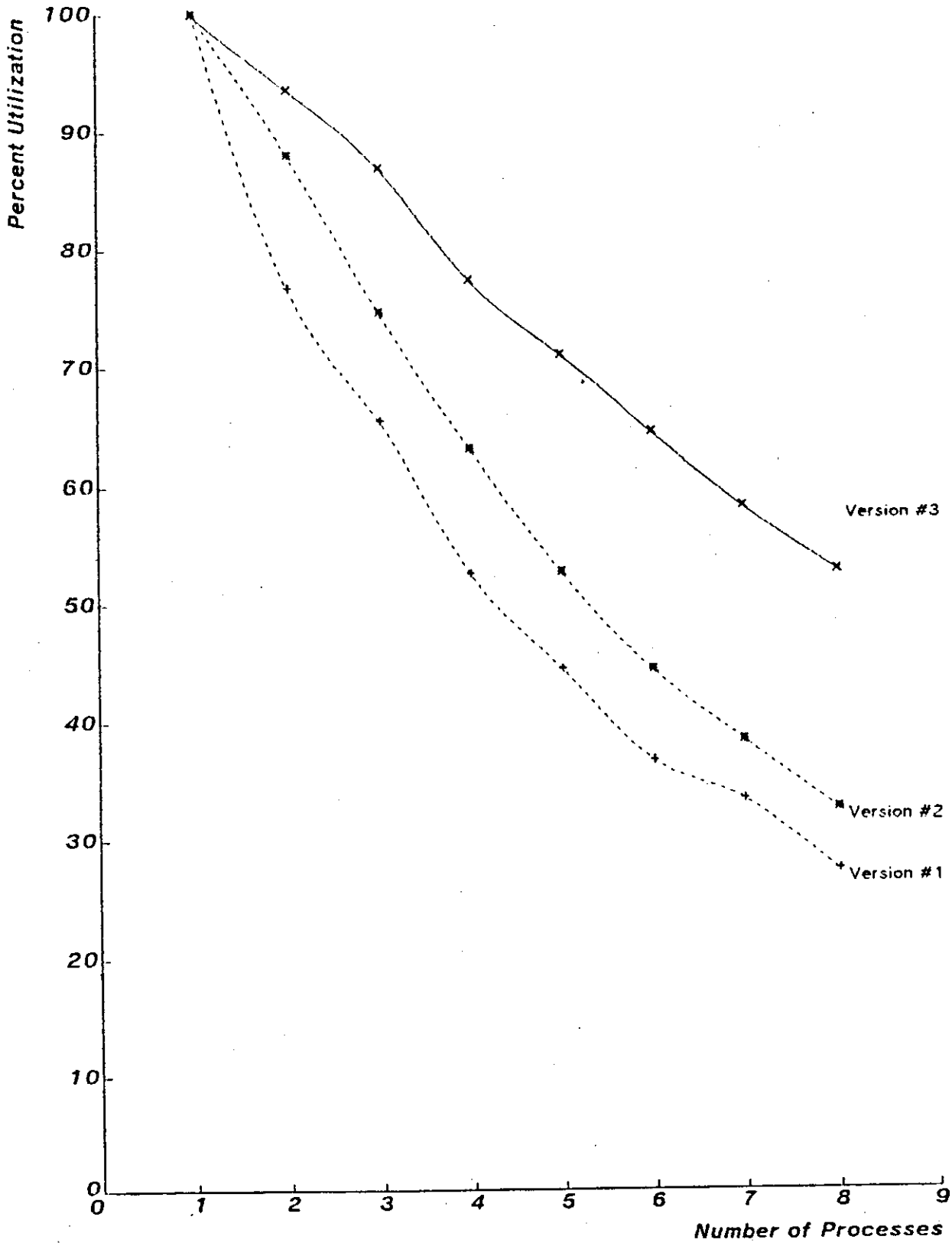


Figure 41 Decomposition of the Forward Step-- Version #3

Although the elapsed time to perform the forward step has not been significantly reduced, the process utilization has dramatically increased, as shown in figure 41. At maximum parallelism, eight processes, this version exhibits a 53% process utilization, compared to 32.7% utilization for the first refinement, and only 27.5% for the initial implementation. The substantial improvement results from sharing the (F,S) pairs more equally among the cooperating processes.

We compare the speedup of the three implementations, relative to the single process instantiation of the initial implementation, in figure 42. The latest version of the forward step is initially slower than the two previous versions-- a speedup of 0.79. However, as the parallelism increases, the latest version outperforms the two previous implementations; at eight processes a speedup of 3.535 compared to 3.479 for the first refinement, and 2.915 for the initial version.

To summarize, the latest enhancement to the algorithm, splitting the forward step into the two sub-tasks CHECKNEXT and CHECKNEXT, resulted in an initial overhead in the form of a 20% increase in elapsed time, caused by the additional manipulation of the (F,S) pairs. This is the cost we pay to share the (F,S) pairs equally among the task force. However, as the parallelism increased, a small improvement over the two previous versions was realized due to a substantial increase in process utilization. The table below compares the performance of the three versions when eight processes are in the task force.

Performance Measure	Version #3	Version #2	Version #1
Elapsed Time	14.966	15.206	18.148
Processor Utilization	52.99%	32.70%	27.46%
Speedup	4.456	3.471	2.94

5.5.3 The Third Refinement

In the previous implementation, the two sub-tasks CHECKNEXT and CHECKSTATE were still performed sequentially despite their being identified as separate sub-tasks. Any performance improvement obtained was achieved by sharing the computational load more equally among the cooperating processes. In this final refinement to the original implementation, we will perform the two sub-tasks of the forward step in parallel, obtaining still greater peformance improvement.

