T H E   U N I V E R S I T Y   O F   M I C H I G A N
COLLEGE OF LITERATURE, SCIENCE, AND THE ARTS
Computer and Communication Sciences Department

COMPARISON OF GENETIC ALGORITHMS AND GRADIENT-BASED OPTIMIZERS ON PARALLEL PROCESSORS: EFFICIENCY OF USE OF PROCESSING CAPACITY

Albert D. Bethke

November 1976

Logic of Computers Group
Technical Report No. 197

with assistance from:

National Aeronautics and Space Administration
Grant No. NSG-1176
Langly Research Center
Hampton, Virginia

UMI0339

ABSTRACT

Parallel computers such as the ILLIAC IV [3], CDC STAR, and C.mmp [12] are in limited use today. The use of microprocessors promises to make parallel processor computers very practical in the near future. In this paper, we will explore the question of performing function optimization on a parallel processor computer. In particular, we will compare Holland's reproductive plans [6], [8] (suitably modified for parallel execution) with more standard (gradient based) function optimizers (again, modified for parallelism) [2], [9].

We will first consider an elementary reproductive plan and later a more sophisticated reproductive plan. Since reproductive plans are based on models from population genetics, we may reasonably expect reproductive plans to be well suited to parallel implementation. Indeed, recasting the elementary plan into a parallel form is rather straightforward. The more sophisticated plan poses some problems however. Gradient based optimizers are easily put into parallel form, but less efficiently utilize a multi-processor than the reproductive plans.

Table of Contents

Elementary Reproductive Plan

The elementary reproductive plan has the following form:

```
generate initial (random) population
repeat until desired stopping criterion:
    compute selection probabilities
    generate new population (using crossover and mutation)
```

Each of the steps above may be done in a parallel fashion, but each step must be completed before the next step can be started. In the following assume that there are N individuals in the population denoted by A(1), A(2),...,A(N). The objective function will be denoted by f. Associated with each individual are two numbers. First, VALUE(I) will be the function value for the point represented by A(I). That is, VALUE(I) = f(A(I)). (The "fitness" of an individual is simply the function value for that individual, but all fitnesses are shifted to ensure that they are positive: fitness(I) = VALUE(I) - FMIN + 1.0, where FMIN is the smallest function value encountered so far.) Second, the selection probabilities are kept in an array PROB. PROB(I) is the probability of selecting individual A(I) to participate in an application of the crossover operator while generating a new population.

Lipton [11] suggests an ALGOL-like notation for describing parallel algorithms. In the same spirit, we will use

```
PARFOR index := initial STEP increment UNTIL final DO <statement>
```

to indicate a repetition of <statement> for the specified values of the index variable with parallel execution for different index values and arbitrary interleaving of these executions.

So, to generate the initial population:

```
PARFOR I := 1 UNTIL N DO
   A(I) := random point in domain of f;
```

To compute the selection probabilities we must do the following:

```
PARFOR I := 1 UNTIL N DO
   TEMP1(I) := TEMP2(I) := VALUE(I) := f(A(I));
compute SUM = VALUE(1) + VALUE(2) + ... + VALUE(N)
   and find new FMIN;
PARFOR I := 1 UNTIL N DO
   PROB(I) := (VALUE(I) - FMIN + 1.0)/(SUM - N*(FMIN - 1.0))
```

We may compute the sum of the function values and find the minimum function value in the same loop:

```
M := N;
WHILE M > 1 DO
   BEGIN
      PARFOR I := 1 UNTIL FLOOR(M/2) DO
         BEGIN
            TEMP1(I) := TEMP1(2*I-1) + TEMP1(2*I);
            IF TEMP2(2*I-1) < TEMP2(2*I)
               THEN TEMP2(I) := TEMP2(2*I-1)
               ELSE TEMP2(I) := TEMP2(2*I);
         END;
      IF M is odd THEN
         BEGIN
            TEMP1(CEILING(M/2)) := TEMP1(M);
            TEMP2(CEILING(M/2)) := TEMP2(M)
         END;
```

```
        M := CEILING(M/2)
     END;
  SUM := TEMP1(1);
  FMIN := TEMP2(1);
```

The parallelism is rather limited here. The PARFOR loops will be able to utilize only half as many processors each time through the WHILE loop. And there is the special case when M is odd plus the need to halve M each time. This can be improved somewhat if N is a power of two. In that case, we need not check for M being odd or find FLOOR(M/2) or CEILING(M/2). Alternatively,

let $K = \lceil \log_2(N) \rceil$, $L = 2^K$, so $N \leq L < 2N$.

And, after generating the initial population, set

TEMP1(J) = 0 and TEMP2(J) = VALUE(1), for $N < J \leq L$.

These values will remain unchanged until the reproductive plan terminates. This also eliminates the need to test for odd values of M, since we can begin with M := L which is a power of two. In fact, if we have L/2 processors, each of them can execute the same program. Processor J's program:

```
FOR COUNT := 1 UNTIL K DO
   BEGIN sum and compare block
      TEMP1(J) := TEMP1(2*J-1) + TEMP1(2*J);
      IF TEMP2(2*J-1) < TEMP2(2*J)
         THEN TEMP2(J) := TEMP2(2*J-1)
         ELSE TEMP2(J) := TEMP2(2*J);
   END sum and compare block;
```

This assumes that the processors work synchronously.

To generate the new population:

```
PARFOR I := 1 UNTIL N DO
   BEGIN
      PARENT1 := SELECT;
      PARENT2 := SELECT;
      NEW(I) := CROSS_OVER(PARENT1, PARENT2);
      mutate NEW(I)
   END;
```

SELECT is a procedure which chooses an individual from the current population at random based on the selection probabilities computed earlier. CROSS_OVER is a procedure which performs a cross-over on 2 individuals to produce a new individual. PARENT1 and PARENT2 must be local to the PARFOR loop - or better yet, there should be one PARENT1 and one PARENT2 associated with each processor. These two values could be kept in registers (rather than keeping them in storage) since they are only needed for a very short time. In addition, each processor should have its own random number seed. This seed might be kept in a register also since the random number generator is heavily used by this reproductive plan.

Finally, to replace the old population by the new population:

```
PARFOR I := 1 UNTIL N DO
   A(I) := NEW(I);
```

Notice that if we have N processors, they can all execute exactly the same program (synchronously). It is therefore very well suited for execution on a single-instruction-stream-multiple-data-stream computer. This program has only a very few conditional statements, so it would be efficient to issue the instructions for both choices (the THEN part and the ELSE part), one after the other, and disable the processors which should not execute that part. Using fewer than N processors poses no problems except the obvious scheduling problem of associating the proper index variable values (in a PARFOR loop) with the proper processor. If we use an "asynchronous multiprocessor", then the processors will have to be artificially synchronized and we may proceed only as rapidly as the slowest processor allows us. However, since each processor executes the same code, using the same arrays at the same time, we would not expect any significant differences in execution times.

Efficiency Analysis

The bottleneck for this algorithm is computing the sum of the function values and minimum function value in order to find the selection probabilities. Even with N processors, the summation cannot be done in one step. The best that can be done is to add the values in pairs following a binary tree to obtain the final result in $\log_2 N$ steps. But the computational sequence to find this sum (and the minimum value at the same time) is very short and simple. Let the time required for one processor to execute the block labeled "sum and compare block" above for only one value of COUNT be C_S. Let the time required to execute the remainder of the program (excluding initialization) be C_R. Notice, C_R includes the final step in computing selection probabilities (PROB(I) := VALUE(I)/SUM), the time required to generate a new individual, and the time to place that individual into the population. So $C_R >> C_S$.

