1988

Concurrency Enhancement Through Program Unification: A Performance Analysis

Vernon J. Rego
Purdue University, rego@cs.purdue.edu

Aditya P. Mathur
Purdue University, apm@cs.purdue.edu

Report Number:
88-739

CONCURRENCY ENHANCEMENT THROUGH PROGRAM UNIFICATION: A PERFORMANCE ANALYSIS

Vernon Rego
Aditya P. Mathur

CSD-TR-739
January 1988

1. Introduction

Vector supercomputers, such as the Cray X/MP and the Cyber 205, and vector mini-supercomputers such as the Alliant FX/8 or the SCS 40, perform best when a program under execution is able to fully exploit vector pipes and multiple processors (the latter facility on multiprocessors only). Compilers are now available on many of these machines for exploiting such architectural features. These compilers detect vector loops and generate vector [2,3,8] and/or concurrent codes [1,5,11]. If the compiler is unable to vectorize and/or concurrentize a loop, the user has the option of providing the additional information to enable the compiler to perform the necessary optimization. In many programs, however, the code itself is inherently scalar and does not vectorize, or fails to vectorize due to *data dependencies*

In another class of programs, the vector loops are typically too short to provide any noticeable speedup† over the corresponding non-vectorized code. In [6] a technique, known as *program unification*, was presented to overcome the above cited problems. Benchmarks presented in [6] clearly showed the utility of the technique. In this paper we present a formal analysis of program unification. More specifically, we present:

1. A class of *Urn models* for the behaviour of a *unified program*
2. Analytic formulae to compute the *speedup* that can be obtained by using the transformed program instead of using the original program.
3. Analytic and simulation results of the effects of different path management alternatives on the speedup.

The remainder of this paper is organized into six sections. The next section provides the motivation for the present work. Section 3 summarizes the program unification technique that is described fully in [6]. In section 4 we present an analysis of th execution time for program P, and in section 5 this is extended to an analysis of $\tilde{P}$ via an urn model scenario. The results of our analysis are given in section 6. Finally, section 7 contains a brief outline of ideas for future work. Briefly, the analysis demonstrates that average speedup is given by the ratio of expected times to absorption in two different Markov chains. Once these Markov chains are constructed, it is shown that there is much information to be gained about speedup. However, due to space limitations, we restrict our graphical results to displaying averages.

† For a precise definition of *speedup* refer to section 3.5.

2. Motivation for Concurrency Enhancement

Typical vectorization techniques examine the source code for statements that are potential candidates for vectorization. *Program Unification* is useful when these techniques fail. It induces concurrency by merging together several instances of a program that is desired to be executed over multiple data sets. One can find problems, in several areas of computer applications, where a program P is executed, successively, over different data sets. A representative list of such applications appears in [5]. We expect that these and similar applications are potential beneficiaries of our technique.

The problem of analyzing the program unification technique in order to estimate the degree of speedup is a formidable one. It can be mapped into a problem of comparing Markov chains, in order to determine, from their behaviour, a particular chain or set of chains that possesses a given property. The latter problem is interesting in itself and raises a number of questions concerning the comparison of Markov chains, with solutions that are highly applicable in a variety of settings. The urn model formulation in section 4 is actually seen to lead to a sequential occupancy problem, and the question of comparison crops up when optimal sequential occupancy rules are required. We expect that the current formulation will give us some insight into the asymptotic behaviour of such systems and heuristic solutions to path management that are close to optimal.

3. Program Unification: The Technique

In this section we briefly describe the program transformation technique which we have analyzed in this paper. For details, the reader is referred to [6]. We will denote by P the program that is to be executed on a vector uniprocessor. Let $d_1, d_2, \ldots, d_N$ denote N independent data sets over which P is to be executed. Let $P_1, P_2, \ldots, P_N$ denote instances of P when P executes over data sets $d_1, d_2, \ldots, d_N$, respectively. We shall soon show how it is possible to combine the instances of P into one program that executes over all the N data sets *concurrently*. We shall denote the transformed program† by $\vec{P}$. While describing the execution of $\vec{P}$, we shall refer to P_i, an instance of P, as the i^{th} component of the merged program $\vec{P}$.

The benchmarks, presented in [6] show that the time taken to execute multiple instances of P is, in many cases, significantly *greater* than the time taken to execute $\vec{P}$. These benchmarks therefore favour executing $\vec{P}$ rather than executing multiple instances of P.

---

† We call $\vec{P}$ a *vectorized program* due to the fact that the multiple data sets appear as *vector inputs* to $\vec{P}$. One may also call $\vec{P}$ a *merged program*.

3.1 Multiple execution paths

It is evident that though the source codes of all instances of P are identical, the paths that are followed when these codes are executed depend on the input data sets. As $\vec{P}$ simulates the execution of all instances concurrently on a uniprocessor* , it must be able to handle the situation that arises when its components follow different paths.

The problem of dealing with multiple paths can be described by a simple example. Suppose that P contains the following statement:

$$X = Y + Z$$

Then, each instance of P would also contain the above statement. However, when these instances are executed, some might execute this statement and others might not, implying that X may or may not be assigned during the execution of a program instance.

3.2 The Partition vector

To take care of the problem that arises when at least one instance follows a path different from the others, we introduce a *partition vector* denoted by PV. PV is an N element logical vector used by $\vec{P}$. For each component of $\vec{P}$ there is one element in PV. A component of $\vec{P}$ is said to be *active* if $PV(i)$ is *true*, otherwise the component is said to be *inactive*.

PV is used by $\vec{P}$ to ensure that an assignment is not made to a variable, such as X in the above example, if it is not in the path of a component of $\vec{P}$. PV is thus used as a *mask* for all assignments in $\vec{P}$ as described in the next section. *Masked execution* of statements is a feature available on all widely used vector processors.

3.3 The Pending List

At any point of time during the execution of $\vec{P}$, it is possible to classify its components as either *in execution* (active) or *pending* (inactive). The ones in execution are also said to be *in-step*. As mentioned earlier, the partition vector PV indicates which component belongs to each of these two sets. $PV(i) = true$ implies that component i is in-step with all those components j, $1 \le j \le N$, and $j \ne i$, for which $PV(j)$ is true. Similarly, $PV(i) = false$ implies that component i is currently pending execution.

---

* The method, as described in this paper, does not explicitly exploit the advantages of a *multiprocessor* like the Cray X/MP.

All components which are pending execution are recorded in a list which is termed the *pending list*† . Each entry in the pending list is a pair referred to as *(id, st)*, where *id* is the component identifier and *st* is the statement number in $\bar{P}$ from where the corresponding component shall resume execution when it gets out of the pending list.

3.4 Block selection policies

Throughout our analysis in this paper, we assume, without loss of generality, that the given program P is in *normalized form*. Thus, each statement of P can be classified into one of the following:

1. An assignment statement.
2. A conditional or an unconditional branch.
3. A program termination statement, such as a **RETURN** in Fortran.

In order to model the execution profile of P, we define the notion of a *block*. A block of length l in P is a sequence of statements $S_1 , S_2 , \ldots S_l$. The first statement of a block, S_1, is one or more of the following types:

1. First statement of the program.
2. A statement that is the target of an unconditional or a conditional branch.
3. A statement that immediately follows a conditional or an unconditional branch or a program termination statement.

The last statement of a block, S_l, is one or more of the following types:

1. Last statement of P.
2. A conditional or an unconditional branch statement.
3. A program termination statement.

It is now possible to characterize P as a sequence of blocks denoted by $B_1 , B_2 , \ldots , B_K$. When P is transformed, using the technique described in [5], the transformed program, denoted by $\bar{P}$,

† The actual implementation details of the *pending list* may be found in [4].

consists of exactly the same number of blocks as in P. We refer to the blocks of $\vec{P}$ as $\vec{B}_1, \vec{B}_2, \ldots, \vec{B}_K$.

During the execution of $\vec{P}$, exactly one of the K blocks is executed by one or more program components when $\vec{P}$ is executing on a uniprocessor. The other components wait for their turn at some other blocks. As mentioned above, the information regarding which program component is waiting on which block, is recorded in the *pending list.*

On completion of the execution of one block, the next block to be executed needs to be determined. There are several policies which could be used to select the next block for execution. Such a policy is referred to as a *block selection policy.* Three block selection policies were proposed in [5], and a fourth proposed in this paper. Assuming that the program components of $\vec{P}$ put forward distinct *candidate blocks* $B_{j_1}, B_{j_2}, \ldots, B_{j_k}$, for execution, with $k \leq K$, the four different selection policies are:

Complete First Policy: This policy selects block B_m such that $m = \min \{j_1, j_2, \ldots, j_k\}$.

Move Forward Policy: This policy selects block B_m such that $m = \max \{j_1, j_2, \ldots, j_k\}$.

Majority Rule Policy: This policy selects a block that has the maximum number of program components waiting to execute it. In the case of a tie, the block with least index is selected.

Random Choice Policy: This policy selects one block from the possible candidate blocks *randomly* (i.e., uniformly).

3.5 Block and Program Speedups

For ease of discussion, we refer to block B_j simply as block j, for $1 \leq j \leq K$. Similarly, block $\vec{B}_j$ is also referred to as block j, $1 \leq j \leq K$, and any reference we make to blocks of P and blocks of $\vec{P}$ should be clear from context. We now describe the speedup that is obtained by executing a number of program blocks concurrently (i.e., executing $\vec{P}$) versus their sequential execution (i.e., N executions of P).

Block speedup in $\vec{P}$

When any instance of program P is executed on a vector uniprocessor, we assume that each block j takes t_j time units to complete execution, $1 \leq j \leq K$. Thus the serial execution of P_1 through P_N would require time $N \cdot t_j$ to execute block j, if each of these programs executes block j once. Consider now the time required to perform the same step by the transformed program $\vec{P}$. Since the components of $\vec{P}$ now utilize the vector capability more efficiently than a single instance of program P, the time taken for $\vec{P}$ to execute block j once (effectively performing the

serial executions on block j in a single step, provided all N components are set to execute block j concurrently) is expected to be much smaller (see [5,6]) than in the serial case. Formally, for each block j, let t_j be the time taken by $\vec{P}$ to execute block j, $1 \le j \le K$. For each block j, $1 \le j \le K$, we define the inverse block speedup, α_j, as

$$\alpha_j = \frac{t_j}{N \cdot t_j} \tag{3.5.1}$$

and the *average* inverse block speedup as $\alpha = \sum_{j=1}^{K} \pi_j \alpha_j$. Empirical data suggests that for most types of blocks α is a decreasing function of the number of programs N, suggesting a drastic reduction in execution time of $\vec{P}$ with increasing N (see section 6.1).

Speedup given by $\vec{P}$

Given data sets $\{d_1, d_2, \ldots, d_N\}$, define the *speedup* obtained by executing $\vec{P}$ once, as opposed to N executions of P, as

$$\hat{S}_{\vec{P}} = \frac{\sum_{j=1}^{N} t(P_j)}{t(\vec{P})} \tag{3.5.2}$$

where $t(P_j)$ and $t(\vec{P})$ are the times required for the execution of P_j, $1 \le j \le N$, and $\vec{P}$, respectively. As defined, (3.5.2) yields an empirical estimate of speedup. However this is an exact value of speedup for the given data set. If we somehow managed to average results over all possible data sets, and denote the expected values of execution times for P and $\vec{P}$ as $E[T_P]$, and $E[T_{\vec{P}}]$, respectively, then *average speedup* over all data sets for a given program P can be defined as

$$S_{\vec{P}} = \frac{N \cdot E[T_P]}{E[T_{\vec{P}}]} \tag{3.5.3}$$

4. A Stochastic Model of Program Execution

Given an instance P_i of program P, the execution of P_i is purely deterministic. That is, if P is described as a flow graph (see Figure 1a), data set d_i determines the execution path of P_i uniquely. In general, for an arbitrary data set d_i, predicting the execution path and thus determining P_i's run-time characteristics *apriori* is a nontrivial issue. A simple way to obtain an idea of

P's behaviour is to construct a stochastic flow graph that represents the execution path of any P_i in an average sense.

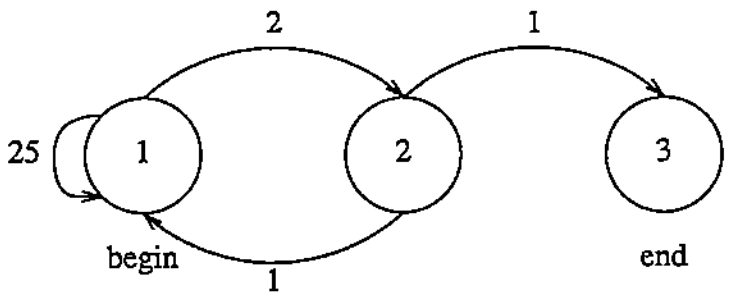


Figure 1a. Deterministic flow graph for P, with data set d_1 and $K = 3$.