In this implementation, we sequence CHECKSTATE after CHECKNEXT with an *asynchronous*

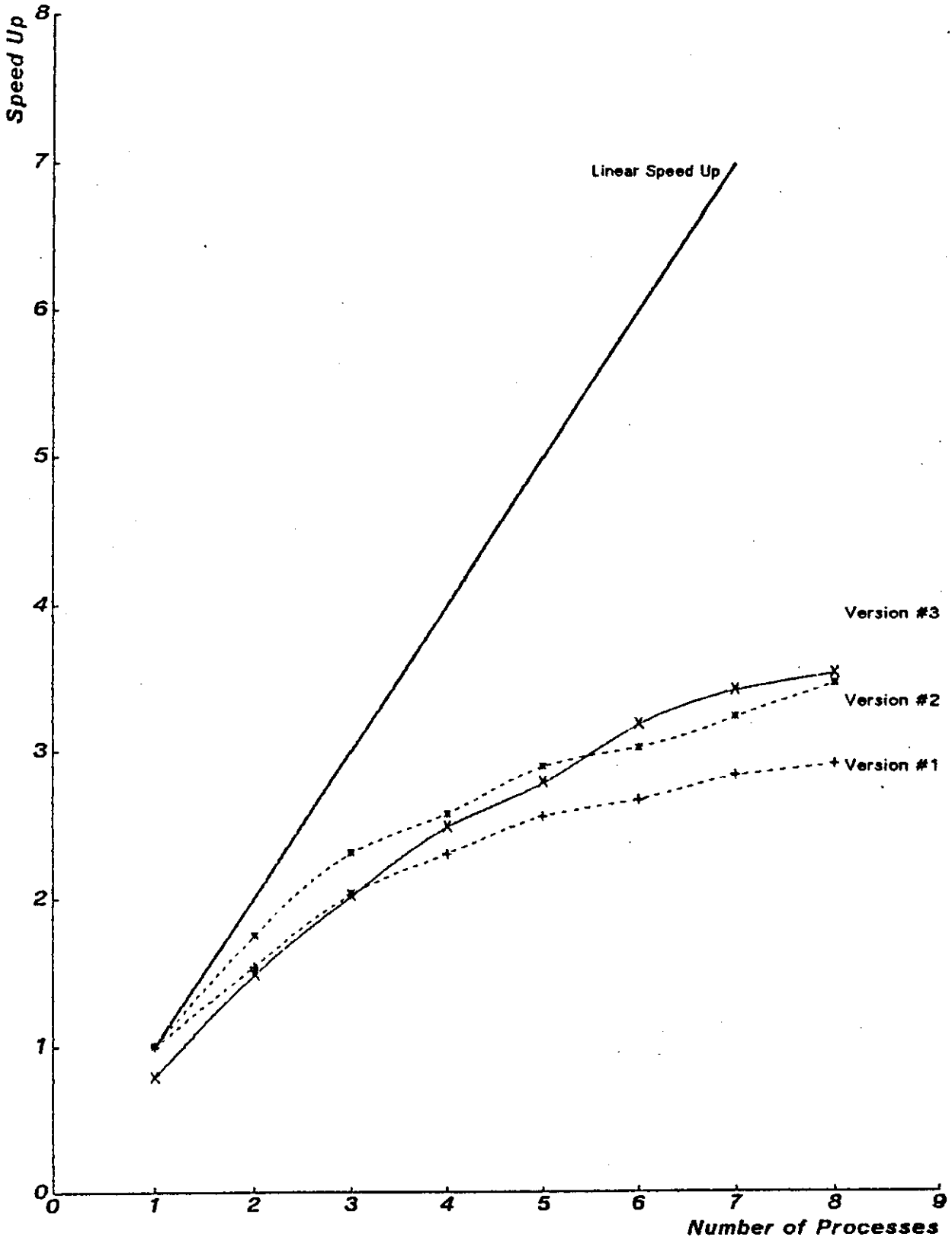


Figure 42 Decomposition of the Forward Step-- Version #3

control structure.[1] This allows the CHECKSTATE task force to begin calculating transition probabilities for the (F,S) pairs before the CHECKNEXT sub-task is completed. Thus, processes that cannot find any candidate states for expansion no longer become idle, waiting for their companions to finish. Instead these processes immediately begin to perform the transition probability calculation on the (F,S) pairs already produced. We can allow this type of parallelism because the CHECKNEXT task force only adds new (F,S) pairs to the CHECKSTATE task force's input stack; they do not modify any pairs already on the stack.

The elapsed time to perform the forward step for all four alternative implementations is compared in figure 43. The latest version of the algorithm outperforms the three previous versions from the three process instantiation on. The elapsed time is reduced from 65.3 seconds to 12.3 seconds at maximum parallelism-- more than two and one-half seconds faster than the next best version.

Performing the two sub-tasks in parallel has substantially increased the forward step's performance by maintaining higher process utilization. In figure 44, the process utilization of this version is compared to the three previous ones. At maximum parallelism, the final version of the forward step maintains a process utilization of 63.7%, compared to 53%, 32.7% and 27.5% for the earlier implementations.

In figure 45 we compare the four implementations of the algorithm in terms of speedup. The final version of the algorithm is initially slower than the first version due to the extra storing and retrieving of the (F,S) pairs from the data stack. However, as the parallelism increases, the final version of the algorithm outperforms the three previous versions, speeding up the execution of the algorithm by a factor of 4.29, compared to 3.54, 3.48, and 2.92 for the previous versions.