Using N processors, the computation time for one generation is

$$\lceil \log_2 N \rceil \cdot C_S + C_R$$

Using only 1 processor, one generation requires

$(N-1)C_S + NC_R$ units of time.

And with $1 \leq P < N$ processors we have [1]

$$\left(\left\lceil \frac{N-1}{P} \right\rceil + \lceil \log_2 P \rceil - 1 \right) C_S + \left\lceil \frac{N}{P} \right\rceil C_R$$

as the computation time per generation. To carry this one step further, if $N/P = m$ where $0 < m \leq N$ and m is an integer, then the computation time per generation is

$$\left(\frac{N}{P} + \lceil \log_2 P \rceil - 1 \right) C_S + \frac{N}{P} C_R$$

Experience with reproductive plans [4], [5], [6] suggests that larger populations lead to better long term performance at the expense of the short term performance. So the population size must be chosen accordingly. For sequential machines, populations larger than a few hundred individuals generally lead to unacceptably slow optimization. For $N \leq 200$, and with $C_S << C_R$ (say $C_S < \frac{1}{20} C_R$), the computation time per generation using P processors is well approximated by

$$T_P \approx \left\lceil \frac{N}{P} \right\rceil C_R$$

For larger populations, this approximation is not as good unless the ratio C_R/C_S is increased also. See figures 6-10.

Define the "speed-up (coefficient)" for P processors to be the ratio of the execution times using only 1 processor and using P processors. Then

$$S_P = \frac{T_1}{T_P} \approx \frac{NC_R}{\left\lceil \frac{N}{P} \right\rceil C_R} = \frac{N}{\left\lceil \frac{N}{P} \right\rceil}$$

Now define the "processor utilization" or "efficiency" for P processors to be the fraction of time during which each processor is busy (average over the P processors). Notice this is just $\frac{S_P}{P}$. So

$$E_P = \frac{S_P}{P} \approx \frac{\frac{N}{P}}{\left\lceil \frac{N}{P} \right\rceil} \approx 1^*$$

*$\frac{\frac{N}{P}}{\left\lceil \frac{N}{P} \right\rceil}$ never exceeds 1 and is minimal for $P = N - 1$. For $P = N - 1$,

$E_P \approx \frac{1}{2}$. Using P processors where P does not divide N is clearly wasteful since some processors remain idle during part of each PARFOR loop. See figures 1-5.

Therefore, the processors are kept busy nearly all the time. The elementary reproductive plan lends itself well to parallel implementation.

Using more than N processors would reduce the computation time even more. The loop which does the summation and finds the minimum of the function values for the current population could be sped up by having 2 or 3 or even more processors executing the program for processor J. The addition of TEMP1(2*J-1) + TEMP1(2*J) and the comparison of the TEMP2 values can be done concurrently. Some of the sections of the rest of the program can be similarly sped up by using more than N processors. PARENT1 and PARENT2 can be chosen simultaneously for example. And the code for SELECT and CROSSOVER and MUTATE undoubtably can be implemented with reasonable efficiency using a small number of processors. So using 2N or 3N or maybe mN, for small m, processors should reduce the computation time per generation by almost m over the time required with only N processors. Synchronizing the processors would be more difficult and there might be some not yet discovered bottlenecks which would reduce the efficiency of using more than N processors. In any case, if N is 50, 100, or 200, 2N or 3N processors may not be available.

Sophisticated Reproductive Plan

The plan we will consider was developed by deJong [6] and performs much better than the simple reproductive plan. The form of this sophisticated plan is:

```
generate initial (random) population
repeat until desired stopping criterion:
   compute selection probabilities
   generate new individuals
   incorporate new idividuals into population
```

Here we do not replace the entire population at once, but only $G \leq N$ individuals each generation. Other differences between the elementary and sophisticated plans include forcing each individual to have very nearly the expected number of offspring (based on selection probabilities), and to apply crossover with a probability less than 1 (the elementary plan always applied crossover in generating new individuals). These and other differences will become more apparent as the sophisticated plan is put into parallel form.

The initial population may be generated as before.

The selection probabilities are computed in the same way as for the elementary plan with one exception - in the same loop that sums the function values and finds the minimum value, we also find the index of the best individual. (The best individual is saved from one generation to the next to ensure that the optimizer does not "regress".)

```
PARFOR I := 1 UNTIL N DO
   BEGIN
      TEMP1(I) := TEMP2(I) := VALUE(I) := f(A(I));
      BEST(I) := I
   END
M := N;
WHILE M > 1 DO
   BEGIN
      PARFOR I := 1 UNTIL FLOOR(M/2) DO
         BEGIN
            TEMP1(I) := TEMP1(2*I-1) + TEMP1(2*I);
            IF TEMP2(2*I-1) < TEMP2(2*I)
               THEN TEMP2(I) := TEMP2(2*I-1)
               ELSE TEMP2(I) := TEMP2(2*I);
            IF VALUE(BEST(2*I-1)) > VALUE(BEST(2*I))
               THEN BEST(I) := BEST(2*I-1)
               ELSE BEST(I) := BEST(2*I)
         END
      IF M is odd THEN
         BEGIN
            TEMP1(CEILING(M/2)) := TEMP1(M);
            TEMP2(CEILING(M/2)) := TEMP2(M);
            BEST(CEILING(M/2)) := BEST(M)
         END;
      M := CEILING(M/2)
   END;
```

```
SUM := TEMP1(1);
FMIN := TEMP2(1);
BEST_INDEX := BEST(1);
save best string if desired;
PARFOR I := 1 UNTIL N DO
   PROB(I) := VALUE(I)/SUM;
```

Again, the parallelism can be improved greatly by using some dummy variables. Let

$$K = \lceil \log_2 N \rceil,\ L = 2^K,\ \text{so } N \leq L < 2N$$

After generating the initial population, set

$$\text{TEMP1}(J) = 0,\ \text{TEMP2}(J) = \text{VALUE}(1),\ \text{BEST}(J) = J,\ \text{for } N < J \leq L.$$

These values will not be changed by the reproductive plan. With L/2 processors the WHILE loop above can be effectively carried out if each processor follows this program:

```
FOR COUNT := 1 TO K DO
   BEGIN
      TEMP1(J) := TEMP1(2*J-1) + TEMP1(2*J);
      IF TEMP2(2*J-1) < TEMP2(2*J)
         THEN TEMP2(J) := TEMP2(2*J-1)
         ELSE TEMP2(J) := TEMP2(2*J);
      IF VALUE(BEST(2*J-1)) > VALUE(BEST(2*J))
         THEN BEST(J) := BEST(2*J-1)
         ELSE BEST(J) := BEST(2*J)
   END;
```

Here J is the index of the processor. This requires that the processors act synchronously.