4.1 Stochastic flow-graph representation

Let $D = \{d_1^*, d_2^*, \ldots, d_n^*\}$ be a class of *trial* data sets, for sufficiently large n. On giving data set d_1^* as input to the flow graph of P, one obtains the execution path of P_1^*. Repeating this for all d_m^*, $1 < m \leq n$, yields a set of execution paths from which a stochastic flow graph can be constructed. In Figure 1a, we see that program P has $K = 3$ blocks (i.e., nodes) and each block of P a maximum outdegree value of two. Assuming that blocks of P are labelled $1, 2, \ldots, K$ (with 1 as initial block, and K as terminating block), let R_i denote the set of blocks that can be reached from block i *in one step* (i.e, R_i is the *reachability set of block i*). Thus for Figure 1a, $R_1 = \{1,2\}$, $R_2 = \{1,3\}$, and $R_3 = \Phi$. Given the execution path of P_m^*, for any m, $1 \leq m \leq n$, let $f_m(i, j)$ be the number of times that block j is visited from block i during execution of P_m^*, for $j \in R_i$. The probability estimate $\hat{p}_{ij}$ that the stochastic flow graph (see Figure 1b) will cause control to move from block i to block j during execution of P on *any* data set in the sampling domain of the trial data sets is

$$\hat{p}_{ij} = \frac{\sum_{m=1}^{n} f_m(i, j)}{\sum_{m=1}^{n} \sum_{j \in R_i} f_m(i, j)} \qquad (4.1.1)$$

for all i, j, $1 \leq i, j \leq K$. The stochastic graph in Figure 1b is constructed from the frequency

based graph in Figure 1a via an application of (4.1). For sufficiently large n, it is clear that $\hat{p}_{ij}$ is a reasonably good point estimator of the likelihood of control moving from block i to block j during execution of P on a data set that is *similar* to data sets in D. Henceforth, we refer to this estimate as probability p_{ij}. In the next two sections, we present urn model formulations for the serial execution of P and the concurrent execution of $\tilde{P}$. These models enable us to obtain an idea of how much is to be gained by resorting to the transformed program $\tilde{P}$ instead of P.

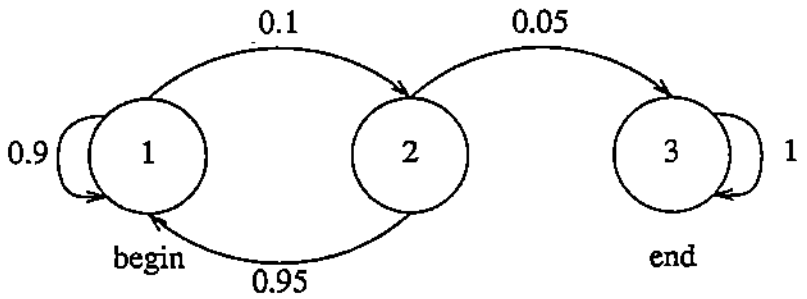


Figure 1b. Non-deterministic flow graph for P, with $K = 3$.

4.2 Modelling the execution of program P

The probabilistic flow graph depicted in Figure 1b is an alternative representation of a discrete time Markov chain $\{X_k ; k \geq 0\}$ defined over the space $\{1, 2, \ldots, K\}$ (see for example, [10]) with transition probability matrix $\mathrm{P} = [p_{ij}]$. Since we deal with terminating programs, in each case the chain $\{X_n ; n \geq 0\}$ will have initial state $X_0 = 1$ and absorbing state $X_k = K$, for some $k, k \geq 1$.

An easy way to think of P's execution on any data set d_i in terms of an urn model is as follows. Assume that we have K urns indexed 1 through K, arranged from left to right in increasing order of urn index (see Figure 2a). A single ball that is to represent the program P is initially placed in urn 1. The urn occupied by the ball at any given instant is taken to be the block currently being executed in P. Let t_j be the time taken to execute block j on the uniprocessor, $1 \leq j \leq K$. After t_1 units of time have elapsed, we pick up the ball from urn 1 and toss it into some other urn (or possibly the same urn) depending on P's stochastic flow graph. For example, in the case of Figure 1b, the choices would be the urns with indices in R_1, i.e., blocks 1 and 2,

with probabilities 0.9 and 0.1, respectively. We repeat this procedure until the ball finally reaches urn K (which in the case of our example is urn 3). After the ball remains for a period of t_K time units in urn K, the procedure stops, and program P is said to have terminated. Our immediate goal is to determine the amount of time that elapses from the instant that the ball is first tossed into urn 1, up to the instant that the ball expends t_K time units in urn K. This is the execution time of P on a single data set.

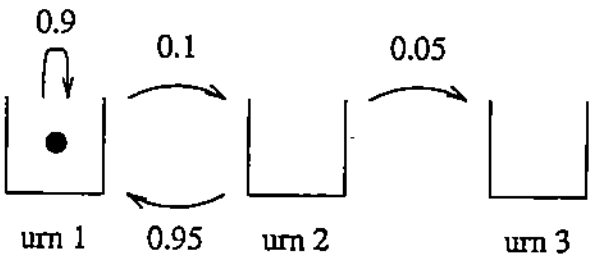


Figure 2a. Urn model for P, with $K = 3$.

4.3 The Execution Time for program P

Let T be a $(K-1) \times (K-1)$ substochastic matrix obtained from P by deleting its K^{th} row and K^{th} column. Define the K-dimensional row vector α and column vector β, as $\alpha = (1, 0, \ldots, 0)$ and $\beta = e - Te$, where e is a column vector with all entries set equal to one. The probability that program P requires $T = k$ *steps* (i.e., block executions) for its completion is precisely the time to absorption of the chain $\{X_k ; k \geq 0\}$, with initial vector $(\alpha, 0)$, is expressed as

$$Pr[T = k] = \alpha\, T^{k-1}\, \beta\,, \qquad k \geq 1\,. \tag{4.3.1}$$

That is, T is a random variable whose density is of phase-type [7], with the representation (α, T).

The density in (4.3.1) only gives the number of *steps* that P requires to complete execution. Since the block execution times of P are generally different for different blocks, (4.3.1) needs to be suitably modified to obtain an estimate of its execution time. Hence, T needs to be scaled by the average time taken by the uniprocessor to execute each block of P. In order to obtain this average, we first convert P to a nonabsorbing stochastic matrix P* by exchanging the

entries p_{K1} and p_{KK}. This effectively restarts the Markov chain in state 1 each time the program terminates, thus allowing us to observe a program that runs forever in order to calculate an average block execution time. Next we solve the linear system $\pi = \pi\, \mathrm{P}^*$ for the left invariant vector $\pi = <\pi_1, \ldots, \pi_K>$, where π_j is the steady state probability that program P will be seen executing block j, $1 \le j \le K$, if P is repeatedly executed. The scaled unit execution time for each block is now the average

$$t = \sum_{j=1}^{K} \pi_j\, t_j \tag{4.3.2}$$

and the estimated execution-time density is modified to $Pr[T_P = kt] = Pr[T = k]$, where T_P denotes the estimated execution time of P on the uniprocessor. Observe that this form of scaling gives us an approximation to the execution time of program P. We next go on to obtain the exact mean and variance of P's execution time.

Mean and Variance of Execution time

Let $\mathbf{Z} = (\mathbf{I} - \mathbf{T})^{-1}$ be the fundamental matrix corresponding to chain $\{X_n\,; n \ge 0\}$, with $\mathbf{Z} = [z_{ij}]$. The exact *mean* execution time of program P is given by

$$E[T_P] = t_K + \sum_{j=1}^{K-1} t_j\, z_{1,j}\ . \tag{4.3.3}$$

Define a matrix $\mathbf{Y} = \mathbf{P}\,[2\,diag\,(\mathbf{P}) - \mathbf{I}] - \mathbf{U}$, where U is a matrix obtained by squaring the entries in P. With $\mathbf{Y} = [y_{ij}]$, the exact *variance* of the execution time for program P is given by

$$V[T_P] = \sum_{j=1}^{K-1} t_j\, y_{1,j}\ . \tag{4.3.4}$$

Using the probability transition matrix P given by Figure 1b, and using block execution times $t_1 = 8$, $t_2 = 13$, and $t_3 = 7$, we obtain an average block execution time of 8.45 time units. This average can be used in the computation of an approximate execution-time distribution for P through (4.3.1). The mean execution time for P, as given by (4.3.3), is $E[T_P] = 1866.99$ time units, and the variance, as given by (4.3.4), is $V[T_P] = 323339.96$ time units.

Execution time distribution of N programs

It is finally left to determine the amount of time required to execute P_1 through P_N serially, for N data sets. If T_j is the execution-time random variable for program P_j, then T_1 through T_N are independent and identically distributed random variables since they represent independent execution times for the same program through the same stochastic graph. Hence the serial execution time T_S is a sum of i.i.d random variables,

$$T_S = T_1 + T_2 + \cdots + T_N \tag{4.3.5}$$

and the distribution of T_S is obtainable via convolution. In [7] it is shown that T_S is also a phase-type random variable. However, its computation via an expression like (4.3.1) is prohibitive for large n and large K, since a matrix of size $O(N \cdot K)$ is involved. We present an alternative recursive algorithm from [9] to obtain the density of T_S in *linear* time and *constant* space.

Defining $Pr[T_S = n] = C(n)$, the quantity $C(n)$ is given by

$$C(n) = \sum_{D(n)} \prod_{k=1}^{N} [I_{\{n_k = 0\}}\, \alpha_{k,K} + I_{\{n_k > 0\}}\, \alpha_k\, T_k^{n_k - 1}\, \beta_k] \tag{4.3.6}$$

where (α_k, T_k) is a discrete PH density of order K with initial vector $(\alpha_k, \alpha_{k,K})$, and $D(n) = \{(n_1, \ldots, n_N) \mid n_j \geq 0,\ 1 \leq j \leq N,\ \sum_1^N n_j = n\}$. The quantity $C(n)$ is given by $\alpha_N\ C_N(n)\ \beta_1$, via the recurrence

$$C_i(j) = T_i\ C_i(j-1) + g_i\ A_i\ C_{i-1}(j) - g_i\ T_i\ A_i\ C_{i-1}(j-1) + A_i\ C_{i-1}(j-1) \tag{4.3.7}$$

for $2 \leq i \leq N$, $1 \leq j \leq n$, where

$$C_1(j) = T_1^{\,j-1} \text{ for } j \geq 1 \tag{4.3.8}$$

$$C_i(0) = g_i\ A_i \cdots g_1 A_1 \text{ for } 1 \leq i \leq N\ , \tag{4.3.9}$$

$A_1 = I$, $A_i = \beta_i\ \alpha_{i-1}$ for $2 \leq i \leq N$, and $g_i = \dfrac{\alpha_{i,K}}{\alpha_i\ \beta_i}$ for $1 \leq i \leq N$.

Since each program P requires at least one step to complete execution, it cannot reach the absorbing state in zero steps, and we set $\alpha_{k,K} = 0$ for $1 \leq k \leq N$. The run time complexity of this computation is $O(N \cdot n)$, where N is the number of PH densities considered.

5. Urn Models of Unified Program Execution

The execution of transformed program $\vec{P}$ is illustrated with the following urn model formulation (see Figures 2a and 2b). Note that $\vec{P}$ is a single program whose execution represents the concurrent execution of programs P_1 through P_N. We work with N balls (where each ball

corresponds to an instance of program P) and K independent urns (where each urn corresponds to a program block) arranged from left to right in increasing order of indices. When $\bar{P}$ begins its execution, all N balls are in urn 1 (i.e., block B_1, or simply block 1), reflecting the fact that all N program components (i.e., P_1 through P_N) are executing block 1 concurrently. After t_1 time units have elapsed, *each* program P_m will require control to be moved to some block j, $j \in \mathbf{R}_1$, this event occurring with probability p_{1j}, $1 \leq m \leq N$. If any two distinct components P_m and P_n of program $\bar{P}$ require control to be moved to blocks i and j, respectively, for $i \neq j$, then it is clear [5] that $\bar{P}$ cannot execute *all* its components concurrently on the next step.

The unified program $\bar{P}$ continues its execution by executing one type of program block at a time, since it utilizes a vector uniprocessor. At the end of each block execution, $\bar{P}$ will have to select a subset of components $\{P_j : j \in S\}$, $S \subset \{1, 2, \ldots, N\}$, such that components in the subset can execute the same block concurrently. This amounts to examining the current pending list in order to select a block that *will* be next executed. While it is apparent that wastage of vectorized execution at each step is locally minimized if $(N - |S|)$ is minimized, it is decidedly a nontrivial issue to obtain an optimal block selection policy. With a view towards determining such a policy, we model the behaviour of $\bar{P}$ under the four different policies introduced in section (3.4).

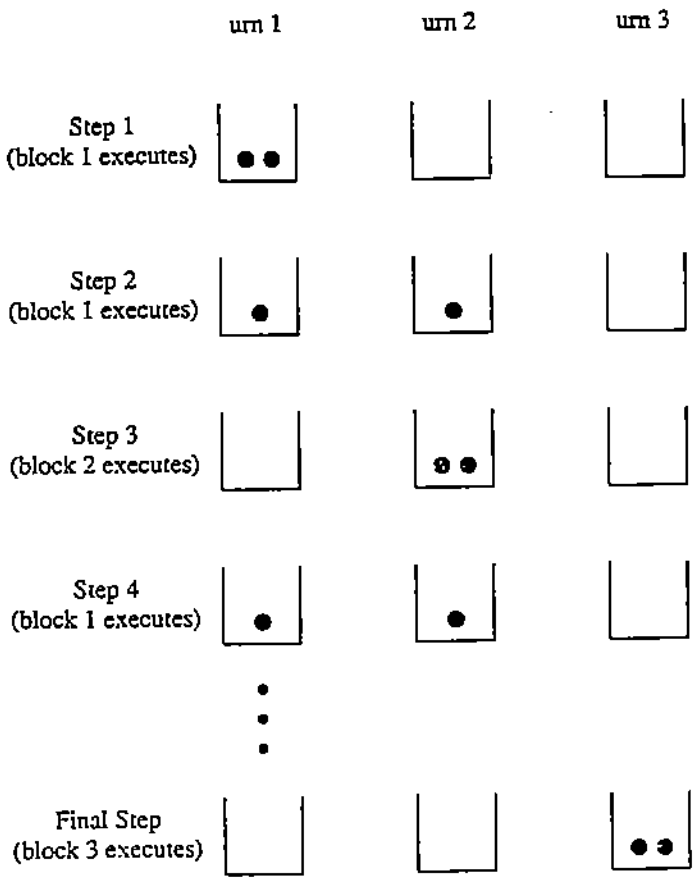


Figure 2b. Urn model for $\bar{P}$, with $K = 3$ and $N = 2$.