Again, performance has been improved by increasing process utilization. In this version, the increase in utilization was achieved by sequencing the two sub-tasks asynchronously instead of synchronously. Thus, not only were individual sub-tasks performed in parallel by task forces of processes, but also two sub-tasks were processed simultaneously. If a process could not find work to perform in the CHECKNEXT task force, it looked for work to perform in the CHECKSTATE task force.

Unfortunately, this method of enhancing performance by increasing parallelism only partially solves the problem of not enough data in the data stream. Those processes that

[1] If sub-task(j) takes as input the output of sub-task(i), and if sub-task(j) does not have to wait for sub-task(i) to be completed before it can begin, then the control structure sequencing sub-task(j) after sub-task(i) is an asynchronous control structure.

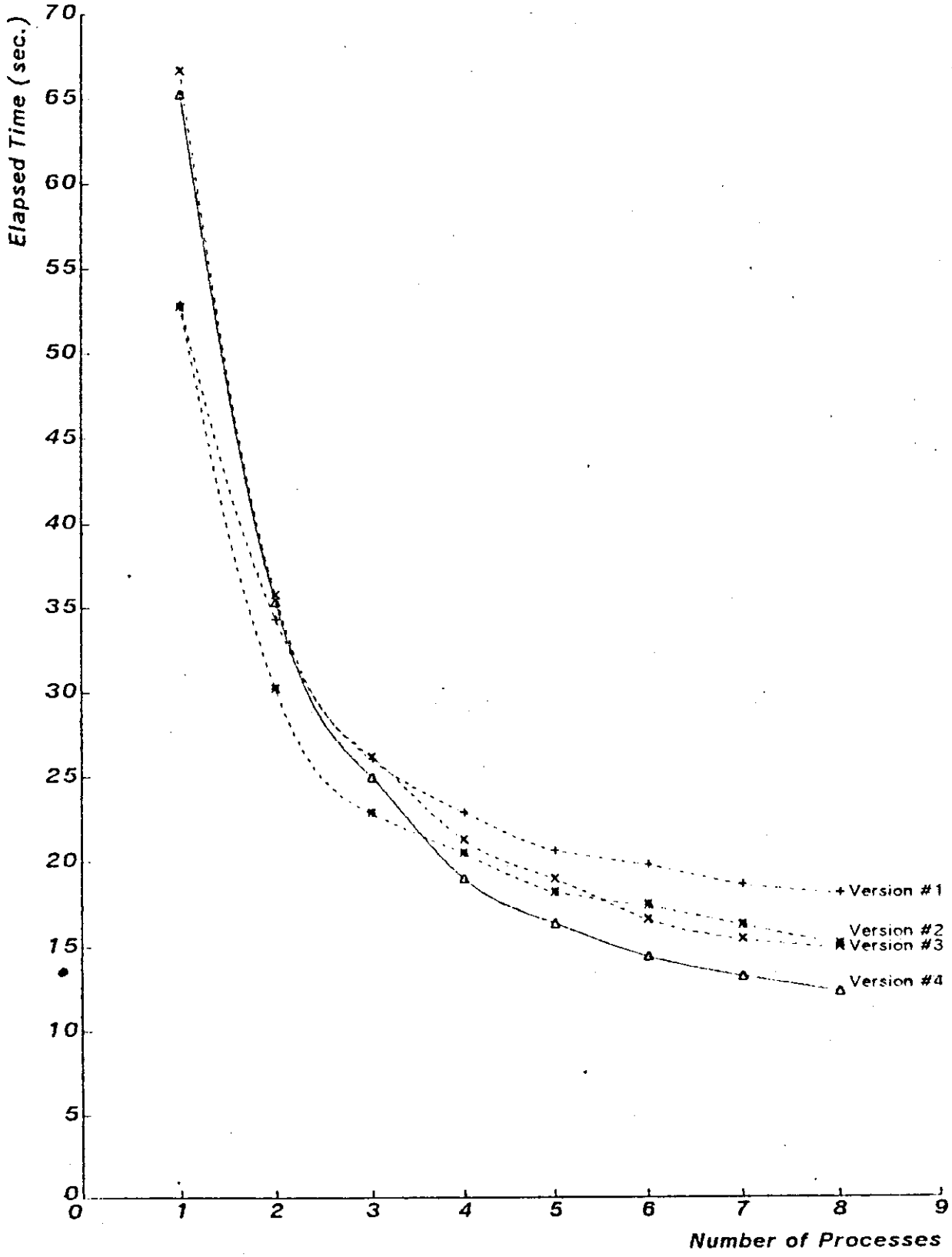


Figure 43 Decomposition of the Forward Step-- Version #4

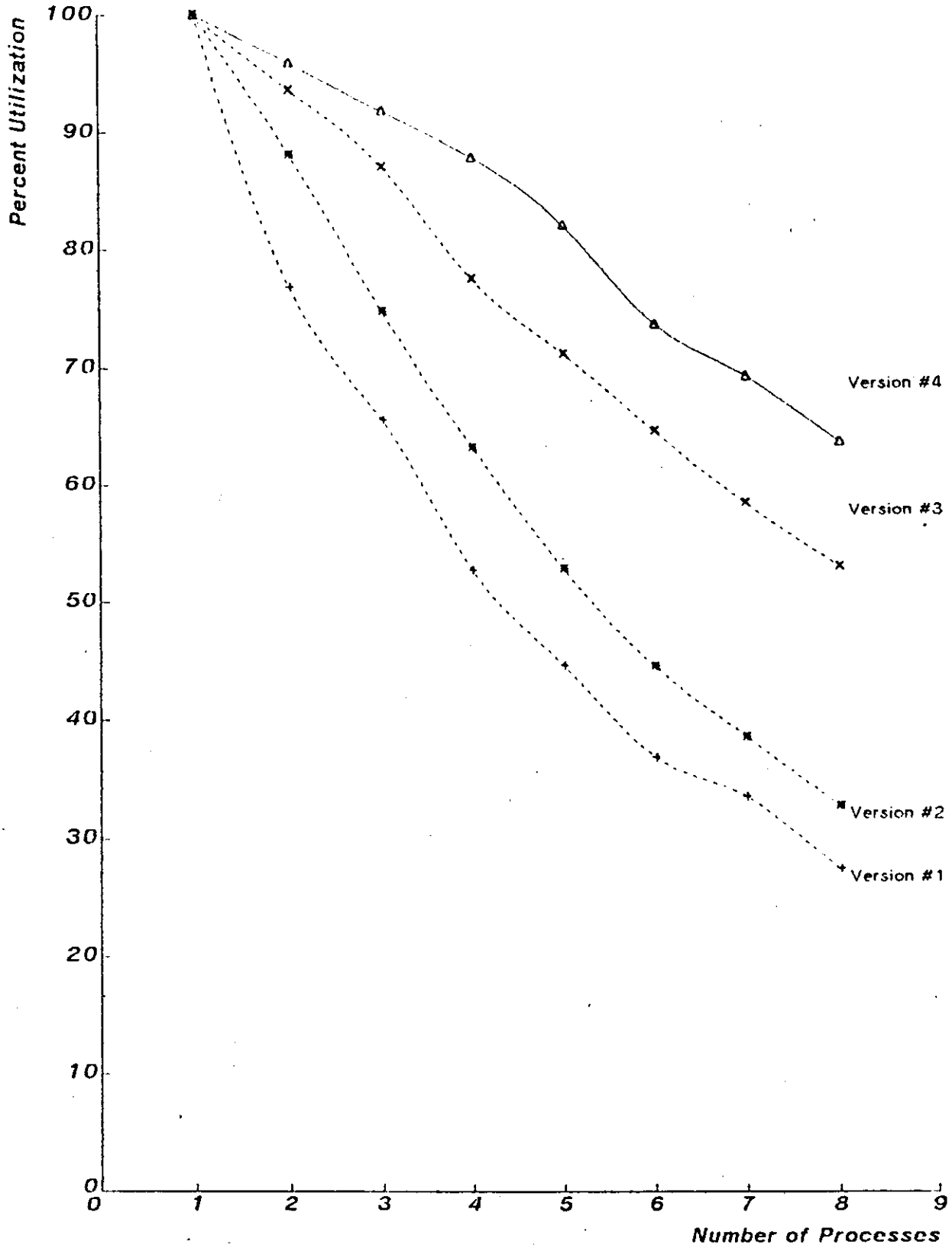


Figure 44 Decomposition of the Forward Step-- Version #4

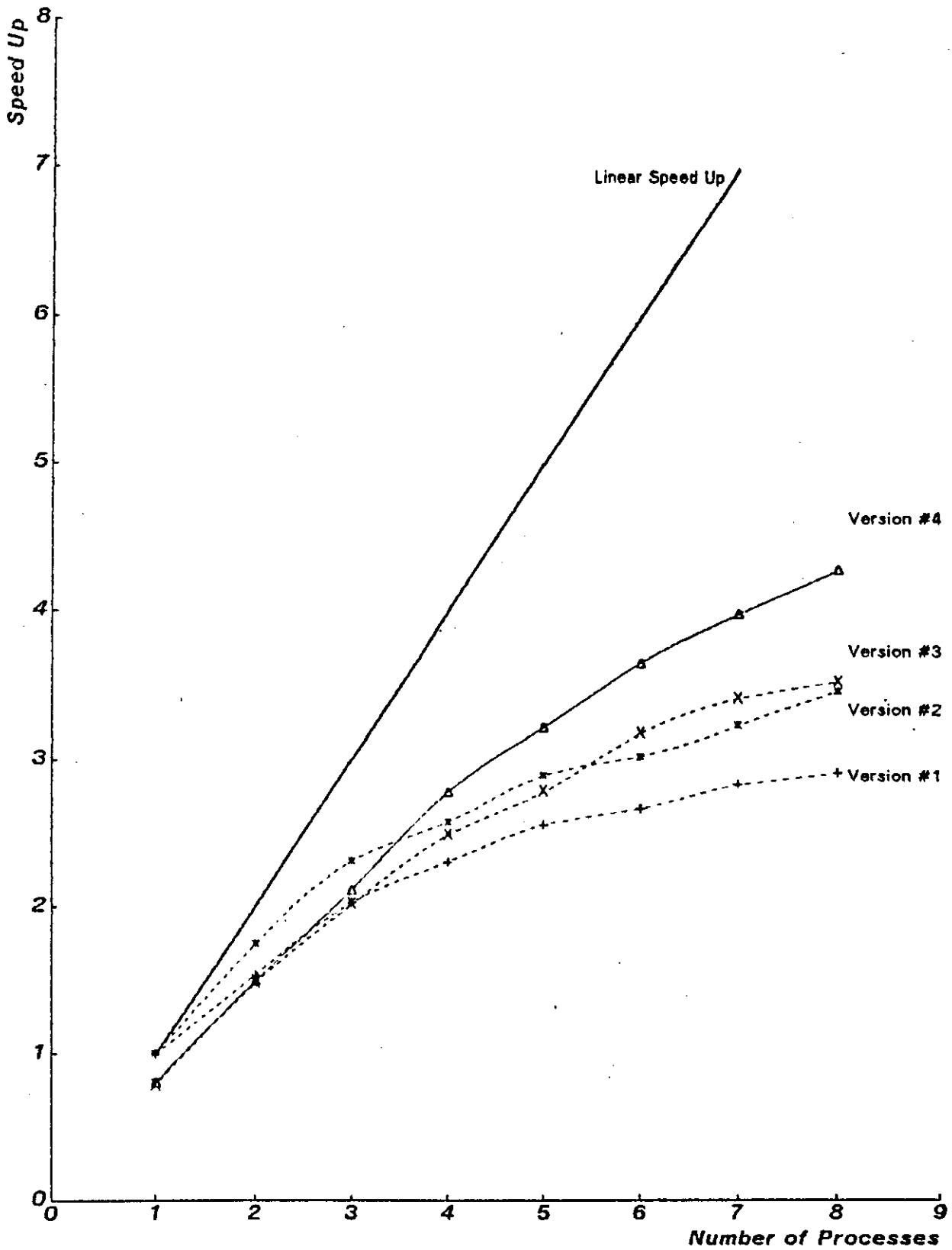


Figure 45 Decomposition of the Forward Step-- Version #4

cannot find any work to perform in the CHECKNEXT task force are not guaranteed to find work by becoming part of the CHECKSTATE task force. In addition, although further subdividing the forward step into smaller sub-tasks will increase process utilization by creating more work units, it will also introduce new overheads in manipulating the data items. At some point, the overheads in manipulating the new work units will outweigh the performance improvement resulting from higher process utilization. This investigation, to locate the optimum number of sub-tasks, is beyond the scope of this study.

The performance of all four versions of the forward step is summarized in the table below.