To generate G new individuals to be incorporated into the population later, requires the following sort of structure.

```
PARFOR NEW := 1 UNTIL G DO
    IF random value $\leq$ crossover probability
        THEN
            BEGIN
                UNTIL successful selection
                    BEGIN
                        PARENT1 := SELECT;
                        test and decrement PARENT1's offspring counter
                    END
                UNTIL successful selection
                    BEGIN
                        PARENT2 := SELECT;
                        IF PARENT1 $\neq$ PARENT2
                            THEN test and decrement PARENT2's offspring
                                counter
                    END
                NEW(I) := CROSS_OVER(PARENT1, PARENT2);
            END
        ELSE no crossover
            BEGIN
                UNTIL successful selection
                    BEGIN
                        PARENT1 := SELECT:
                        test and decrement PARENT1's offspring counter
                    END
            NEW(I) := A(PARENT1)
         END;
      mutate NEW(I);
```

The test and decrement operation must be a single indivisible operation such that if more than one processor attempts to modify the same counter all but one will be "locked out" until the first processor completes the test and decrement. The test portion of the test and decrement operation tells whether the counter is positive or not. If the counter is positive, then that individual has not yet had its expected number of offspring and may be selected successfully as a "parent". The indivisibility of the test and decrement operation ensures that the same individual cannot be simultaneously selected by several processors and have its counter decremented to a very negative value (and have more than the expected number of offspring). Of course, the same individual can be selected in very rapid succession by several processors and could be involved in several crossover operations at the same time if the number of expected offspring allowed that.

Incorporating the new individuals into the population is done in one of two ways. If $G \leq N/2$, then G individuals from the current population are replaced by the new individuals. If $G > N/2$, then N-G individuals are chosen from the current population to be retained in the next generation. DeJong's work [6] suggests G should be about N/10, so the first method is almost always the method to be used. DeJong also found that the performance of this reproductive plan is improved if, for each new individual, a small number of candidates is chosen and the candidate most similar to the new individual is replaced by the new individual.

The number of such candidates was called the crowding factor (CF). So the replacement process may be done in this way:

```
PARFOR I := 1 UNTIL G DO
    BEGIN
        PARFOR J := 1 UNTIL CF DO
            BEGIN
                choose one individual from the current population
                    at random (uniform distribution) call this
                    individual A(R(J));
                find Hamming distance between A(R(J)) and
                    NEW(I) = HD(J);
            END
        find minimum of HD(1), HD(2),...,HD(CF), call it
            HD(BEST_MATCH);
        A(BEST_MATCH) := NEW(I);
    END;
```

The replacement operation A(BEST_MATCH) := NEW(I) presents us with the difficulty that several processors may simultaneously replace the same individual with several new individuals causing some of the new individuals to be lost. The solution is to mark each individual as to whether or not it has been replaced yet and then "test and set" this flag when replacing an individual. This can be accomplished with the same test and decrement instruction needed earlier. When an individual is selected for replacement which is marked as already replaced, then another individual must be chosen. Thus, the program becomes:

```
PARFOR I := 1 UNTIL G DO
   BEGIN
      UNTIL REPLACEMENT_FLAG(BEST_MATCH) = not replaced* DO
         BEGIN
            PARFOR J := 1 UNTIL CF DO
               BEGIN
                  UNTIL REPLACEMENT_FLAG(R(J)) = not replaced DO
                     R(J) := RANDOM(1,N);
                  HD(J) := Hamming distance between A(R(J)) and
                     NEW(I);
               END;
            BEST_MATCH := index corresponding to minimum of
               HD(1), HD(2),...,HD(CF);
         END;
      A(BEST_MATCH := NEW(I);
   END;
```

At this point, the more sophisticated reproductive plan has been adapted for parallel implementation. We should mention that the sophisticated plan uses a generalized crossover operator so that the number of crossover points may be greater than one. This should have very little effect on the parallel implementation of the algorithm except that all

*Notice, the test that A(BEST_MATCH) has not already been replaced must be done using the "test and set" (or possibly a "test and decrement") operation for the reason stated above. On the other hand, the test to see whether A(R(J)) has been replaced yet should *not* use the "test and set" operation because this step only finds *candidates* for replacement and does not replace any individual.

of the crossover points can be chosen at once if enough processors are available. (The number of crossover points should probably be a very small number - 1 or 2 or 3. So this may not represent much of a savings.)

Efficiency Analysis

This sophisticated reproductive plan presents some real difficulties to parallel implementation. In the simple plan, it was natural to associate one processor with each of the new individuals to be generated during one generation. We may do that for the sophisticated plan, but here we face the problem of avoiding "collisions" in choosing parents and again when incorporating the new individuals into the population. If we were using G processors - one for each new individual - and a collision occured while selecting which individuals will be replaced, then G-1 processors would be idle while another replacement choice was made by one processor. This would be rather inefficient, unless it were infrequent. If there were only a few processors, such collisions would be less costly (indeed, with only 1 processor efficiency is 100%). There may be some effective way of avoiding these collisions, or some better form of this algorithm for parallel implementation, but we have not found one.*

As the algorithm stands, one can rather easily estimate the expected number of collisions, which is independent of the number of processors.** Assume that m individuals have already been chosen to be replaced. Then

*We considered the possibility of having one processor dedicated to choosing parents and coordinating replacements, but such a processor could not keep up with the other processors. Also, using a multiple of G processors one could choose several candidates for replacement at once, but then finding a distinct subset of G of these is the original problem again.

**Consider rolling 3 dice. The probability of the same number appearing on more than one die is independent of how many dice are rolled at one time. (3 at once, or roll each by itself,...)

the probability of choosing an unchosen individual on the next attempt is simply

$$P_m = \frac{N-m}{N}$$

So the expected number of trials before a successful selection is (since this is a Bernoulli sequence)

$$w_m = \frac{N}{N-m}$$

Thus, the expected number of trials required to choose G distinct individuals from the population is:

$$W_G = \sum_{m=0}^{G-1} w_m = \sum_{m=0}^{G-1} \frac{N}{N-m}$$

$$= N \sum_{k=N-G}^{N} \frac{1}{k}$$

$$= N(H_N - H_{N-G}), \text{ where } H_n = \sum_{k=1}^{n} \frac{1}{k}$$

$$\approx N\left[\ln N + \frac{1}{2N} - \ln(N-G) - \frac{1}{2(N-G)}\right]$$

$$W_G \approx N \ln\left(\frac{N}{N-G}\right) - \frac{G}{2(N-G)}$$

If N = 10 G, then

$$W_G \approx N \ln\left(\frac{10}{9}\right) - \frac{1}{18} \approx 0.10537\ N \approx 1.05\ G$$

So we would expect about 5% of the trials to result in collisions. We have not estimated the variance in the number of collisions, but it seems reasonable to believe that most generations will not involve collisions in choosing which individuals are to be replaced. The same analysis applies to selection of parents to generate the G new individuals. Here we use a non-uniform distribution, but the probability of selecting an acceptable parent (one with a positive offspring counter) after having chosen m previously is still

$$P'_m = \frac{N-m}{N}$$

since the total number of allowed offspring is N and the remaining number is N-m.