5.1 The Selection of Urns for Program Execution

Consider the scene in Figure 2 for $N = 2$ and $K = 3$. Initially, all N balls are in urn 1, and they stay there for t_1 time units after $\vec{P}$ begins its execution. After t_1 time units have elapsed, each of the N balls is tossed into a new (or perhaps the same) urn according to the distribution specified in row 1 of the probability transition matrix $\mathbf{P} = [p_{ij}]$. At each step, the currently executing block is known as the *active block*, and the components of $\vec{P}$ that are currently executing this active block constitute the *active set* (i.e., the others are masked out, and thus inactive) of $\vec{P}$'s components. We can safely assume that it takes zero time to pick up and toss balls into their destination urns. This is actually the overhead associated with concurrent execution of P_1 through P_N and is discussed in [5]. Including this overhead in the block execution time justifies our assumption.

At the end of execution of the currently active block, those balls occupying the urn representing the active block are tossed, and these fall into some destination urn. It is now up to the *block selection policy* to decide which of the currently nonempty urns (representing instances of P waiting to execute some block) is to act as the urn representing the next active block. This urn selection and ball tossing procedure is repeated until all N balls are in block K where they are forced to remain for t_K time units. We are assuming that all components of $\vec{P}$ terminate. At the end of this period, program $\vec{P}$ is said to have terminated. The four block selection policies we choose to consider (see [5]) have already been described in section (3.4).

Once the block selection policy is fixed, the balls are seen to move between urns in a random fashion. Clearly, $\vec{P}$ terminates its execution when all N balls are in urn K. The running time of $\vec{P}$ is given by the amount of time it takes for all N balls to move from urn 1 to urn K. The problem thus formulated is a sequential urn occupancy problem, made complicated (as far as explicit formulas are concerned) by the asymmetry of transition probabilities in $\mathbf{P}$, and the interference (between paths taken by the individual balls) brought about by block selection policy.

Remark

The algorithm outlined above allows for the possibility in which each of the N components of $\vec{P}$ executes block K at a different time. Since all N components of $\vec{P}$ are required to execute block K in order to terminate, we modify the algorithm slightly to shorten the expected length of $\vec{P}$'s execution time. In executing the block selection policy at each step, we exclude the possibility of selecting block K for execution until *all* N balls are in urn K. In this way we force program components that would like to execute block K to wait until all components can execute block K concurrently. This saves up to a maximum of $(N - 1)$ block execution steps.

■

5.2 Urn Model for the Complete First Policy

In order to describe the analytic model clearly and briefly, let us work with the *complete first policy* for block selection. At the very outset, let us establish that our goal is to determine the execution time of transformed program $\tilde{P}$, i.e., the number of steps (or sets of tosses) required for all N balls to move from urn 1 to urn K. Once this random variable can be defined (i.e., its distribution obtained), we can use ideas identical to those given in section 4.1 to obtain an estimate of $\tilde{P}$'s execution time distribution, and the exact mean and variance of $\tilde{P}$'s execution time.

Since the components $P_1, P_2, \ldots, P_N$ of $\tilde{P}$ are N independent, but identically operating Markov chains $\{X_{1,k}; k \geq 0\}, \{X_{2,k}; k \geq 0\}, \ldots, \{X_{N,k}; k \geq 0\}$, we can ignore the specific *identities* of the balls in the different urns and merely keep track of the *number* of balls in each urn at each step k, $k \geq 0$. That is, since $X_{1,k}, X_{2,k}, \ldots, X_{N,k}$ are *exchangeable* random variables [10], using $Y_{j,k}$ to denote the number of balls in urn j at step k, we have

$$Y_{j,k} = \sum_{i=1}^{N} I_{\{X_{i,k} = j\}}, \qquad 1 \leq j \leq K \tag{5.2.1}$$

for any k, where $I_{\{X\}}$ is the indicator function for the event $\{X\}$.

Let $\mathbf{Y}_k = (Y_{1,k}, \ldots, Y_{K,k})$ be a vector whose j^{th} entry, $Y_{j,k}$, gives the number of balls in urn j, $1 \leq j \leq K$, at step k, $k \geq 0$. Observe that the number of blocks of $\tilde{P}$ that execute at step k, when the complete first policy is used, is given by $Y_{m,k}$ where $m = \min \{j \mid 1 \leq j \leq K,\ Y_{j,k} > 0\}$. That is, we use m to denote the generic index of the active block at any step k. The size of the active set at step k is $Y_{m,k}$, namely, the number of balls in the selected urn, or the number of program components of $\tilde{P}$ executing block m. The configuration $\mathbf{Y}_{k+1}$ of balls in the different urns at step $(k + 1)$ can be obtained from the configuration $\mathbf{Y}_k$ at time k. Note that the only allowable change between the two configurations is that the $Y_{m,k}$ balls in urn m (the least index nonempty urn at step k) are tossed into a set of urns whose indices are given by the reachability set $\mathbf{R}_m$.

Let $Z_{i,j}$ be a Bernoulli random variable that takes on the value 1 with probability $p_{i,j}$ and the value 0 with probability $(1 - p_{i,j})$. The configuration $\mathbf{Y}_{k+1} = (Y_{1,k+1}, \ldots, Y_{K,k+1})$ is obtained from $\mathbf{Y}_k$ as follows. Given that block m is the block selected by the complete first policy at step k, let $\mathbf{R}_m = \{i, j\}$ be the reachability set of the selected urn if $|\mathbf{R}_m| = 2$, and $\mathbf{R}_m = \{i\}$ if $|\mathbf{R}_m| = 1$. Since the contents of those urns with indices not in $\mathbf{R}_m$ cannot change, we have that

$$Y_{r,k+1} = Y_{r,k} \qquad r \notin \mathbf{R}_m,\ 1 \leq r \leq K \tag{5.2.2}$$

In order to determine how the contents of the urns with indices in $\mathbf{R}_m$ change in the transition from $\mathbf{Y}_k$ to $\mathbf{Y}_{k+1}$, we must consider the random variable which is a sum of $Y_{m,k}$ Bernoulli

random variables,

$$Z = \sum_{1}^{Y_{m,k}} Z_{m,i} \tag{5.2.3}$$

satisfying $Y_{i,k+1} = Z$ if $|\mathbf{R}_m| = 1$, and $Y_{i,k+1} \leq Z$ if $|\mathbf{R}_m| = 2$. That is,

$$Y_{r,k+1} = \begin{cases} Y_{i,k} + Z & r = i \\ Y_{j,k} + Y_{m,k} - Z & r = j \end{cases} \tag{5.2.4}$$

Since $\mathbf{Y}_{k+1}$ is obtainable from $\mathbf{Y}_k$ and an independent random variable, and since transition probabilities do not depend on time, the sequence $\{\mathbf{Y}_k ; k \geq 0\}$ is clearly a time-homogeneous Markov chain. The probability of making a transition from a configuration $\mathbf{Y}_k = (i_1, \ldots, i_K)$ with active block m, to a configuration $\mathbf{Y}_{k+1} = (j_1, \ldots, j_K)$ is given by

$$Pr[(j_1, \ldots, j_K) | (i_1, \ldots, i_K)] = \begin{cases} 0 & j_m > i_m \\ 0 & j_r < i_r, r \neq m \\ \left[\dfrac{i_m!}{j_m!} \right] (p_{m,m})^{j_m} \prod_{r=1, r \neq m}^{K} \dfrac{p_{m,r}^{(j_r - i_r)}}{(j_r - i_r)!} & \text{otherwise} \end{cases} \tag{5.2.5}$$

An application of (5.2.5) to all states $(j_1, j_2, \ldots, j_K)$ such that $\left[\sum_{r=1}^{K} j_r \right] = n$ yields a probability transition matrix denoted by $\mathbf{Q}_{CF}$ (where the subscript denotes that a complete first policy is used) for the transformed program $\bar{P}$. The ideas outlined in Section 4 can now be applied to obtain the exact mean and variance of execution time, and an estimate of the distribution of execution time $T_{\bar{P}}$ for $\bar{P}$.

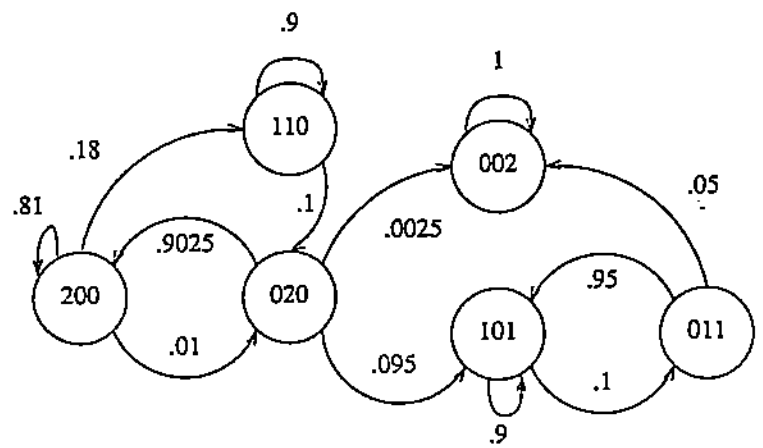


Figure 2c. Non-deterministic flow graph for $\bar{P}$.

A Numerical Example

In the case of the example given in Figure 2c, for $N = 2$ and $K = 3$, this method gives the set of six states (i.e, distinct set of urn configurations) (2,0,0), (1,1,0), (1,0,1), (0,2,0), (0,1,1), and (0,0,2). In Figure 2c is displayed the Markov flow graph obtained by using (5.2.5) and the complete first block selection policy. For our example, since we consider only $N = 2$, we take $\alpha = 0.98$. In general, our empirical results indicate that α is typically in the interval (0.01,0.1) for moderate values of N and K (say $N > 10$, and $K > 10$). We discuss this further in the next section. In using the stochastic matrix Q_{CF} and the ideas outlined in section 4, we obtain the average execution time of $\vec{P}$ as 6197.16 time units, and the variance of execution time of $\vec{P}$ as 828802.86 time units. Thus the average speedup obtained by utilizing $\vec{P}$ instead of N serial executions of P is given by

$$S_{\vec{P}} = \frac{N \cdot E[T_P]}{E[T_{\vec{P}}]} \tag{5.2.6}$$

$$= \frac{2 \times 1866.99}{6197.16} = 0.6025 < 1$$

which tells us that it is *not* beneficial to use $\vec{P}$ in place of P for $N = 2$, the given value of α, and given transition probability matrix P for program P. This is only to be expected for small N and large α since a speedup of greater than unity can be attained only when α is small enough to offset any loss in waste-factor effect [5], i.e., the time wasted during $\vec{P}$'s execution when components of $\vec{P}$ are masked out of the current execution step. However, we will show that when α is sufficiently small, and N is sufficiently large, a tremendous degree of speedup may be obtained by resorting to the execution of $\vec{P}$.

5.3 Other Block Selection Policies

In the case of a block selection policy other than complete first, only a small modification of (5.2.5) is required in order to obtain the corresponding transition probability matrix. For the *move forward policy*, let m be defined by $m = \max \{j \mid 1 \le j \le K-1,\ Y_{j,k} > 0\}$. If $Y_{j,k} = 0$ for $1 \le j \le K-1$, then all balls must finally be in urn K, and we set $m = K$. Then, using Q_{MF} to denote the move forward version of the transition probability matrix, Q_{MF} is computed exactly as in (5.2.5), with this new definition of active block (or urn) m.

In the case of the *majority rule policy*, we define m to be the index of the urn containing the largest number of balls. If there is more than one urn with the same (largest) number of balls, m is set to be the least index from among these urn indices. That is, define the set M of urns with the largest number of balls at step k as

$$\mathbf{M} = \{\, j \mid Y_{i,k} \le Y_{j,k},\ 1 \le i \le K-1,\ i \ne j \,\}. \tag{5.3.1}$$

If $|\mathbf{M}| = 0$, then all N balls are in urn K and we set $m = K$. If $|\mathbf{M}| = 1$, then there is only one candidate j, $1 \le j \le K-1$, with $\mathbf{M} = \{j\}$, for the active set at time k. In this case, the urn to be selected *must* be urn j and we set $m = j$. If $|\mathbf{M}| > 1$, we take m to be the urn whose index is the least from among all those urns whose indices are listed in M. Once m is obtained, the probability transition matrix $\mathbf{Q}_{MJ}$ corresponding to the majority rule policy is computed exactly as in (5.2.5).