Measure	Version #4	Version #3	Version #2	Version #1
Elapsed Time	12.339	14.966	15.206	18.148
Pc Utilization	63.69%	52.99%	32.70%	27.46%
Speedup	5.295	4.456	3.471	2.94

5.6 Summary

5.6.1 Comparing the Four Versions of the Algorithm

The performance of the initial implementation was discussed in detail, uncovering several problems limiting the performance of the algorithm. In the three subsequent implementations, enhancements to the algorithm were directed towards eliminating the performance problems of the initial version.

In the initial parallel version of the algorithm, statically pre-allocating an equal number of candidate states to each process resulted in under utilization of the processes for two reasons: the compute time to process a candidate state was not a constant, and the number of candidates per segment of speech was quite often less than the number of processes.

The first enhancement to the algorithm was to dynamically allocate the candidate states among the processes. This prevented one process from developing a backlog of unstarted work while other processes were forced to remain idle. A 16% reduction in the elapsed time to perform the forward step of the algorithm resulted. This technique solved the problem of unequal workload allocation only when there were many candidate states to be processed. When the number of candidates was small, almost two-thirds of the speech segments had less than ten candidate states, under utilization of the processes still resulted.

In the second enhancement to the algorithm, the sub-task performing the forward step was split into two smaller sub-tasks in order to increase process utilization. Dividing the

computation into two separate phases increased process utilization by breaking the relatively small number of computationaly large work units into many smaller, less complex units. Although process utilization increased by 20%, the additional overhead in sharing the new data items and in synchronizing the processes between the two sub-tasks, eliminated any substantial elapsed time improvement.