Therefore, the computation time per generation is roughly

$$T_P \approx \left(\left\lceil\frac{N-1}{P}\right\rceil + \lceil\log_2 P\rceil - 1\right)C_S + \left\lceil\frac{G}{P}\right\rceil\left(\frac{W_G}{G}C_t + C_R\right)$$

where C_t and C_R are the time required to choose one individual as a parent (or for replacement) and the time required to do everything else to generate new individuals and incorporate them into the population. In particular:

$$T_1 \approx (N-1)C_S + W_G C_t + GC_R$$

$$T_G \approx \left(\left\lceil\frac{N-1}{G}\right\rceil + \lceil\log_2 G\rceil - 1\right)C_S + \frac{W_G}{G}C_t + C_R$$

So if $C_R >> C_S$, and $N < 1000$, then

$$T_P \approx \left\lceil\frac{G}{P}\right\rceil\left(\frac{W_G}{G}C_t + C_R\right)$$

Now this makes

$$S_P = \frac{T_1}{T_P} \approx \frac{W_G C_t + GC_R}{\left\lceil\frac{G}{P}\right\rceil\left(\frac{W_G}{G}C_t + C_R\right)} = \frac{G}{\left\lceil\frac{G}{P}\right\rceil} \approx P$$

And $$E_P = \frac{S_P}{P} \approx \frac{\frac{G}{P}}{\left\lceil\frac{G}{P}\right\rceil} \approx 1$$

So, again, the reproductive plan utilizes the processors quite well. But the approximations used in this derivation are not so good as those for the simpler plan. Further investigation is needed to really discover how efficient the sophisticated reproductive plan is.

There is some parallelism possible within the code which generates a new individual and also in the code to place that individual into the population. So using 2G or 3G processors might be relatively efficient, but we have not investigated this possibility.

Standard (Gradient based) Function Optimizers

Standard function optimizers all have one thing in common. They must estimate the gradient of the function at each sample point. In order to do this, either a procedure must be provided to generate an array of partial derivatives (based on algebraic derivatives of the function), or else the optimizer must sample the function at nearby points along each dimension of the domain. The first method is hard to evaluate since it depends very strongly on the form of the objective function. The second method is much easier to deal with, and also more general, so we assume that D (the dimensionality of the domain of the objective function), or possibly 2D, function evaluations are made at points near the current sample point in order to get the gradient information required for choosing the next sample point.

This suggests that using D processors would allow the partial derivatives to all be estimated in parallel in very short time. Indeed, those optimizers which do a large amount of matrix manipulations (using the Hessian) can do the matrix operations in parallel as well if there are D processors. Matrix operations are generally not 100% efficient in using parallel processing [7], [10]. In many cases, the bottleneck is a summation which must be computed. Matrix multiplication using D processors and 2 DxD matrices requires about

$$D^2(\lceil \log_2 D \rceil + 1)$$

computation steps (there are D^3 multiplications and D^2 summations of D values). Using D^3 (or more) processors the computation time is

$$\lceil \log_2 D \rceil + 1$$

Notice, finding the magnitude of the gradient is limited by this same summation bottleneck.

Function optimizers which use a one-dimensional search to generate the next sample point can make use of parallel processing to speed up that search. Instead of using a binary or golden section search, use a P-ary search. Find function values at P evenly spaced points. Compare these to determine which interval to continue searching. The comparison step is the bottleneck here. It requires $\lceil \log_2 P \rceil$ steps to make this P-way comparison. Also, speed-up due to using P-ary search rather than binary or Fibonacci search is only a logarithmic improvement [2], [9]. Using D processors and neglecting the comparison bottleneck, the speed-up is approximately

$$S_D \approx \log_\theta\left(\frac{D+2}{2}\right), \text{ where } \theta = \frac{1+\sqrt{5}}{2} .$$

So

$$S_D \approx \lceil \log_2 D \rceil \log_\theta\left(\frac{D+2}{2}\right) .$$

Thus

$$E_D = \frac{S_D}{D} \approx \frac{\lceil \log_2 D \rceil \log_\theta\left(\frac{D+2}{2}\right)}{D}$$

This is badly sub-optimal for large D. Moreover, the standard optimizer's rate of convergence decreases rapidly as D becomes large. So parallel implementation does not seem to be the solution to getting good performance on functions of a large number of variables using gradient-based optimizers.

Another obvious approach to parallel implementation of the standard type of optimizer is to use P processors each of which runs the function optimizer independently of the other processors. For a unimodal function, the speed-up is essentially zero. The only improvement is that the starting point closest to the optimum is probably better than if only one starting point had been chosen. For multimodal functions, this allows a crude global search to be made at the same time as the local search is performed. Commonly, one would run his favorite gradient-based optimizer several times, starting at points spread throughout the domain. So the speed-up is almost perfect:

$$S_P \approx P$$

and

$$E_P \approx 1$$

using P processors on a multi-modal function. This must be taken with a grain of salt, however, since standard optimizers are not very well suited to multi-modal functions.

Summary

In short, the simple genetic plan seems very well suited to parallel implementation. The sophisticated reproductive plan makes less efficient use of parallel processors, but it is still reasonably efficient for $P \leq G$ (where G is likely to be on the order of 10). The standard, gradient-based function optimizer is less suited to parallel implementation than either reproductive plan, but it may be advantageous to use parallel processing on multi-modal functions.

References

1. Arjomandi, E. "A Study of Parallelism in Graph Theory". Doctoral Thesis, Dept. of Computer Science, Univ. of Toronto. Also Technical Report No. 86, Dept. of Computer Science, Univ. of Toronto. Dec., 1975.

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

3. Barnes, G. H., R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnik, R. A. Stoker. "The Illiac IV Computer". IEEE Trans. on Comp. 17, 1968, p. 746-757.

4. Bethke, A. D., B. P. Zeigler, D. M. Strauss. "Convergence Properties of Simple Genetic Algorithms". Technical Report No. 159, Logic of Computers Group, Univ. of Michigan. July, 1974.

5. Bosworth, J. L., N. Foo, B. P. Zeigler. "Comparison of Genetic Algorithms with Conjugate Gradient Methods". Technical Report No. 00312-1-T, Logic of Computers Group, Univ. of Michigan.

6. DeJong, K. A. "Analysis of the Behavior of a class of Genetic Adaptive Systems". Doctoral Thesis, Dept. of Computer and Communication Sciences, Univ. of Michigan, 1975. Also, Technical Report No. 185, Logic of Computers Group, Univ. of Michigan.

7. Heller, D. "A Survey of Parallel Algorithms in Numerical Linear Algebra". Technical Report, Dept. of Computer Science, Carnegie-Mellon Univ. 1976.

8. Holland, J. H. Adaptation in Natural and Artificial Systems. Univ. of Michigan Press, 1975.

9. Karp, R. M., W. L. Miranker. "Parallel Minimax Search for a Maximum". Journal of Combinatorial Theory, 4, 1968, 19-35.

10. Kung, H. T. "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors". Technical Report, Dept. of Computer Science, Carnegie-Mellon Univ. 1976. Also to appear in New Directions and Recent Results in Algorithms and Complexity, edited by J. F. Traub, Academic Press, 1976.

11. Lipton, R. J. "Reduction: A Method of Proving Properties of Parallel Programs". CACM, 18, 1975, p. 717-721.

12. Wulf, W. A. and C. G. Bell. "C.mmp - A Multi-Mini-Processor". Proc. AFIPS 1972 FJCC. Vol. 41 Part II, AFIPS Press, 1972, p. 765-777.