Finally, in case of the random choice policy, we define a set M of candidate urns (i.e., nonempty urns) at step k,

$$\mathbf{M} = \{\, j \mid Y_{j,k} > 0\ ,\ 1 \le j \le K-1 \}. \tag{5.3.2}$$

If $|\mathbf{M}| = 0$, then all N balls are in urn K and we set $m = K$. If $|\mathbf{M}| = 1$, then there is only one candidate and we set m to be the index of this urn. If $|\mathbf{M}| > 1$, then for each $j \in \mathbf{M}$, we set $m = j$ and compute the probability, say δ_j, given in (5.2.5) with urn j as selected urn. Since the random choice policy essentially chooses one urn at random from among the candidate urns, the transition probability of going from configuration $\mathbf{Y}_k$ to $\mathbf{Y}_{k+1}$ is computed via conditioning as

$$Pr[(j_1, \ldots, j_K) | (i_1, \ldots, i_K)] = \frac{\sum_{j \in M} \delta_j}{|M|} \tag{5.3.3}$$

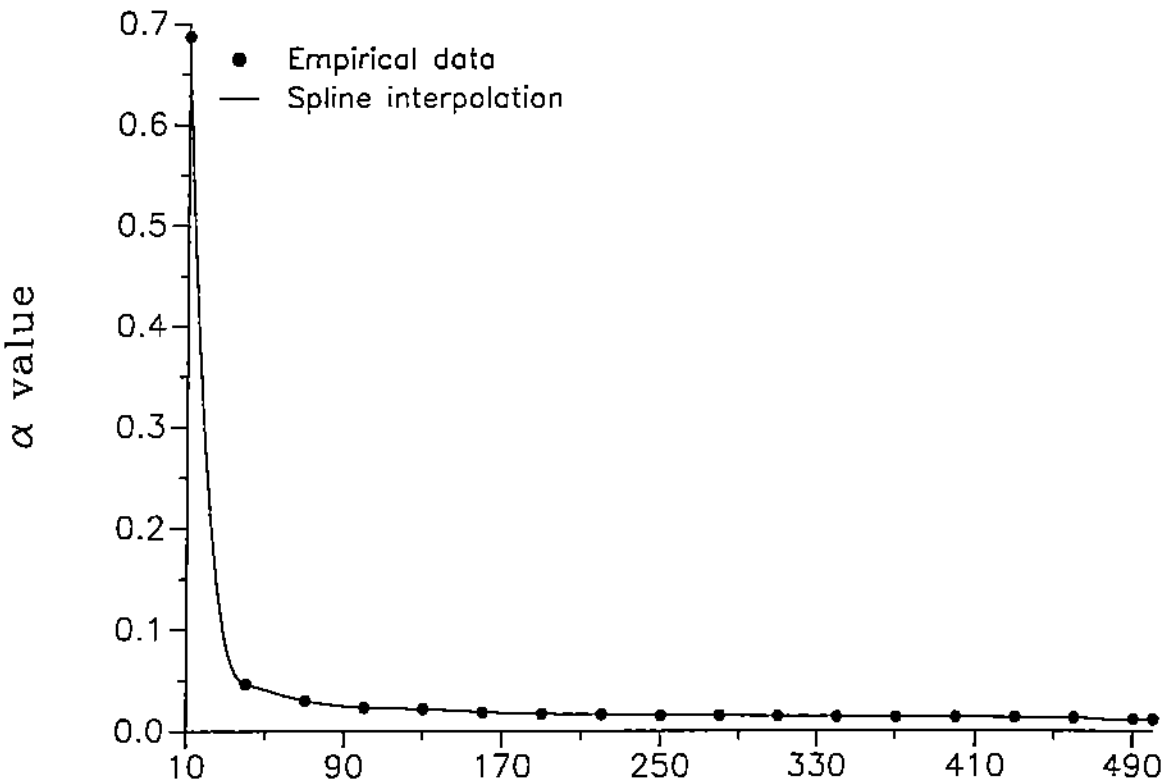
provided $|\mathbf{M}| > 0$. Here, the unconditional probability of selecting any urn in M is given by $1/|\mathbf{M}|$. When $|\mathbf{M}| = 0$, the probability corresponding to state transition $(0, \ldots, K) \rightarrow (0, \ldots, K)$ is 1, and the probability of going to any other state from state $(0, \ldots, K)$ is 0, since the latter state is an absorbing state.

6. Computational Results

In this section we present some computational results to demonstrate the degree of speedup that can be obtained via induced vectorization, for a simple class of program structures, over a wide range of parameter values. Since the unified program's speedup is a function of block speedup, our models require estimates of block speedup in order to evaluate program speedup.

6.1 Empirical Estimates of block speedup

In Figure 3 is shown a set of empirically obtained α values for the Alliant FX/8, for values of N ranging from 10 to 500. We use a spline interpolation procedure to give us values of α as a function of N for arbitrary values of N in this range, for use in our models. The results indicate that the speedup factor is exponential over a wide range of values of N, tending to a constant for large N.



Empirical α vs. N and spline interpolation

Fig 3.

6.2 Specifying the program structure

Since we use a stochastic matrix to represent an arbitrary program P, the form of the matrix is necessarily a parameter of the model. This means that each $K \times K$ matrix will contribute up to $2(K-1)$ parameters, assuming a maximum of two nonzero entries in the first $(K-1)$ rows. Incorporating a model description with this large a parameter set is impractical. Besides, the problem of characterizing absorption times for $K \times K$ matrices given the incidence matrix and/or the values of nonzero entries is an open problem.

In order to make our results meaningful and the size of the parameter set practicable, we restrict our attention to programs whose incidence matrices take the banded form shown in Figure 4. Additionally, by involving only a single parameter β in the matrix, we have more control over the parameter set governing model behaviour. The programs P that we consider are those programs which, when executing a given block j, tend to move forward to block $(j+1)$ with probability β after block j's execution is complete, and failing this, stay to execute block j another time before another attempt to move to the next block.

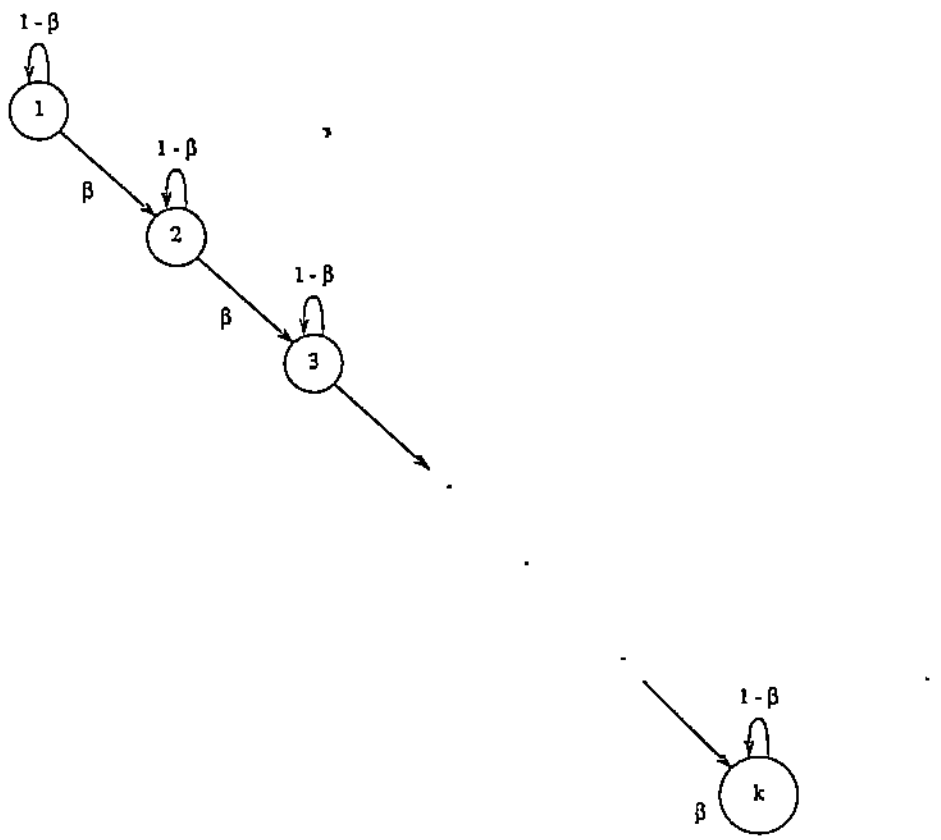


Figure 4. Simple program structure.

6.3 Analytic Results : Testing the Models

To begin with, consider the analytic urn models presented in section 5. In general, we obtain a state space of size $\begin{bmatrix} N+K-1 \\ K-1 \end{bmatrix}$ for a given N and K, and this makes the construction of models with realistic values of N and K (say $N = 100$, $K = 100$) impossible. Nevertheless, suppose that we scale down the function shown in Figure 3 and assume, for a moment, that such a trend is true for values of N ranging from 1 to 10 (see Table 1). For small K, we should expect that our models would give us an idea of the qualitative behaviour of $S_{\vec{P}}$ for fixed K and increasing N.

N	1	2	3	4	5	6	7	8	9	10
α	1.0	.60	.40	.20	.10	.09	.08	.07	.06	.05

Table 1. α values used in analytic model.

Consider a test situation, with $K = 4$, $\beta = 0.5$, and N ranging from 1 to 10. In Figures 5a through 5d, we plot $S_{\vec{P}}$ vs. N for each of the four block selection policies. In each case, we see that $S_{\vec{P}}$ is maximum for the complete first policy, and least for the move forward policy. A little reflection will convince the reader that for *the given program structure*, while the complete first policy tends to wait for, and push straggling components of $\vec{P}$ along towards completion, the move forward policy gives preference to the components of $\vec{P}$ that race towards completion. The former is more cooperative and thus encourages block synchronization, while the latter is more individualistic and sacrifices such synchronization in favour of faster program completions. The majority rule and random choice policy are less easily understood. It appears that while attempting to maximize block concurrency locally, the majority rule policy gives up more globally than the policy of choosing blocks to be executed randomly. It is clear that for different program structures, these relative behaviours will change, but in a manner that is not easily predictable.

6.4 Simulation Results

In order to determine the behaviour of the four policies, as well as the behaviour of $S_{\vec{P}}$ for large values of N and K, we resort to simulations of the urn models described in section 4. In Figures 5a through 5d are displayed the results of these simulations alongside the corresponding analytic curves. The simulation graphs are based on 99% confidence intervals for estimated $S_{\vec{P}}$, with a relative precision of $\gamma = 0.1$. That is, the ratio of the width of the confidence interval at a

point, and the absolute value of the estimate at that point is bounded above by γ. The simulation models were tested extensively with the aid of the analytic models for various values of β, N, and K.

In Figures 6a through 6e are displayed simulated curves for the four block selection policies, showing the variation of $S_{\tilde{P}}$ with $K = 100$ (fixed), and N ranging from 1 to 500. Observe that speedup tends to decrease uniformly as β decreases (from 0.9 to 0.1) for all policies. Since smaller values of β yield programs that take a longer time to execute, this indicates that with increasing N, there tends to be increasing conflict instead of increasing synchronization, and this causes $\tilde{P}$ to take a longer time to execute, consequently causing $S_{\tilde{P}}$ to decrease. In each of these simulations, the empirical data given in Figure 3 is used for the block speedup α values in $\tilde{P}$. The oscillation of $S_{\tilde{P}}$ for N close to 500 is an artifact, due to a small oscillation at the right end of the spline function. This in turn is caused by a small rise and fall in the value of α in that region, and suggests that $S_{\tilde{P}}$ may be quite sensitive to α. While the 99 % confidence intervals are not explicitly displayed, they are readily reconstructed from the estimated values of $S_{\tilde{P}}$ on the graphs and the fact that in each case the relative precision is $\gamma = 0.1$. Not surprisingly, the simulation graphs exhibit the same qualitative behaviour as the analytic curves (where the α values were scaled down), even so far as to keep the speedup characteristics of the four different policies in the same order.

In a final experiment, we keep N fixed at 100 data sets and vary K from 2 to 100, in order to observe the behaviour of $S_{\tilde{P}}$ for the four policies as β decreases from 0.9 to 0.1. These results are displayed in Figures 7a through 7e. As suggested by the previous sets of graphs, once again we find that the complete first policy is the best, and the move forward policy is the worst, for *the kinds of programs* P given by Figure 3. Also as to be expected, $S_{\tilde{P}}$ tends to decrease with decreasing β, suggesting that longer running programs with an increasing number of blocks will exhibit less synchrozization. For a fixed β, except for the complete fixed policy which tends to maintain a constant value of $S_{\tilde{P}}$, the other policies yield decreasing speedup with increasing K. The rate of this decrease is a function of N, and this rate will grow smaller with increasing N.

7. Future Work

From our initial studies, the technique of program unification through induced vectorization appears to us to be a research direction that is worthy of some detailed investigation. Current work includes a mechanized transformation procedure that takes a program P as input and transforms it into a unified program $\tilde{P}$, for a given number of data sets. In addition, benchmarking of block speedup values for α is being carried out on various machines such as the Cray X/MP, the Cyber 205, Alliant FX/8, and the SCS40. These benchmarks will prove useful in

demonstrating speedup both analytically as well as empirically in future work.

Besides utilizing the unification idea on vector uniprocessors,it should be clear that the technique is equally applicable to vector processors and SIMD machines, with much greater speedup to be expected as the number of processors increases. We plan to investigate these ideas further, in order to establish guidelines for block selection policies in terms of given programs P. The last problem is a nontrivial one, as is the problem of analyzing models with large N and K. In view of these computational difficulties, future work will involve analytic approximations for large models, and heuristic solutions to the block selection problem. The most difficult problem in the set of problems we have encountered thus far is that of constructing a stochastic matrix **P** that is truly representative, at least in an average sense, of the behaviour of a arbitrary program P over a fixed, but non degenerate domain of data sets.

References

[1] J.R. Allen and K. Kennedy, "PFC: A Program to Convert Fortran to Parallel Form," Technical Report MASC TR 82-6, Department of Mathematical Sciences, Rice University, Houston, Texas, 1982.

[2] *CFT77* Reference Manual, Cray Research Inc., MN 55120, Sept. 1986.

[3] *FX/FORTRAN*, Programmer's Handbook, Alliant Computer Systems Corporation, MA 01460, March 1987.

[4] E. Galiano, "Vectorization Over Multiple Data Sets," *Design Project Report*, School of ICS, Georgia Institute of Technology, Atlanta, GA 30332, June 1987. Software Engineering Research Center, Georgia Institute of Technology, Atlanta, GA 30332.

[5] A.P. Mathur and Eugene Galiano, "Inducing Vectorization: A Formal Analysis," *Technical Report*, SERC-TR-6-P, Software Engineering Research Center, Dept. of Computer Science, Purdue University, also *to appear in Proceedings of the Third International Conference in Supercomputing*, Boston, May 15-20, 1988.

[6] A.P. Mathur, Eugene Galiano, Walter Ligon III and Terry Greenlaw, "Concurrent Execution Over Multiple data sets On Vector Processors," Technical Report, GIT-SERC-87/12, Software Engineering Research Center, Georgia Institute of Technology, Atlanta, GA 30332.