In the final implementation, the control strategy synchronizing the processes between the two sub-tasks was changed from synchronous to asynchronous. Processes that could not find a candidate state for processing in the CHECKNEXT sub-tasks, migrated to the CHECKSTATE sub-task to start processing the (F,S) pairs without waiting for the rest of the task force to finish the CHECKNEXT sub-task. This enhancement increased process utilization to approximately 64%. However, unlike the previous implementation, a sizable improvement in the elapsed time to perform the forward step was realized; a 17.5% reduction to 12.3 seconds.

5.6.2 A Final Comparison-- The Uniprocessor Algorithm

Up to this point we have confined our performance comparison to the alternative implementations of HARPY on C.mmp. To conclude this investigation, a comparison between a parallel version of the algorithm written for C.mmp and the uniprocessor version of the algorithm written for a DEC KL10 is presented.

In figure 46, the performance of the two machines is compared in terms of the elapsed time to recognize fifteen utterances. The KL10 recognizes the fifteen utterances in approximately 49 seconds. The single process instantiation of the C.mmp version performs the same task in approximately 144 seconds, almost three times slower than the KL10. However, as additional processes are incorporated into the algorithm, the elapsed time to perform the task is sharply reduced. At four processes, C.mmp outperforms the KL10, requiring only 46 seconds to perform the task. At seven processes, maximum measured parallelism, C.mmp is recognizing the fifteen utterances in only 33 seconds, over 30% faster than the large uniprocessor.