Explanation of Figures

Figures 1-5 show how good the approximation

$$E_P \approx \frac{\frac{N}{P}}{\left\lceil \frac{N}{P} \right\rceil}$$

is for various values of N. This approximation is compared to

$$E_P = \frac{T_1}{T_P}$$

where $T_1 = (N-1)C_S + NC_R$

and $T_P = \left(\left\lceil \frac{N}{P} \right\rceil + \lceil \log_2 N \rceil - 1\right)C_S + \left\lceil \frac{N}{P} \right\rceil C_R$

using the ratio $\frac{C_R}{C_S} = 10$.

Figures 6-10 compare the approximation

$$T_P \approx \left\lceil \frac{N}{P} \right\rceil C_R$$

with $T_P = \left(\left\lceil \frac{N}{P} \right\rceil + \lceil \log_2 N \rceil - 1\right)C_S + \left\lceil \frac{N}{P} \right\rceil C_R$

for the same values of N and $\frac{C_R}{C_S} = 10$ also.

EFFICIENCY OF SIMPLE REPRODUCTIVE PLAN

POPULATION SIZE = 10

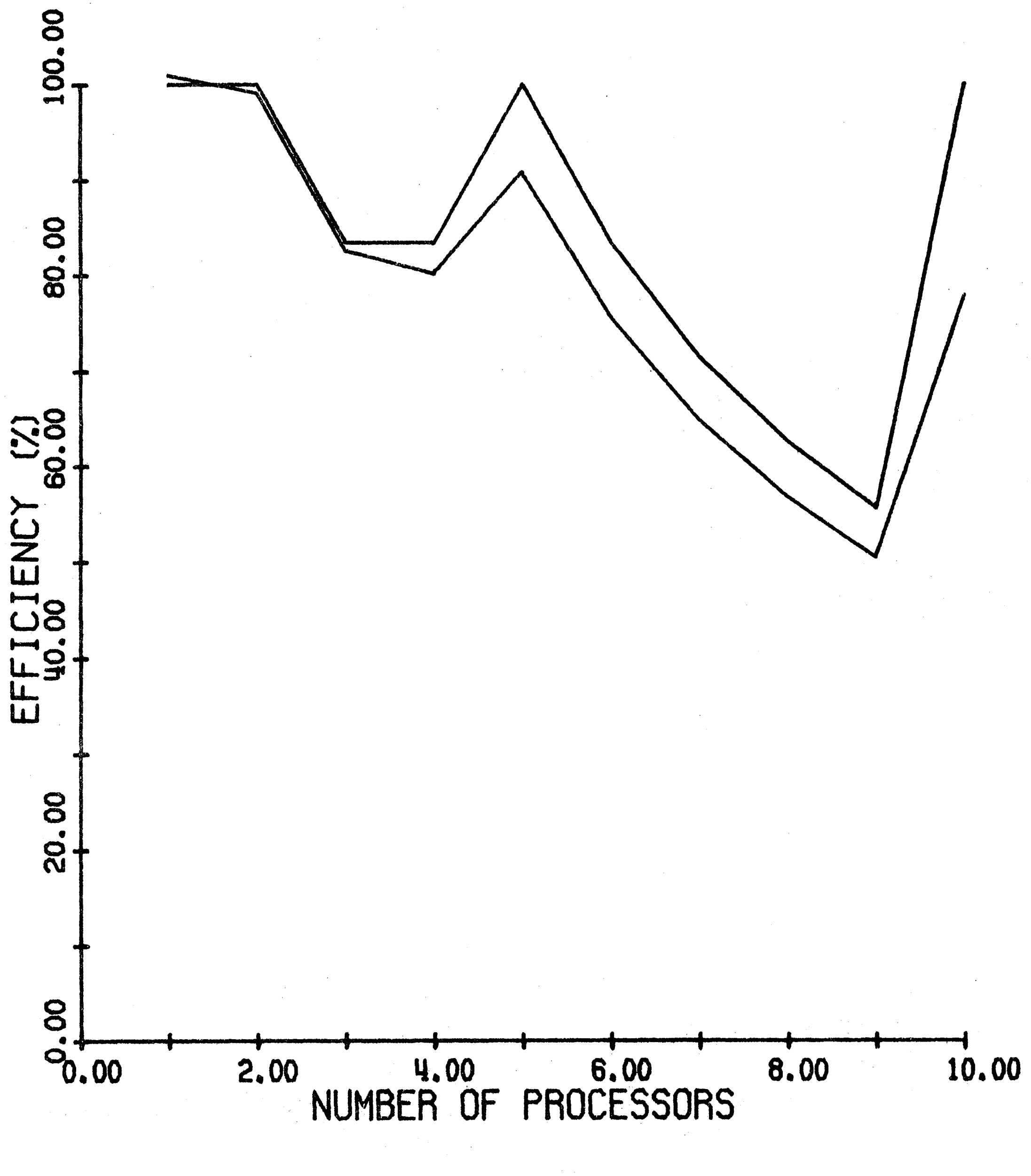

FIGURE 1

EFFICIENCY OF SIMPLE REPRODUCTIVE PLAN

POPULATION SIZE = 100

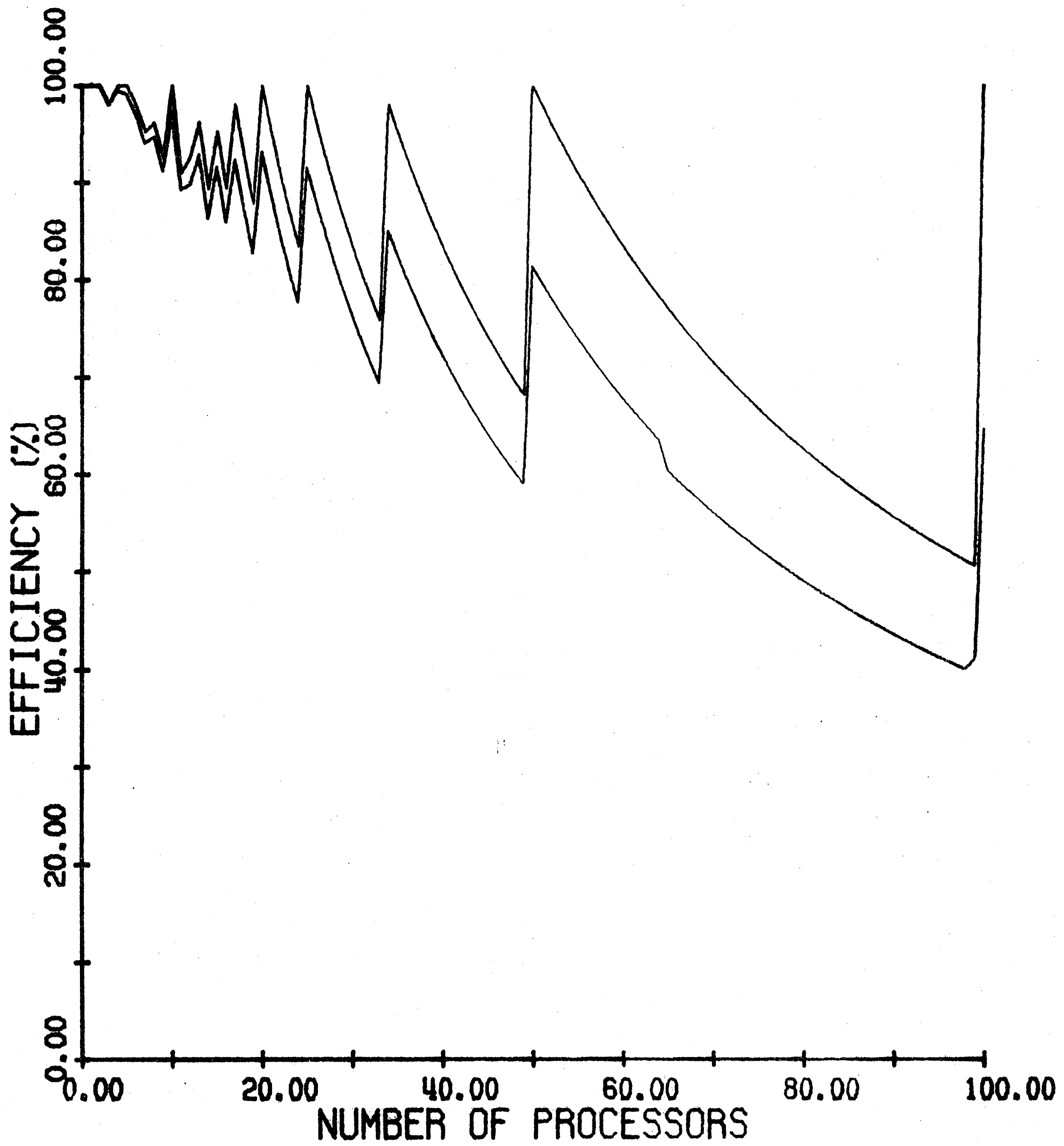

FIGURE 2

EFFICIENCY OF SIMPLE REPRODUCTIVE PLAN

POPULATION SIZE = 1000

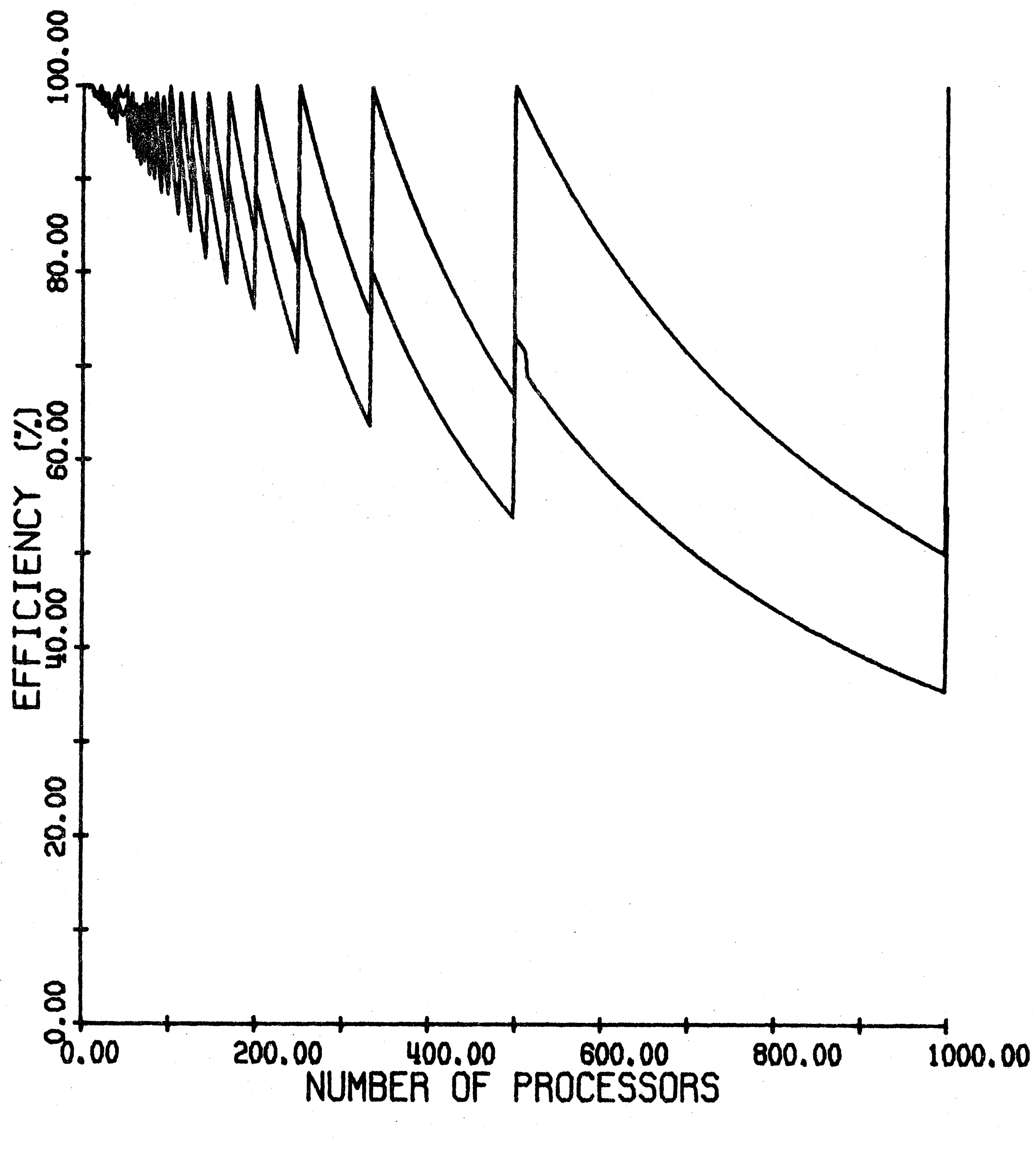

FIGURE 3

EFFICIENCY OF SIMPLE REPRODUCTIVE PLAN

POPULATION SIZE = 10000

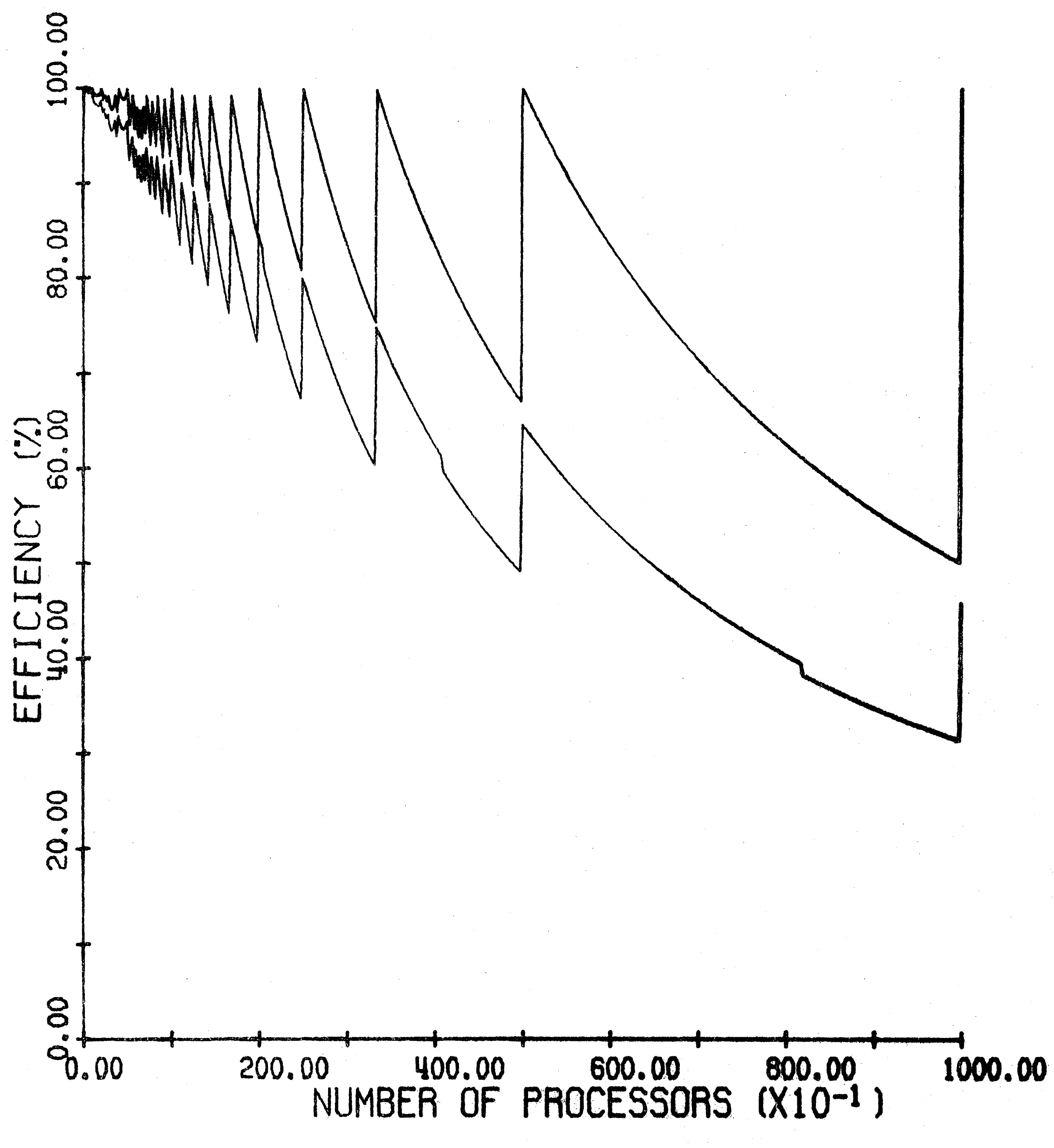

FIGURE 4

EFFICIENCY OF SIMPLE REPRODUCTIVE PLAN