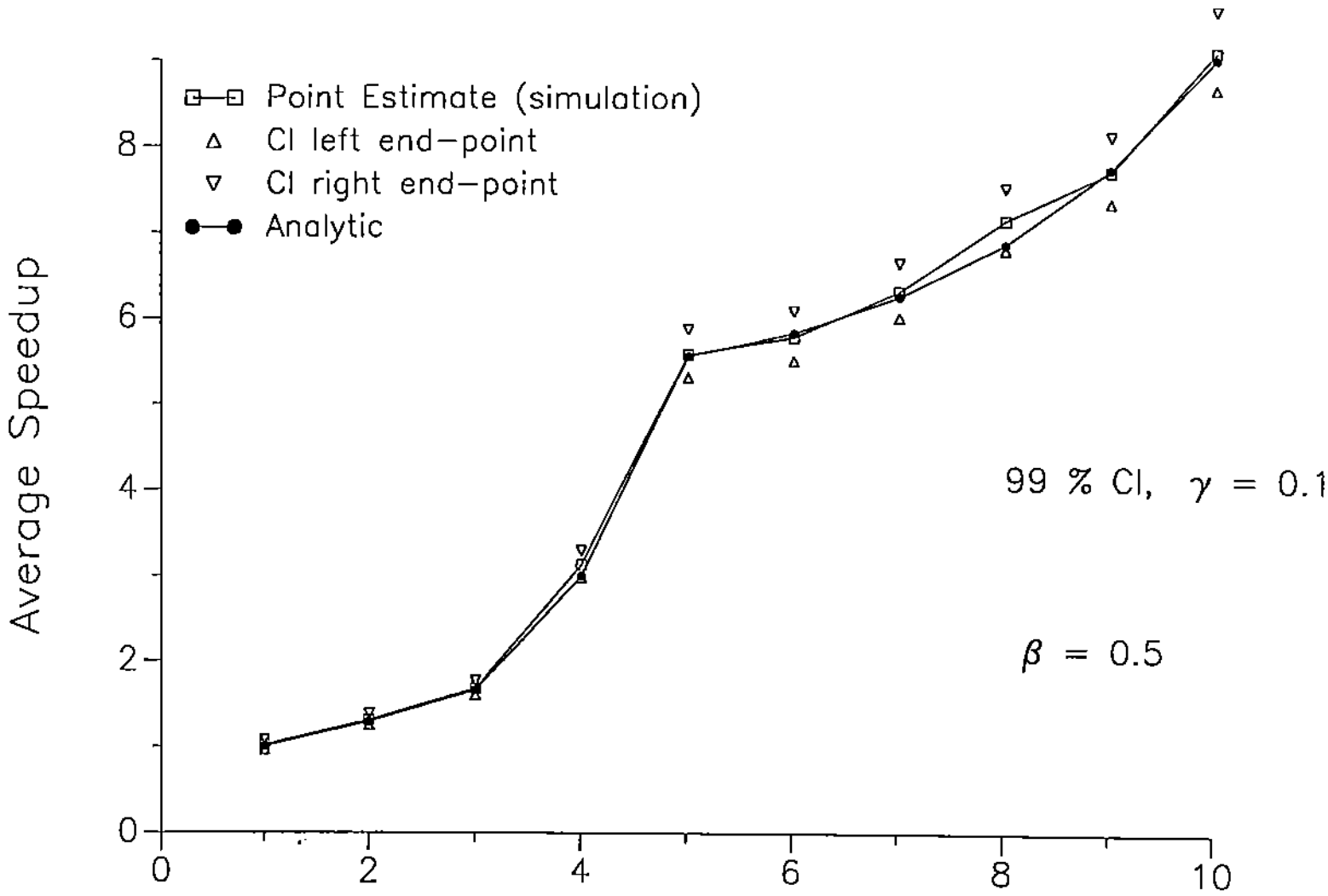
[7] M. F. Neuts, *Matrix-Geometric Solutions in Stochastic Models: An Algorithmic Approach*, The Johns Hopkins University Press, 1981.

[8] D. A. Padua and M.J. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Comm. of ACM*, vol. 12, no. 29, pp1184-1201, Dec. 1986.

[9] V. Rego, "Some Efficient Computational Algorithms Related to Phase Models," Purdue CSD-TR 727, December 1987.

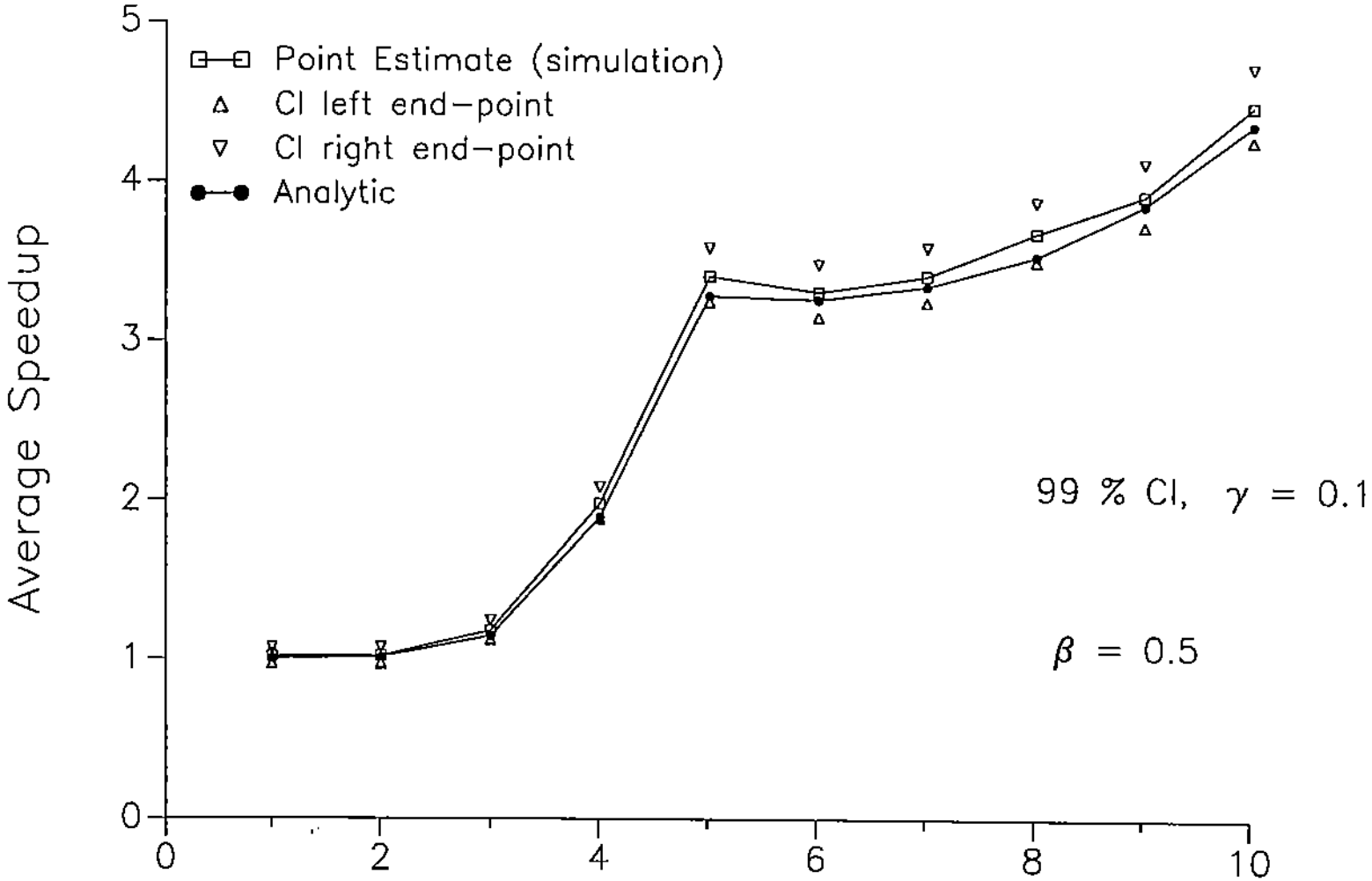
[10] S. M. Ross, *Stochastic Processes*, John Wiley & Sons, 1983.

[11] R. Triolet, "Interprocedural Analysis Based Restructuring of Programs", *Proc. of the International Workshop on Parallel Algorithms and Architectures*, Center National de Recontres mathematiques, Luminy, France, 14-18 April,1986, pp203-217.



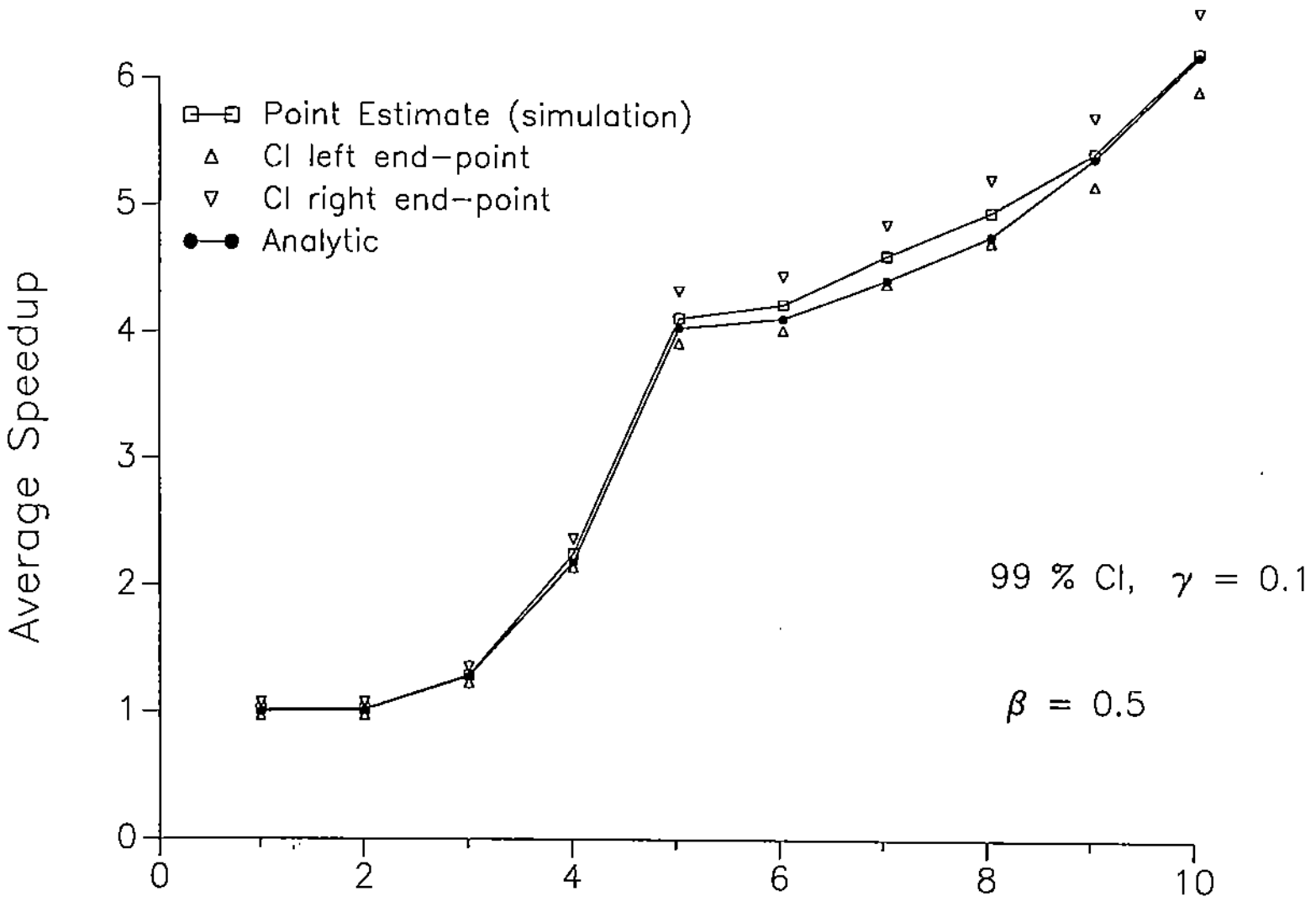
Analytic vs. Simulation result (Complete First)