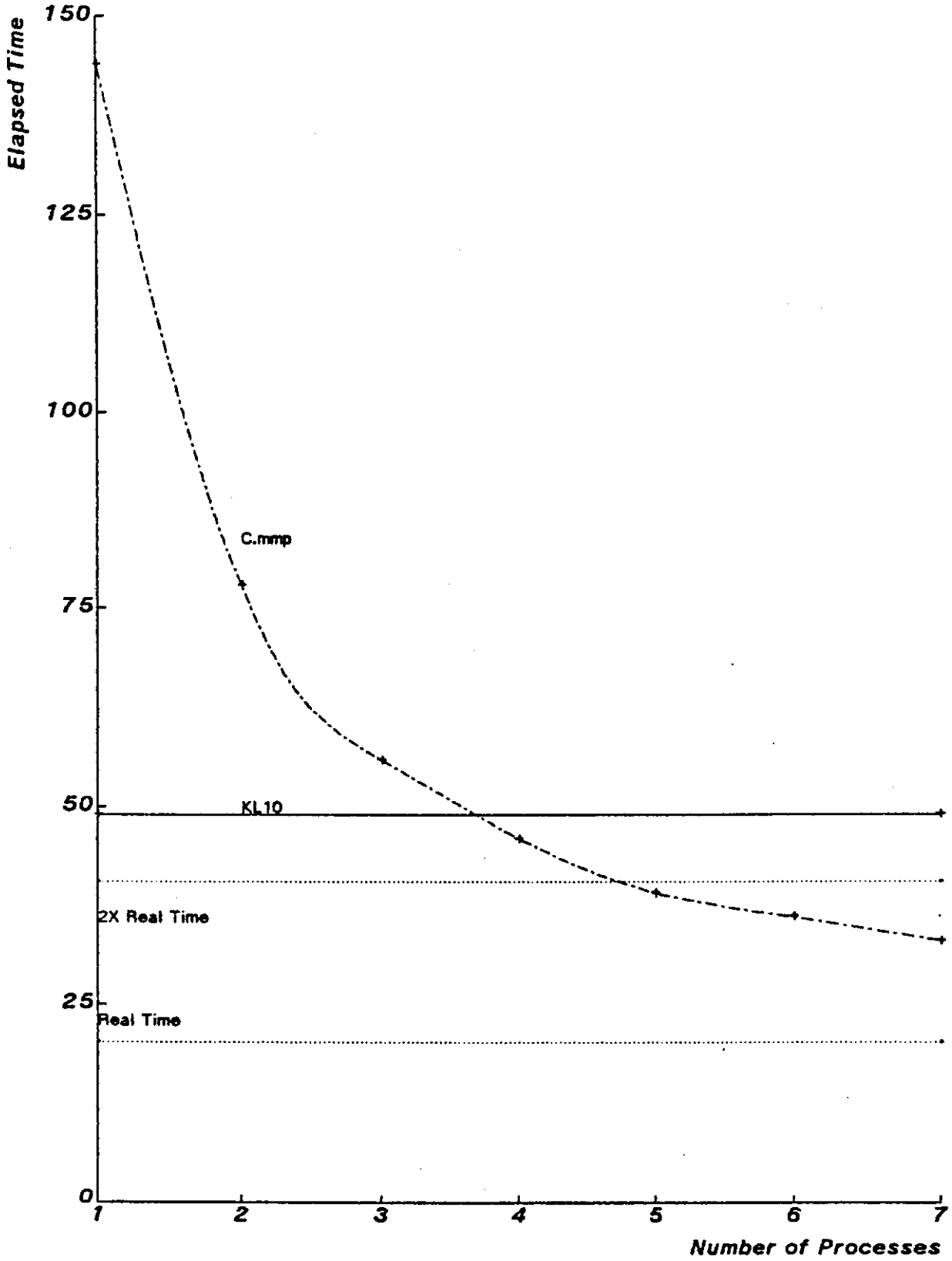


Figure 46 C.mmp vs. KL10 Harpy 1000 Word Task (LAAX05)

6. The Results and Contributions of this Investigation

6.1 A Summary of the Measurements and Results

6.1.1 The Initial Investigation-- The Rootfinder

In order to transform a parallel algorithm into an effective running program on a multiprocessor, one must be aware of the ways the system can affect the performance of the program. To uncover the major sources of performance perturbation, a simple program, a parallel rootfinding algorithm, was developed to act as a vehicle for conducting the study.

The performance of the program was perturbed by a variety of sources. Performance perturbations stemming from the hardware, both the processors and the memories, were identified and measured. Speed variations of individual processors and memories had only a secondary effect on performance. The greatest hardware related perturbation was a 300% performance degradation that was found to be a direct result of central memory bandwidth limitations.

Operating system performance perturbations arose from two sources: interrupts from I/O devices affected the program's performance by randomly interrupting the cooperating processes for short periods. These interrupted processes arrived at the synchronization point later than their uninterrupted counterparts, delaying the entire collection of processes from proceeding. The effect was graphically illustrated with a sample execution trace produced by a software monitor within the operating system. Bottlenecks in the operating system's scheduling processes also caused serious performance degradations in certain situations.

A third source of variability is the function evaluation. The computation time for performing the function evaluation is not a constant, but instead varies with the selection of the evaluation point. Because the processes must synchronize after every iteration, the elapsed time for an iteration is determined by the process with the maximum computation time. Thus, the variance in the distribution of the computation time for performing the function evaluation will greatly affect the performance of the rootfinding processes. A large variance results in only a small speed up, whereas a small variance results in a larger speed up.

Special attention was paid to the synchronization of the cooperating processes because it is a fundamental programming issue in the multiprocessor environment. Our investigation

consisted of a detailed measurement of the performance of several alternative synchronization primitives. We then incorporated each primitive into the rootfinding procedure to perform the necessary interprocess communication. By measuring the performance of the rootfinding program, a range of usefulness was determined for each synchronization primitive. The inter-synchronization time thresholds when a particular primitive became useful varied from 200 milliseconds to 2 milliseconds.

6.1.2 The Implementation of a Complex Task-- The Harpy Speech Recognition System

Using the insight into the C.mmp environment acquired during the initial rootfinding study, a more complex task, the Harpy speech recognition system, was developed on the multiprocessor. Harpy, an algorithm that recognizes connected speech from a variety of speakers, was initially developed at CMU for a uniprocessor. A parallel version of the algorithm was developed by decomposing Harpy into simpler sub-tasks, and then implementing these sub-tasks as task forces of identical processes. The task forces of identical processes speed up the algorithm by dividing the work into independent partitions for simultaneously processing.