Fig. 5a



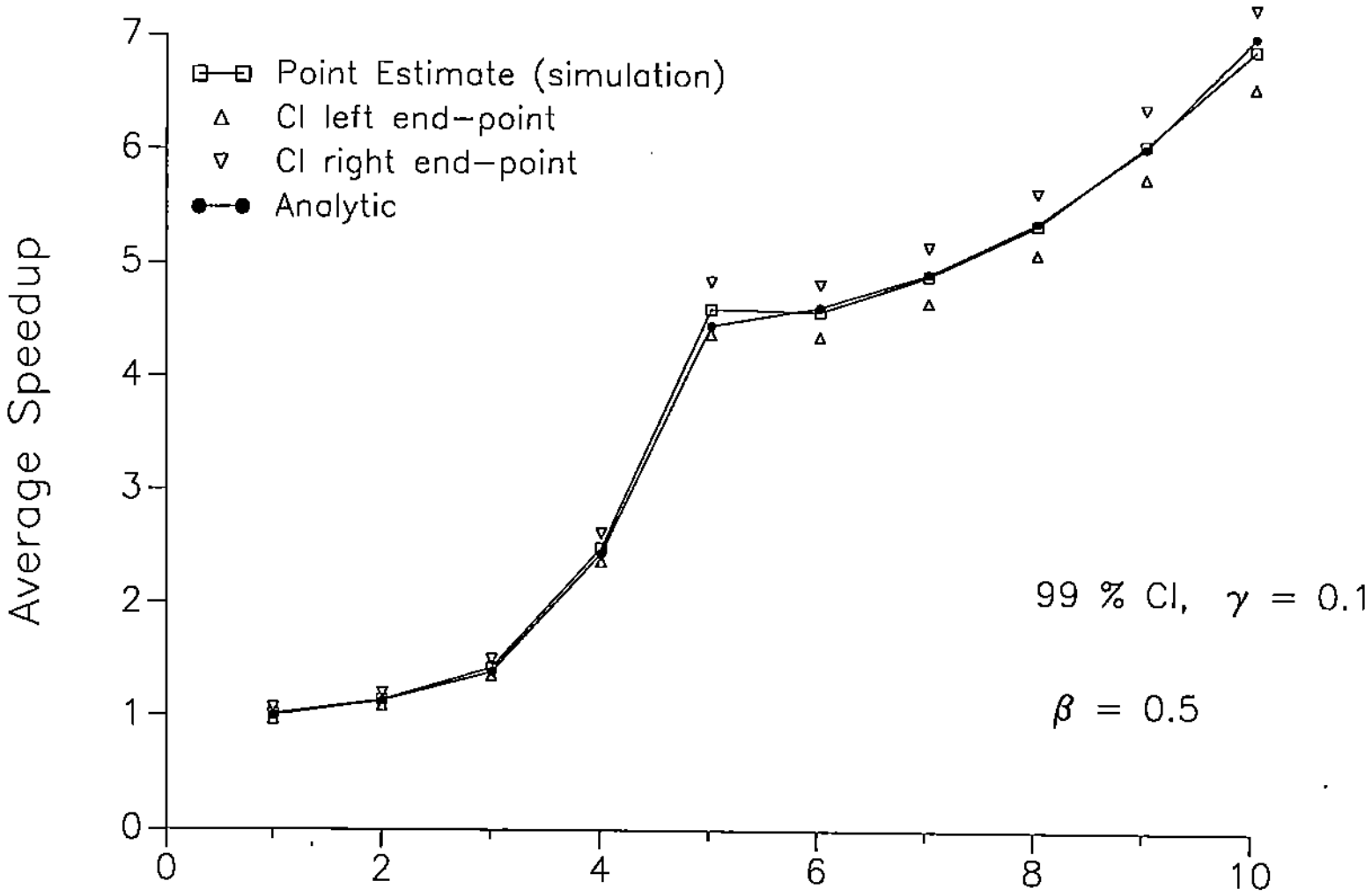
Analytic vs. Simulation result (Move Forward)

Fig. 5b



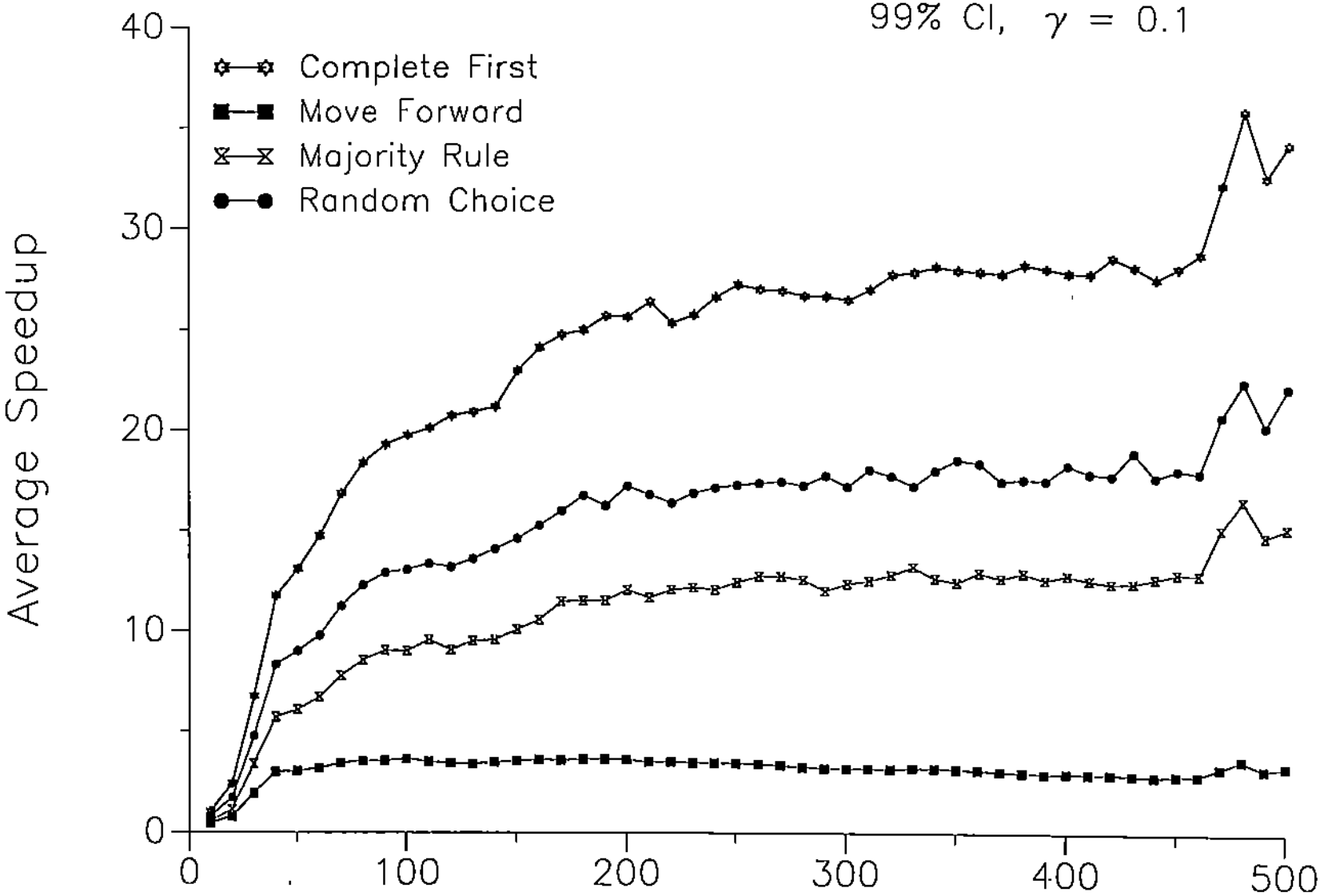
Analytic vs. Simulation result (Majority Rule)

Fig. 5c



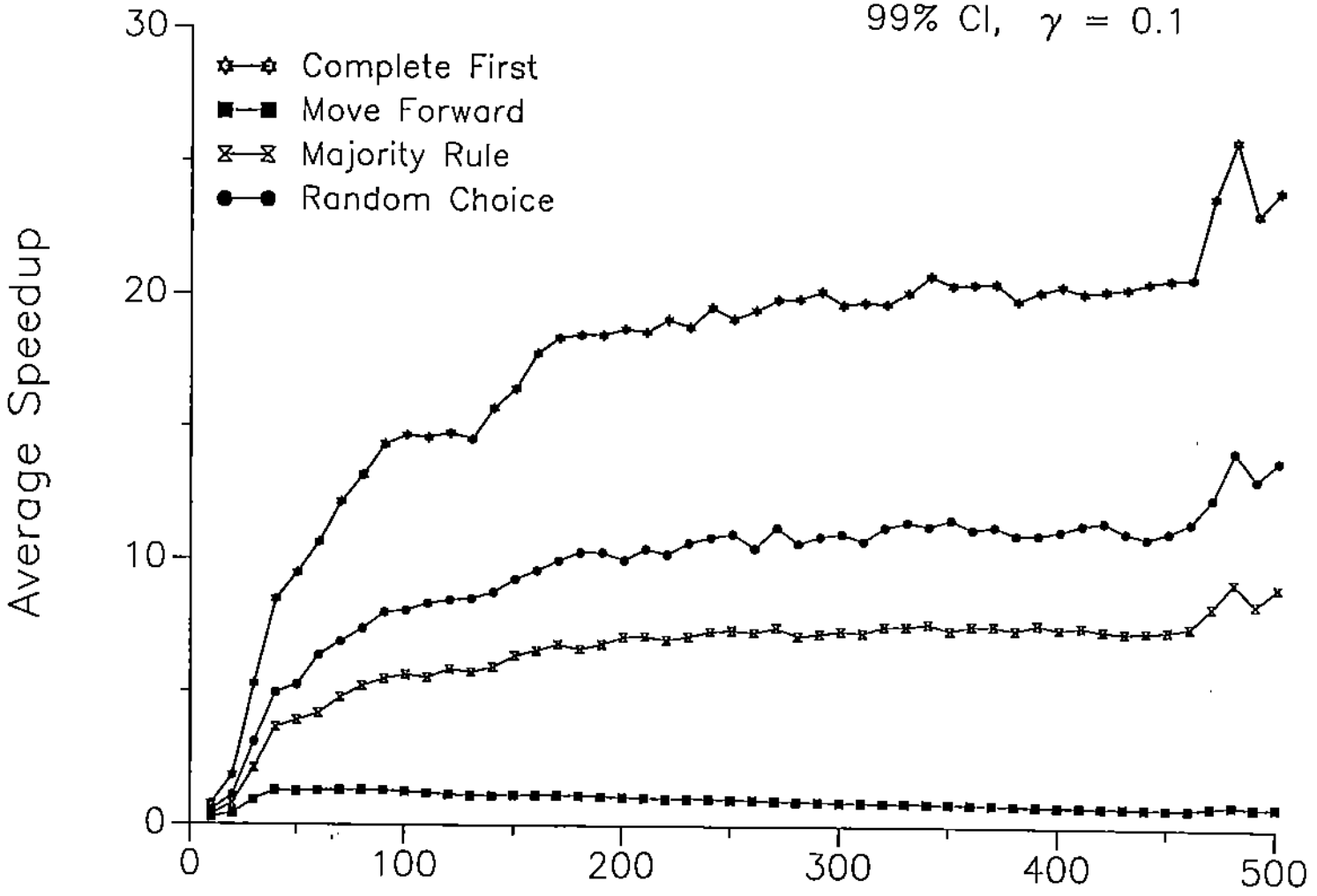
Analytic vs. Simulation result (Random Choice)

Fig. 5d



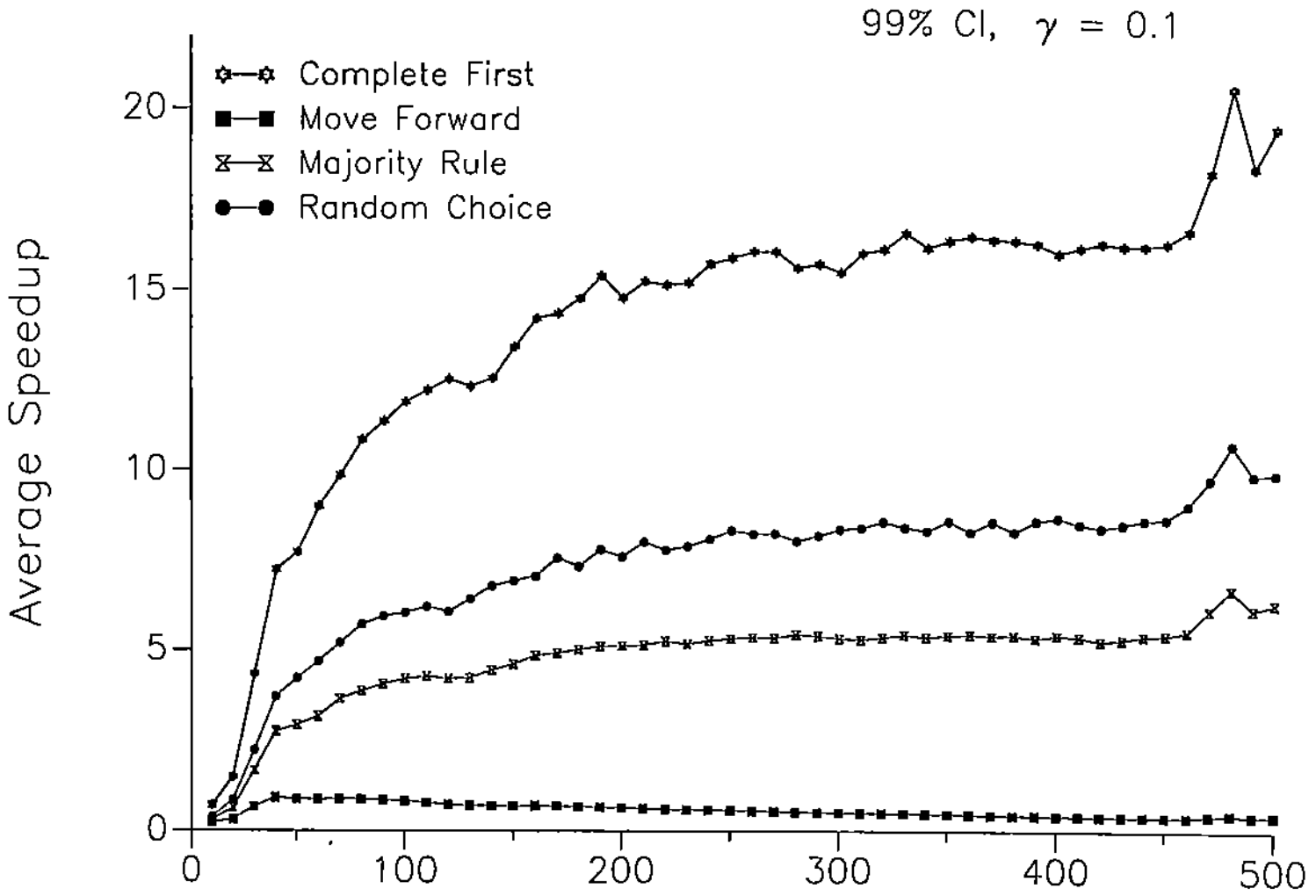
Average Speedup for policies with $\beta = 0.9$

Fig 6a.