In any decomposition involving cooperating processes, two implementation issues arise: how the processes acquire and share data, and how the processes are sequenced and controlled. Data can either be allocated statically, if the processes are given private partitions of data prior to their execution, or dynamically if the processes compete for or share all the data. Similarly, two alternatives for process control are synchronous and asynchronous sequencing. If all the cooperating processes must arrive at the synchronization point before the next step or sub-task can begin, then the processes are sequenced by a synchronous control structure. If, a process is not required to wait for its companions at the synchronization point, then the processes are sequenced by an asynchronous control structure. For both of these issues the two alternatives were discussed and measured in the implementations of Harpy's cooperating processes.

Four alternative implementations of Harpy were investigated. Rather than examining the variations in performance stemming from algorithmic modifications, this investigation measured and evaluated the performance variations arising from modifications related to only the implementation of one algorithm. The performance of the four implementations is compared in chapter five. Refining the algorithm in four implementations gave us the opportunity to observe and measure the performance ramifications of several implementation decisions.

The performance of the four implementations varied substantially, demonstrating the importance of an effective implementation. In the initial implementation a straightforward

decomposition of the uniprocessor algorithm, the elapsed time to perform the task was reduced from 52.89 seconds to 18.15 seconds when eight processes were incorporated into the algorithm. This corresponds to a speed up of only 2.92. In the final implementation this elapsed time was reduced to only 12.33 seconds, which corresponds to a speed up of 4.29. The improvement resulted from an increase in process utilization, the percentage of time a process is performing useful work. Balancing the computational work load across all processes increased the process utilization from 27.5% to 64%.

The best multiprocessor implementation of the algorithm was compared to a sequential implementation of the algorithm designed for a large uniprocessor, a DEC KL10. Initially, the KL10 outperformed a single process instantiation of the multiprocessor implementation by almost a factor of three. However, as more processes were incorporated into the task forces, the C.mmp version outperformed the uniprocessor at four processes and was observed to be 30% faster at the maximum measured parallelism, seven processes.

6.2 The Task Force Approach to Parallel Programming

The measurements and results presented in this investigation demonstrate that the task force approach to writing parallel programs is an effective method for capturing parallelism. As with any programming technique, certain benefits and drawbacks are associated with its use.

The programming effort required to write parallel programs is not much more than the effort needed to write serial programs. By introducing parallelism through replication, the programmer is required to write only a single program, not n different programs. The sharing of data and the synchronization of cooperating processes are well understood problems easily solved without special programming language parallel constructs. Harpy was implemented entirely in BLISS-11, without any special language constructs to coordinate the data sharing, sequencing, or synchronization of the processes.

The task force technique is a general approach to parallel programming; its application is not restricted to only a few special situations. Those tasks that involve the repeated application of functions on data are ideally suited for parallel implementation using the task force approach. The rootfinding algorithm and the Harpy speech recognition system are two dissimilar representatives of this large class of algorithms.

However, the most important aspect of the task force technique is that it is effective at introducing linear speedup into an algorithm. Although linear speedup of the Harpy algorithm was not demonstrated, portions of the data streams were processed by the task forces n times faster than if performed by a single process. Only when work was unavailable to keep

all the processes busy did performance drop below linear speedup. The task force technique tends to favor data streams composed of many elements over those with only few elements in them. Thus, performance can be improved, and in fact can approach linear speedup, simply by increasing the number of work units in the data streams. For example, Harpy's performance could be improved by increasing the complexity of the grammar from which utterances are constructed.

The major drawback to using this approach is that for it to be successful, the programmer must be aware of several primitive "time-constants", i.e. the algorithm's inter-synchronization times, and the synchronization primitive's elapsed times, that characterizes the hardware, and the operating system, and his own algorithm. This requirement runs counter to the popular idea of programming without the need to know about the underlying environment.

6.3 Areas for Further Research

One aspect of the implementation of parallel programs not addressed in this study is the performance degradations caused by a small address space. Despite the fact that the central memory supports up to 32M bytes of primary memory, the PDP-11 is a 16-bit minicomputer and as such limits addresses to only 16 bits. Thus a process can directly address only 64K bytes of primary memory at a time. Initially, it was felt that the small address space limitation would be offset by the ability to create multiple processes, each addressing only a small portion of the total address space. This assumption about the organization of parallel programs is not always true.

For example, in our implementations of Harpy we totally ignored the impact of the small address space problem on the algorithm's performance. If a data item resided outside the process' addressable region, we simply payed the overhead to make it addressable, i.e. a relocation register load. In an early investigation to measure this overhead, we observed in one case a factor of three degradation in the algorithm's performance.

One technique to minimize this small address space problem is to construct data structures so that memory locations tend to be accessed either sequentially or in small clusters. We would expect some improvement in Harpy's performance if we allocated storage for the transistion network such that directly related states were close together.

Obviously, the entire issue of the small address space can be avoided in future multiple computer systems by using larger address space machines as the central processors.

Another area for future research is the investigation of the performance of the multiprocessor when it functions as a general computing facility for multiple users. It was

felt that one important mode of operation would be for C.mmp to support the simultaneous execution of many single process tasks from multiple users. It has been suggested that the multiprocessor is best suited for this type of parallelism. However, little evidence exists to substantiate this claim.

In conclusion, this investigation is only one of the first of many such studies to assess the effectiveness of the multiprocessor. The primary contributions of this study are that it provides several initial data points in the measurement space of multiprocessors, and that some aspects of the implementation of parallel programs are illuminated through the analysis of several example programming efforts.

Appendix

Artificial Intelligence Information Retrieval Task (LAA)

1. Please help me
2. What should I ask
3. What can the system do
4. The first two
5. Give me one more please
6. Thank you I'm done
7. Stop transmitting please
8. Who wrote it
9. Who was the author
10. What was its title
11. When was it published
12. What about Minsky
13. Which is the oldest
14. What facts are stored
15. Please list the authors
16. Print the next one
17. Where does he work
18. What is her affiliation
19. What about formal semantics
20. What about program verification

BIBLIOGRAPHY

[Avriel and Wilde 66] Avriel, M., and Wilde, D.J., "Optimal Search for a Maximum with Sequences of Simultaneous Function Evaluations," Management Science, 12, 1966, pp. 722-731.

[Baudet, Brent and Kung 77] Baudet, G., Brent, R.P., and Kung, H.T., "Parallel Execution of a Sequence of Tasks om an Asynchronous Multiprocessor," Carnegie-Mellon University, Computer Science Dept., Tech. Report. June 1977.

[Baudet 78] Baudet, G., "The Design and Analysis of Algorithms for Asynchronous Multiprocessors," Ph.D. Thesis, Carnegie-Mellon University, Computer Science Dept., April 1978.

[Fuller and Oleinick 76] Fuller, S.H. and Oleinick, P.N., "Initial Measurements of Parallel Programs on a Multi-Mini-Processor," IEEE Fall Compcon 76, pp. 358-363.

[Fuller 1978] Fuller S.H., Ousterhout J.K., Rubinfeld P.I., Sindhu P.J., Swan R.J., "Multi-Microprocessors: An Overview and Working Example," Proc. IEEE Vol.66, No.2, February 1978, pp. 216-228.

[Heller 76] Heller, D., "A Survey of Parallel Algorithms in Numerical Linear Algebra," Carnegie-Mellon University, Computer Science Dept., Technical Report, 1976.

[Jones 78] Jones, A.K., Chansler, R.J., Durham, I., Feiler, P.H., Scelza, D.A., Schwanz, K. and Vegdahl, S.R., "Programming Issues Raised by a Multiprocessor," Proc. of the IEEE, Vol 66 No.2, February 1978, pp. 229-237.

[Karp and Miranker 68] Karp, R.M., and Miranker, W.L., "Parallel Minimax Search for a Maximum," J. Comb. Theory 4, 1968, pp. 19-35.

[Kung 1976] Kung H.T., "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors, Algorithms and Complexity: Recent Results and New Directions," ed. J.F.Traub 1976, pp. 153-200.

[Lesser 75] Lesser, V.R., "Parallel Processing in Speech Understanding Systems," Speech Recognition 1975, pp. 481-499.

[Levin 1975] Levin R., Cohen E., Corwin W., Pollack F., Wulf W.A., "Policy/Mechanism Separation in HYDRA," Proceedings of the ACM/SIGOPS Symposium on Operating Systems Principles, Austin Texas, November 1975, pp. 132-140.

[Lowerre 76] Lowerre, B., "The HARPY Speech Recognition System," Ph.D. Thesis, Carnegie-Mellon University, Computer Science Dept., 1976.

[Lowerre and Reddy 77] Lowerre, B. and Reddy, R., HARPY Speech Understanding System (1977), produced at Carnegie-Mellon University. An 18-Minute 16mm./Color/Sound Film describing the HARPY SUS developed by Lowerre and Reddy.

[Newell and Robertson 1975] Newell A., and Robertson G., "Some Issues in Programming Multi-Mini-Processors," Tech. Rep., Computer Science Dept.,

Carnegie-Mellon University, Pittsburgh, Pa., January 1975

[Rosenfeld and Driscoll 69] Rosenfeld, J.L. and Driscoll, G.C., "Solution of the Dirichlet Problem on a Simulated Parallel Processing System," Information Processing 68, North-Holland Publishing Co., Amsterdam, 1969, pp. 499-507.

[Stone 1973] Stone H.S., "Problems of Parallel Computation, Complexity of Sequential and Parallel Numerical Algorithms," ed. J.F. Traub, Academic Press 1973, pp. 1-16.

[Swan, Fuller and Siewiorek 77] Swan, R.J., Fuller, S.H. and Siewiorek, D.P., "CM*: A Modular Multi-microprocessor," Proc. AFIPS 1977, National Computer Conference, Vol. 46, 1977, pp. 637-644.

[Teichroew 1956] Teichroew D., "Tables of Expected Values of Order Statistics and Products of Order Statistics for Samples of Size Twenty or Less from the Normal Distribution," The Annals of Mathematical Statistics 27,2, June 1956, pp 410-426.

[Thompson and Kung 76] Thompson, C.D. and Kung, H.T., "Sorting on a Mesh-Connected Parallel Computer," Proc. 8th Annual ACM Symposium on Theory of Computing, 1976, pp. 58-64. Also to appear in Communications of the ACM.

[Wulf and Bell 1972] Wulf W.A., and Bell C.G., "C.mmp -- A Multi-Mini-Processor," Proceedings AFIPS 1972, FJCC Vol 41. AFIPS Press, pp. 765-777.

[Wulf 1974] Wulf W.A., Cohen E., Corwin W., Jones A., Levin R., Pierson C., Pollack F., "HYDRA: The Kernel of a Multiprocessor Operating System," Communications of the ACM, 17,6, 1974, pp. 337-345.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: CMU-CS-78-151	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): THE IMPLEMENTATION AND EVALUATION OF PARALLEL ALGORITHMS ON C.MMP	5. TYPE OF REPORT & PERIOD COVERED: Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): PETER N. OLEINICK	8. CONTRACT OR GRANT NUMBER(s): N00014-77-C-0500
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Carnegie-Mellon University, Computer Science Dept., Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Arlington, VA 22217	12. REPORT DATE: November 1978 13. NUMBER OF PAGES: 105
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same as above	15. SECURITY CLASS. (of this report): UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)