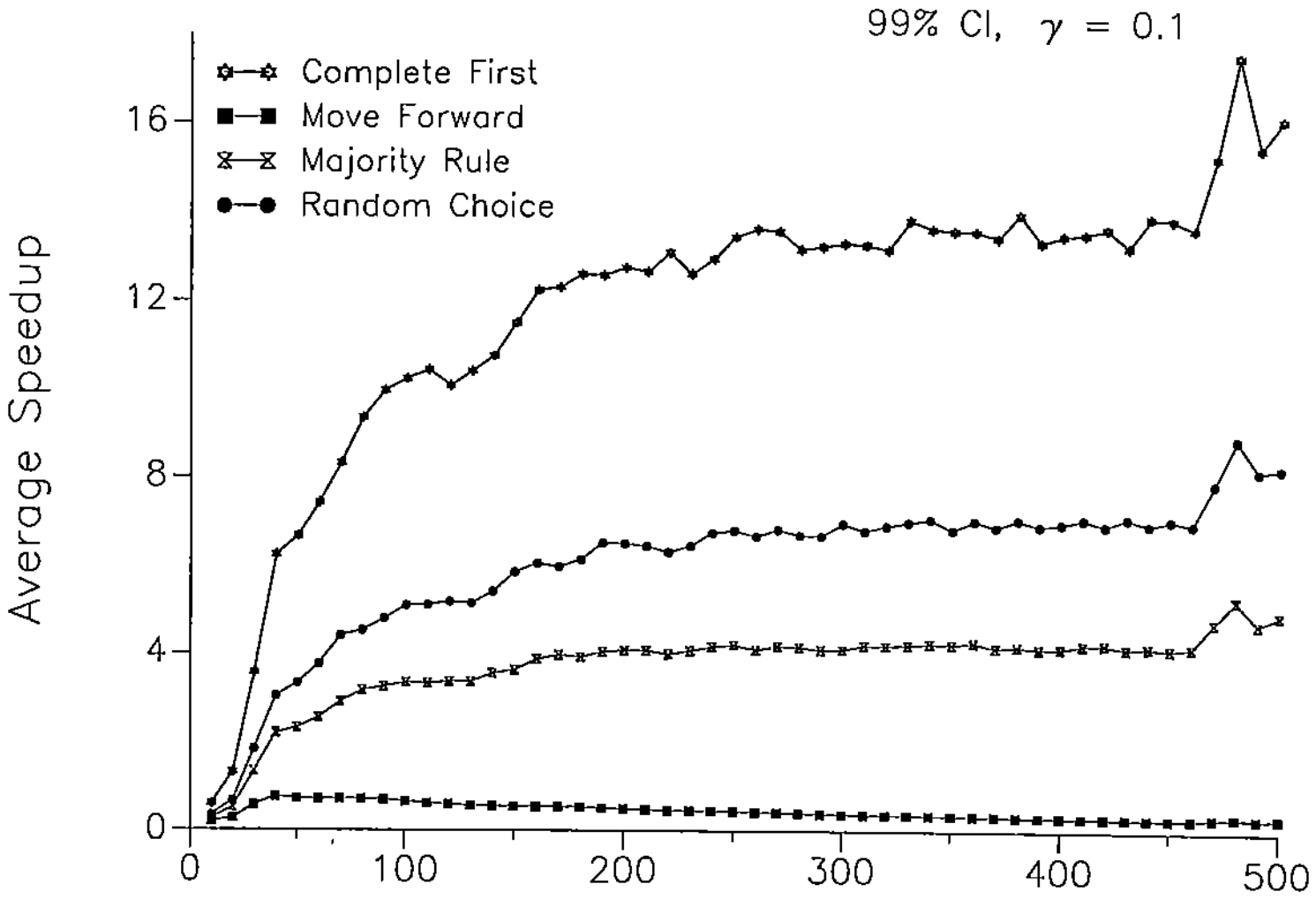
Average Speedup for policies with $\beta = 0.7$

Fig 6b.



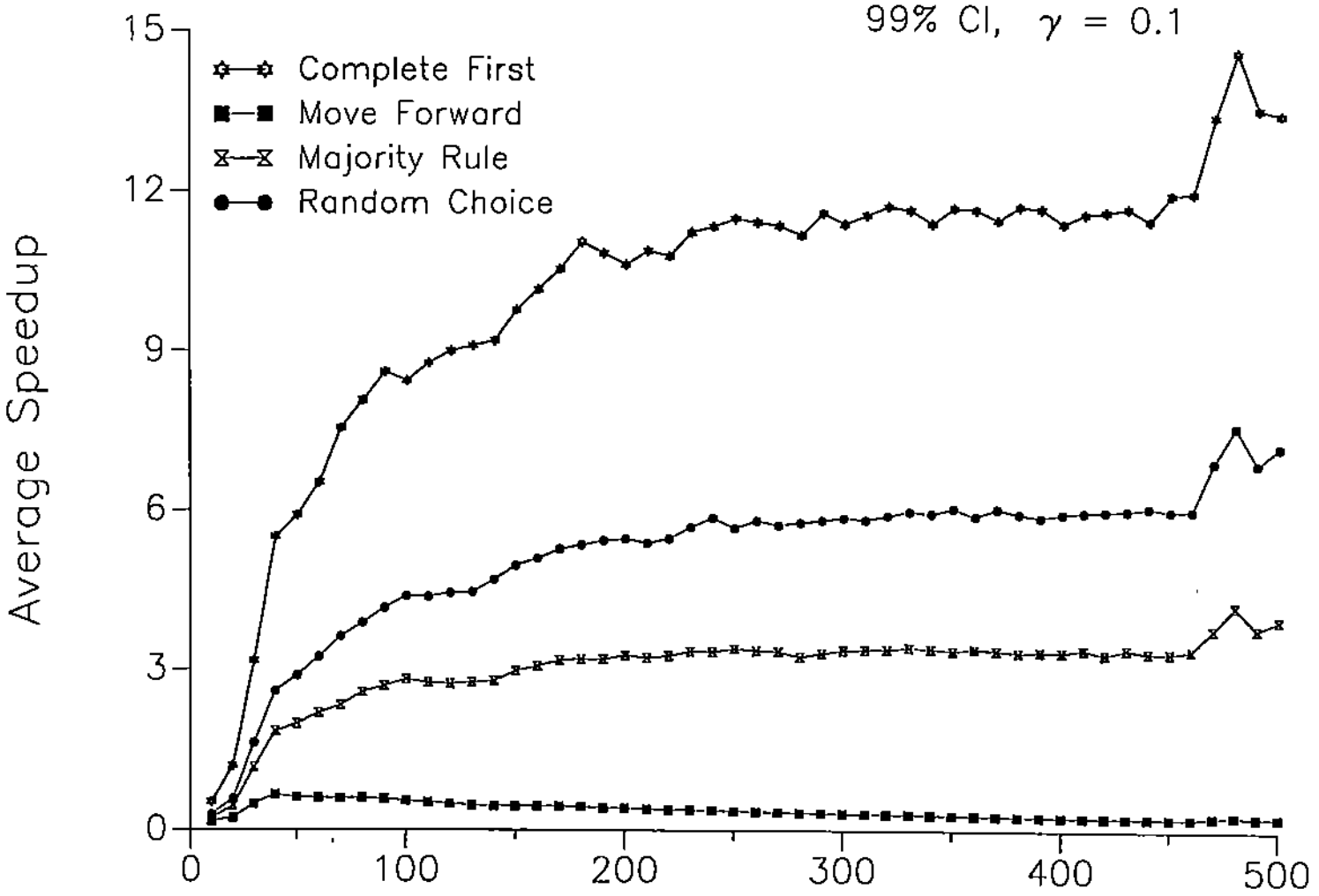
Average Speedup for policies with $\beta = 0.5$

Fig 6c.



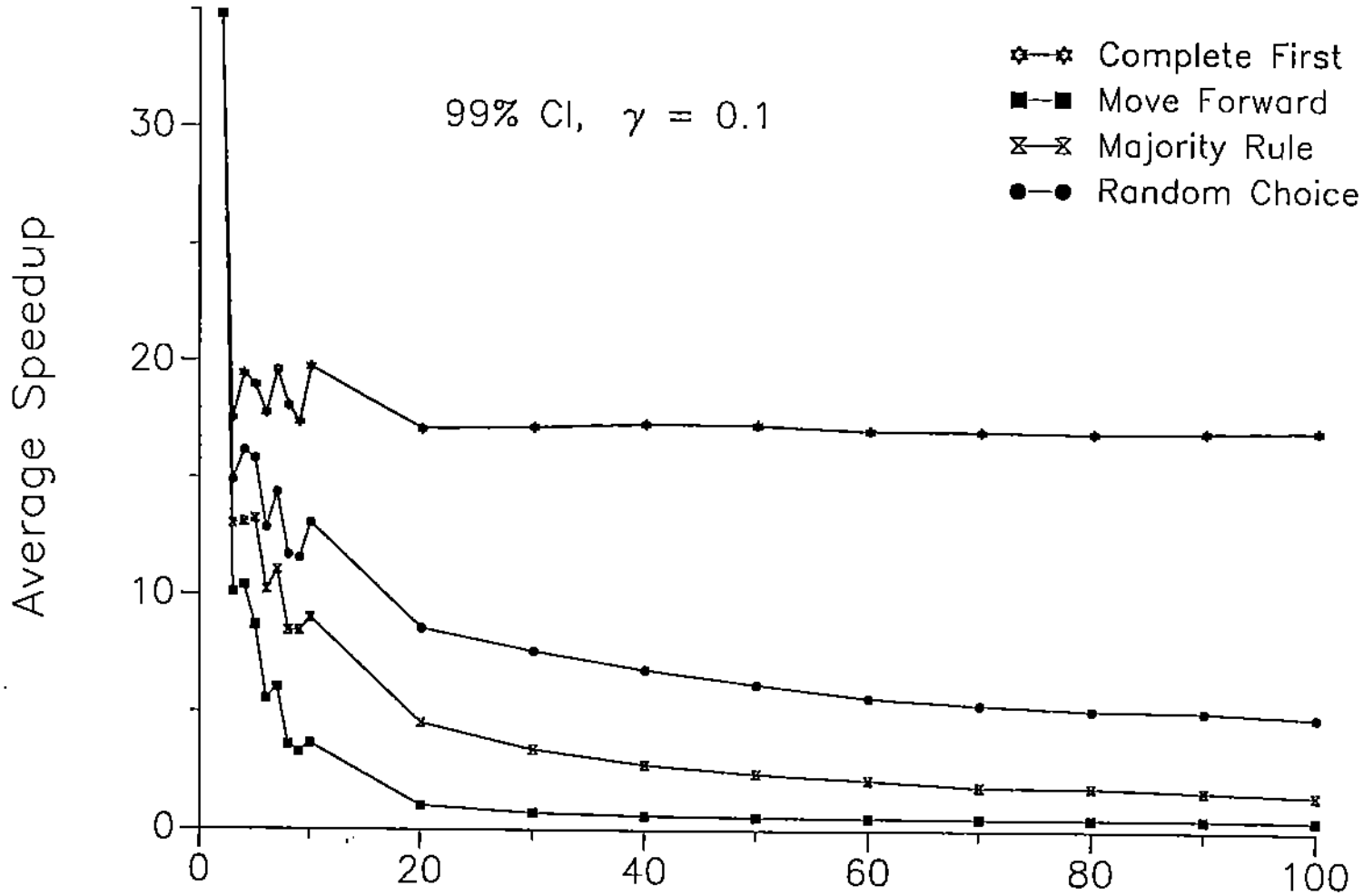
Average Speedup for policies with $\beta = 0.3$

Fig 6d.



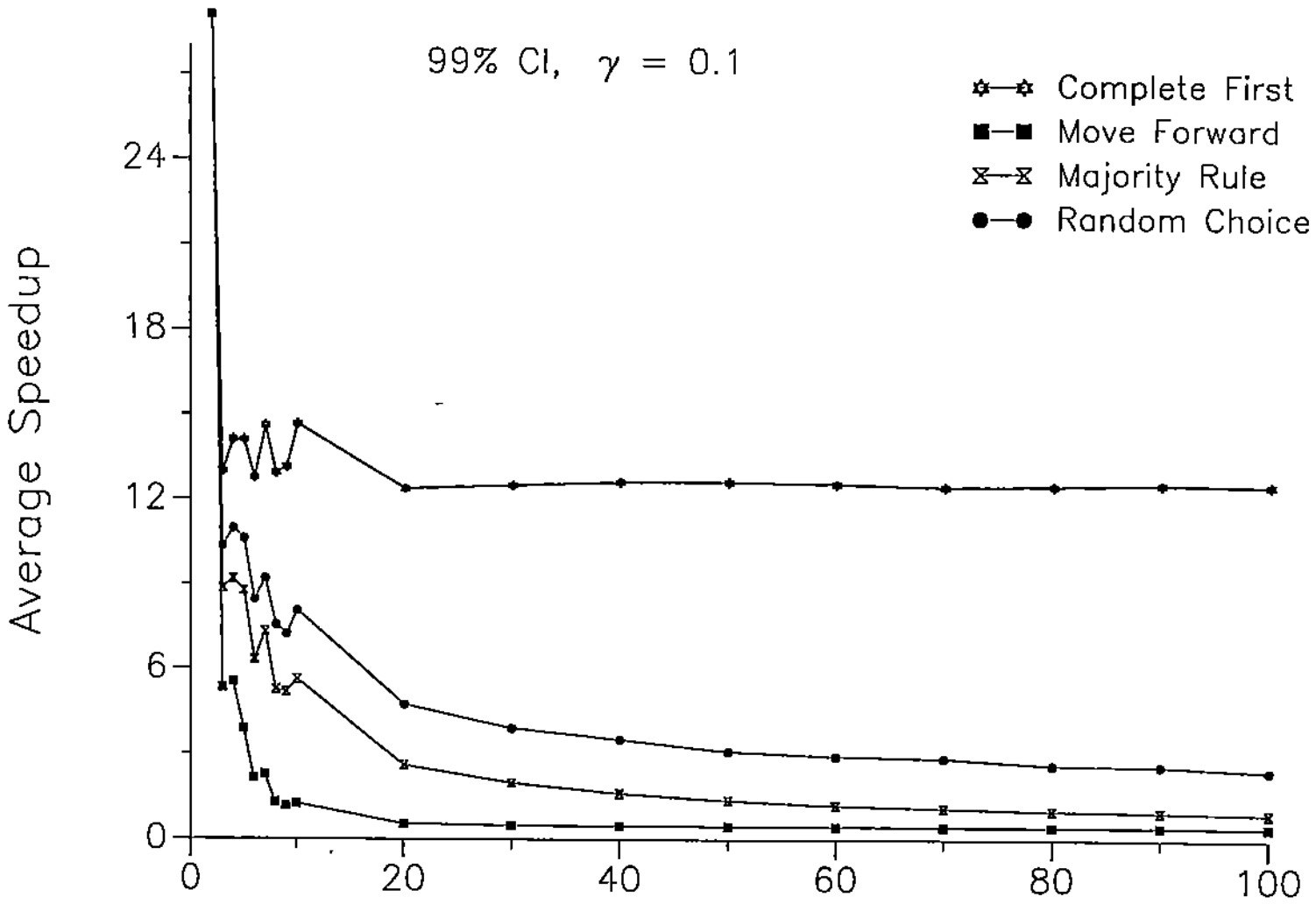
Average Speedup for policies with $\beta = 0.1$

Fig 6e.



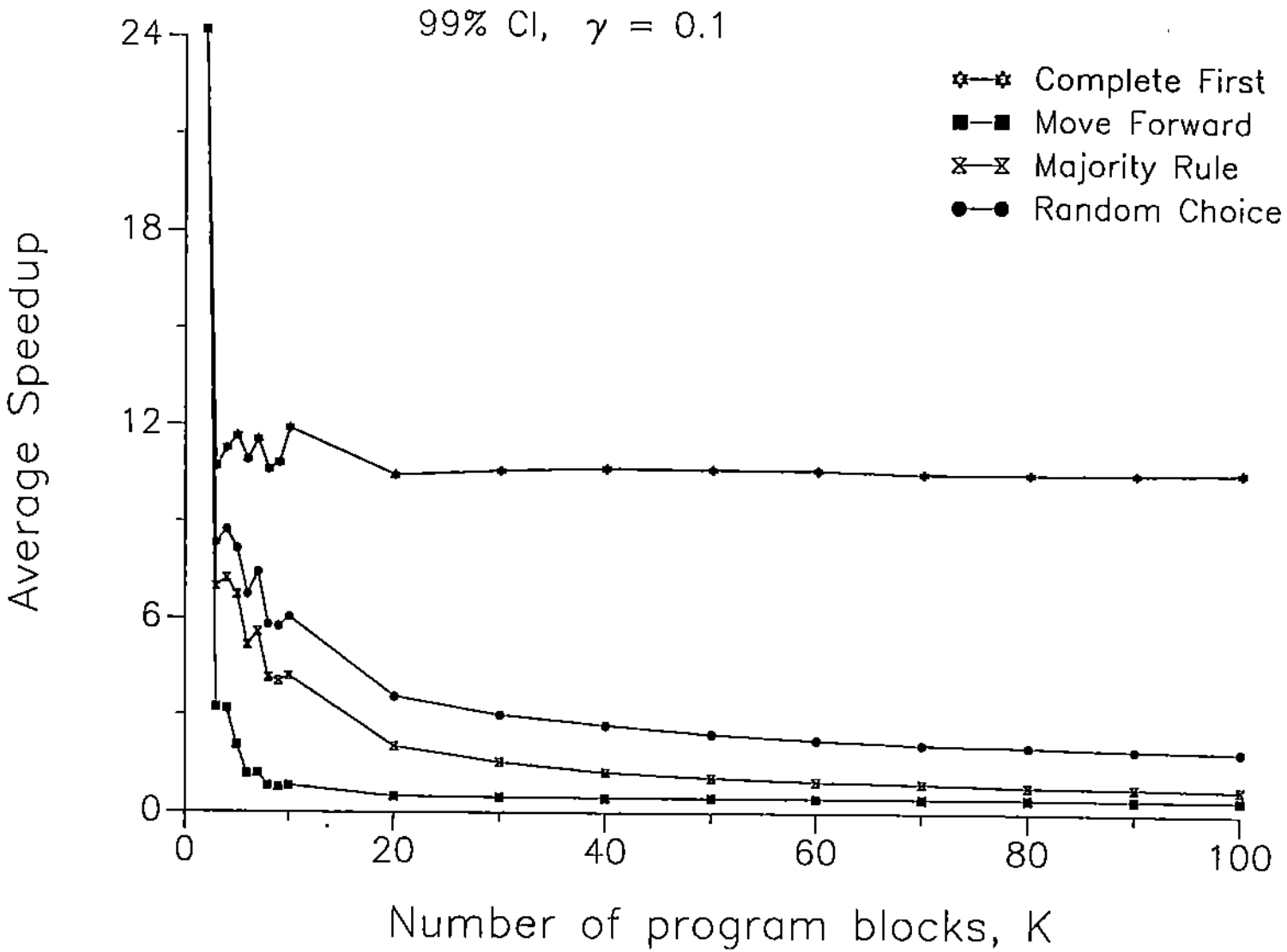
Average Speedup for policies with $\beta = 0.9$

Fig 7a.



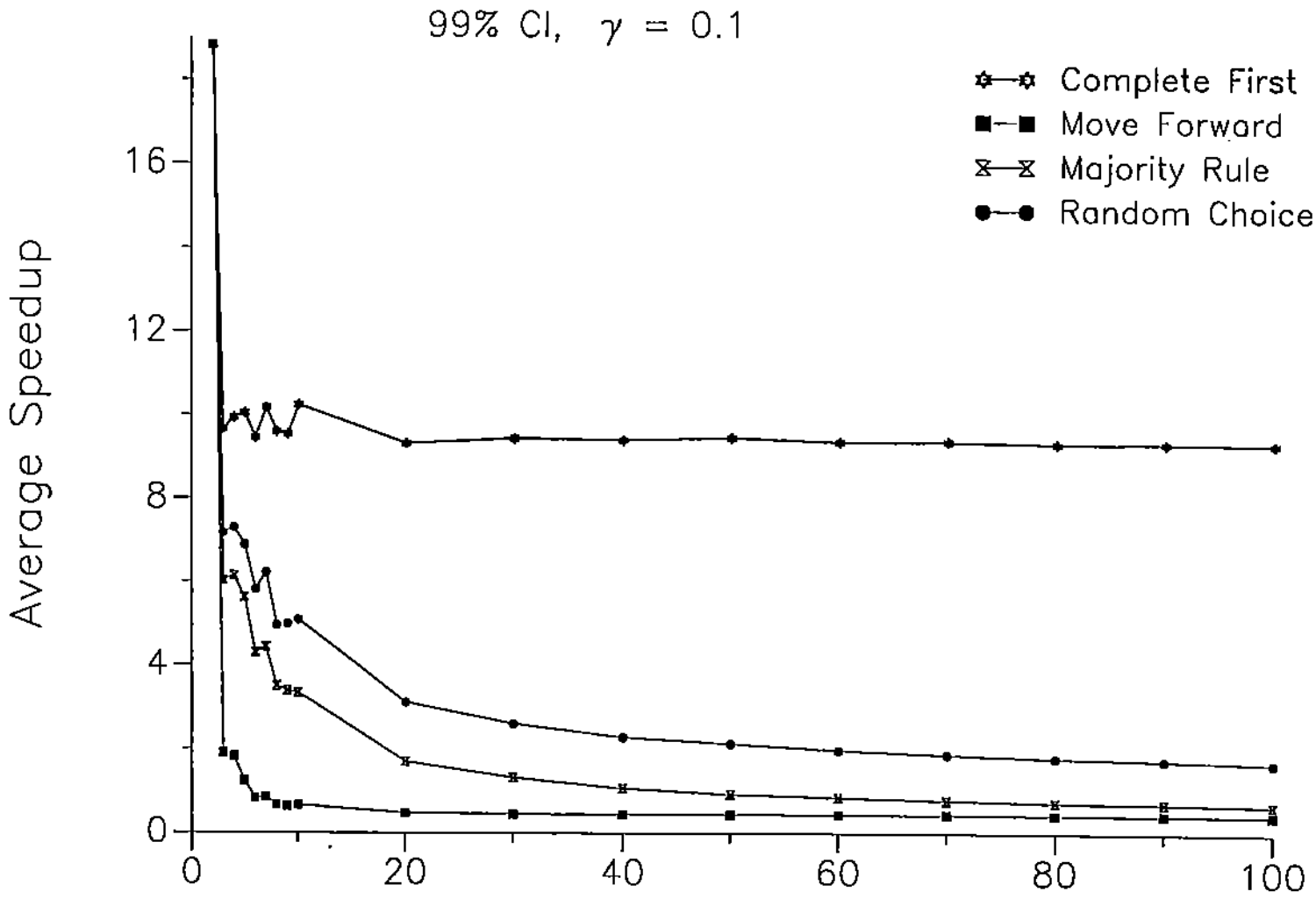
Average Speedup for policies with β = 0.7

Fig 7b.



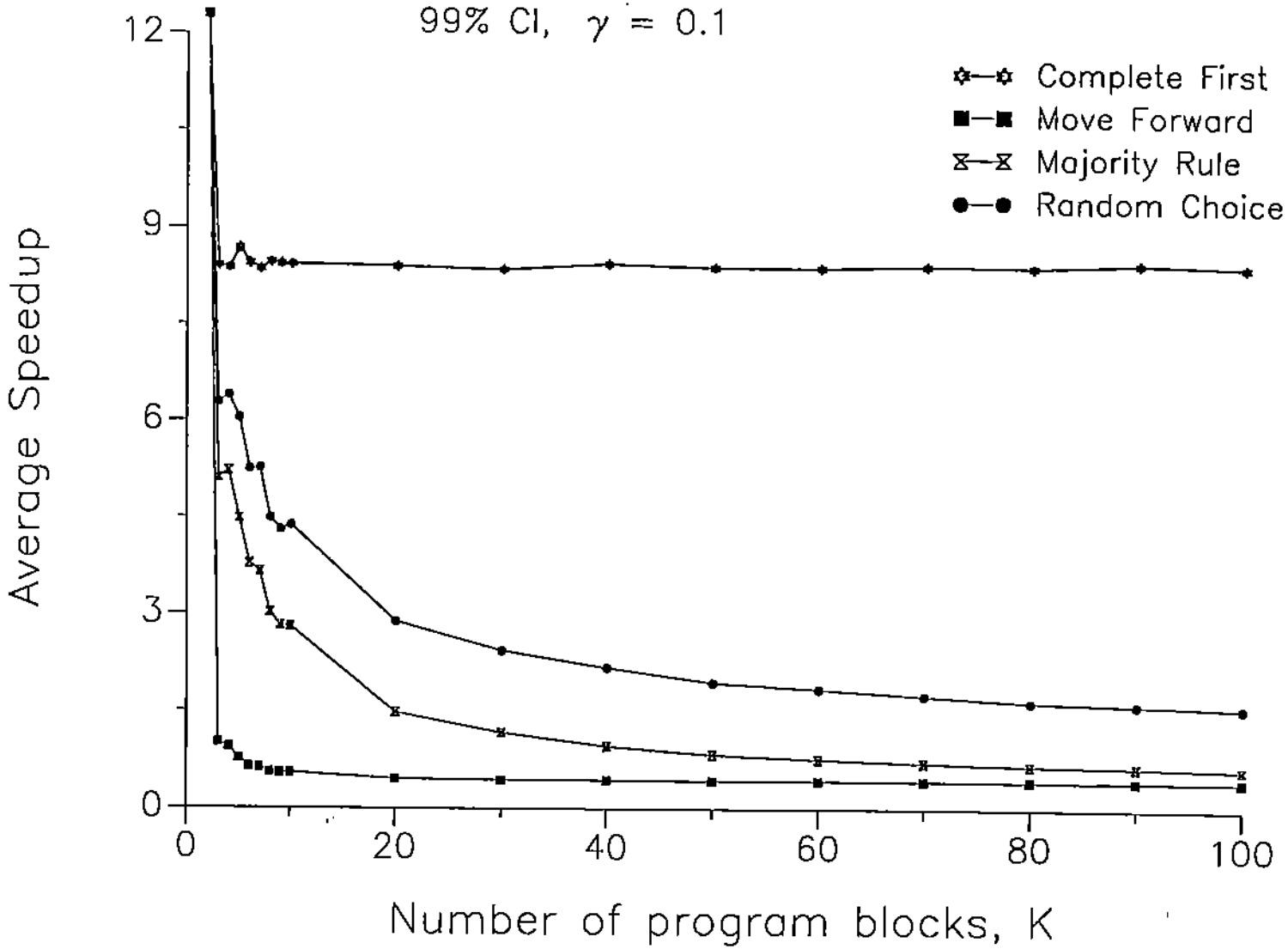
Average Speedup for policies with β = 0.5

Fig 7c.



Average Speedup for policies with β = 0.3

Fig 7d.



Average Speedup for policies with $\beta = 0.1$

Fig 7e.