POPULATION SIZE = 100000

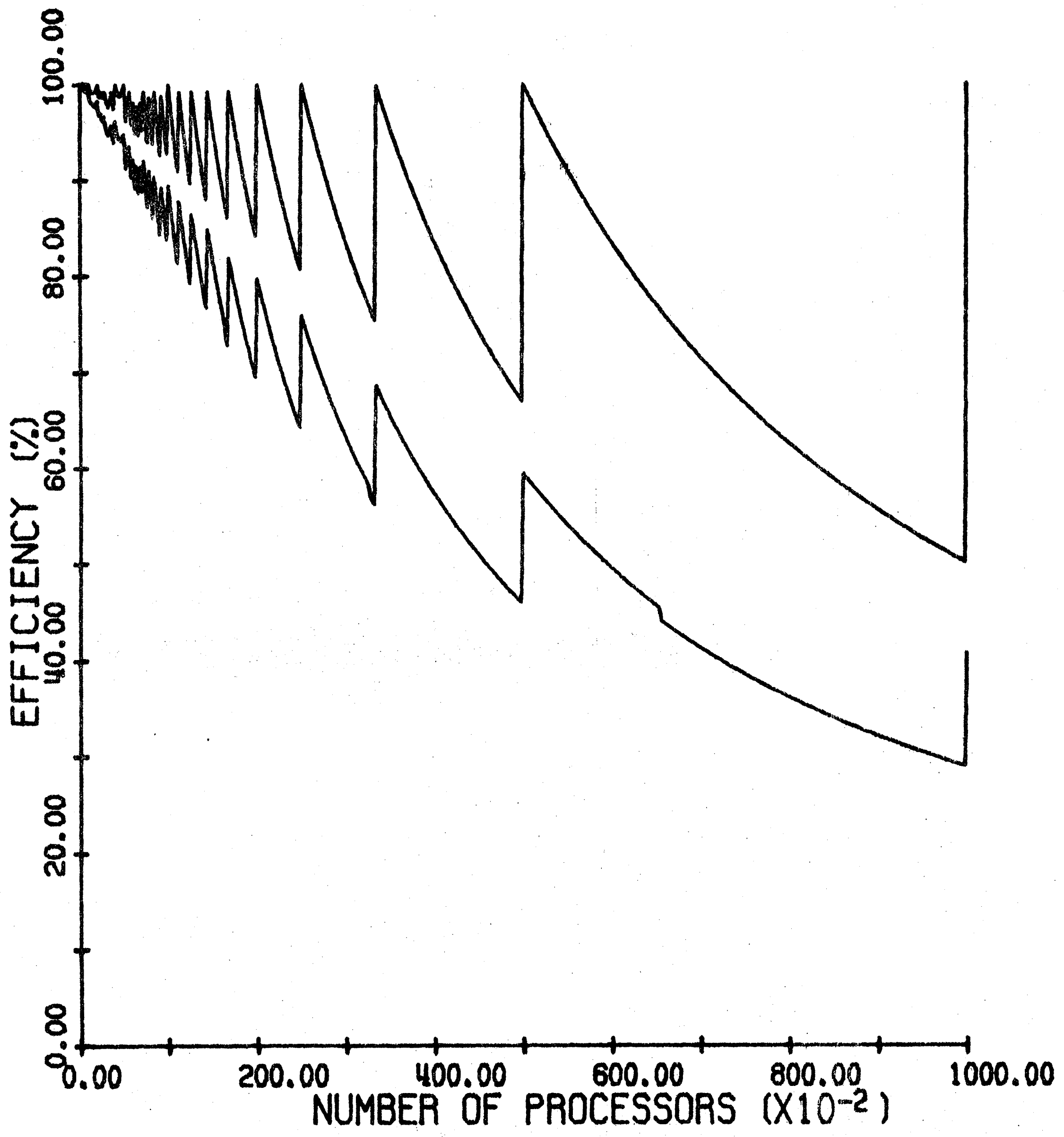

FIGURE 5

COMPUTATION TIME OF SIMPLE REPRODUCTIVE PLAN

POPULATION SIZE = 10

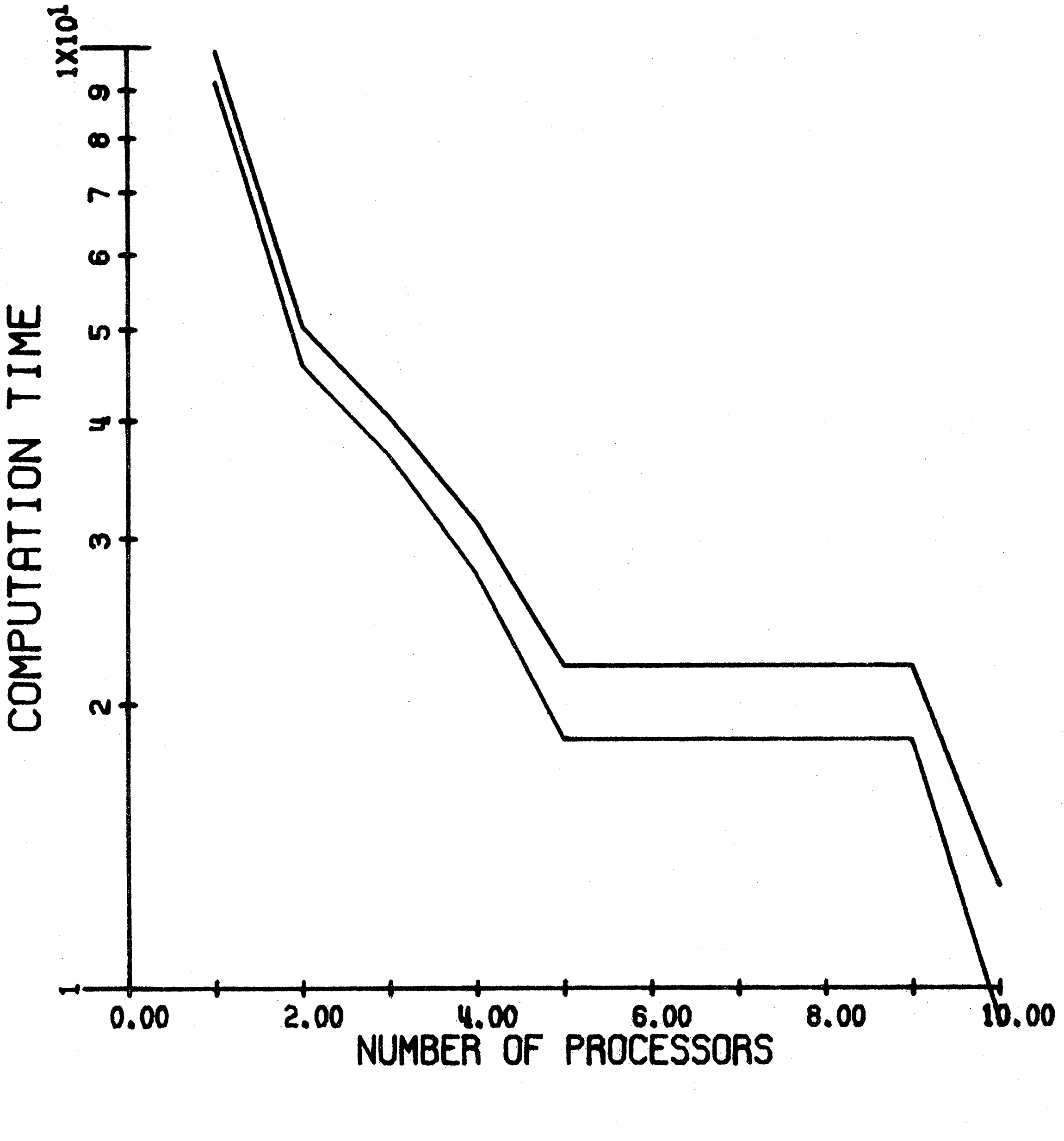

FIGURE 6

COMPUTATION TIME OF SIMPLE REPRODUCTIVE PLAN

POPULATION SIZE = 100

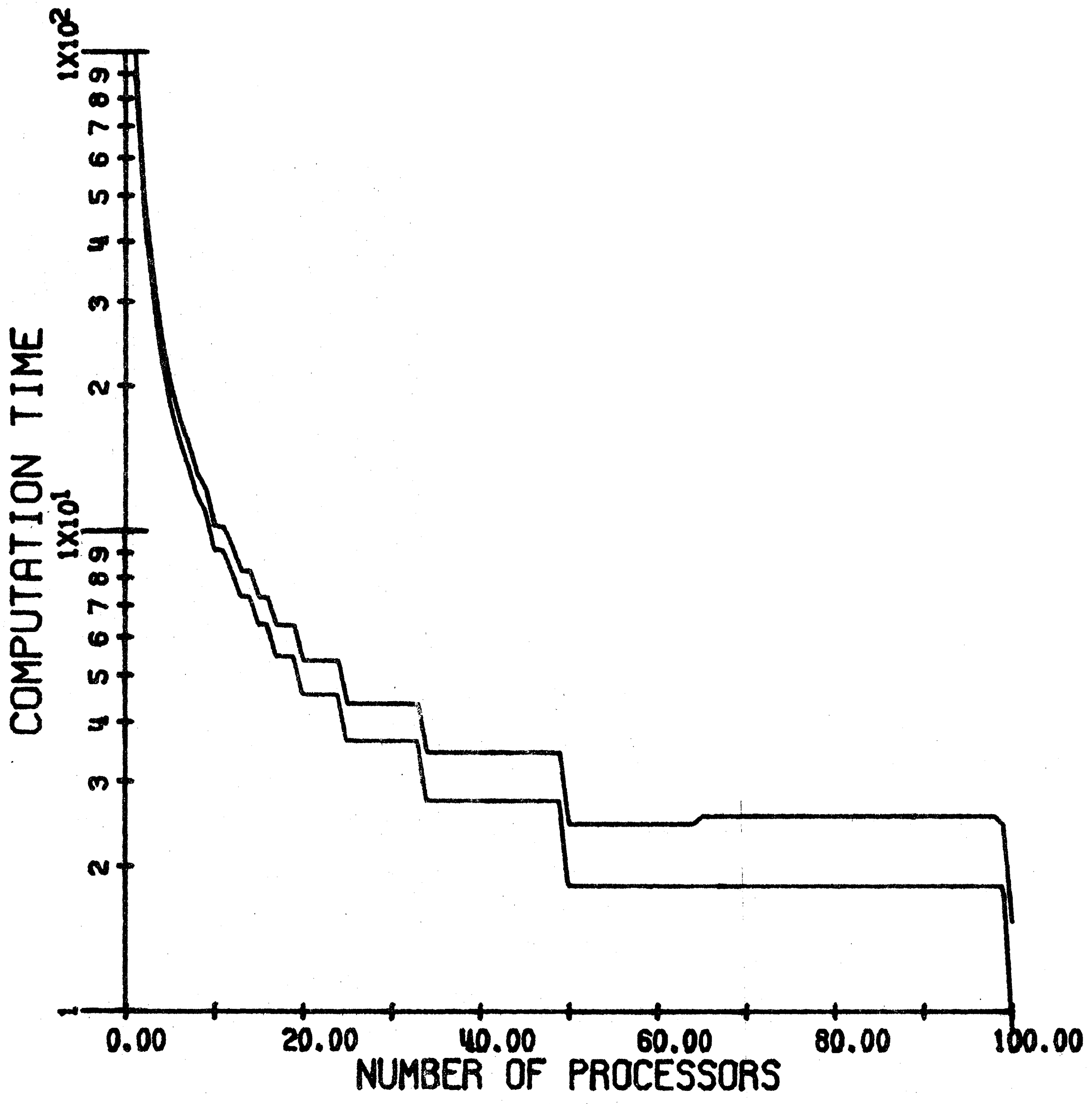

FIGURE 7

COMPUTATION TIME OF SIMPLE REPRODUCTIVE PLAN

POPULATION SIZE = 1000

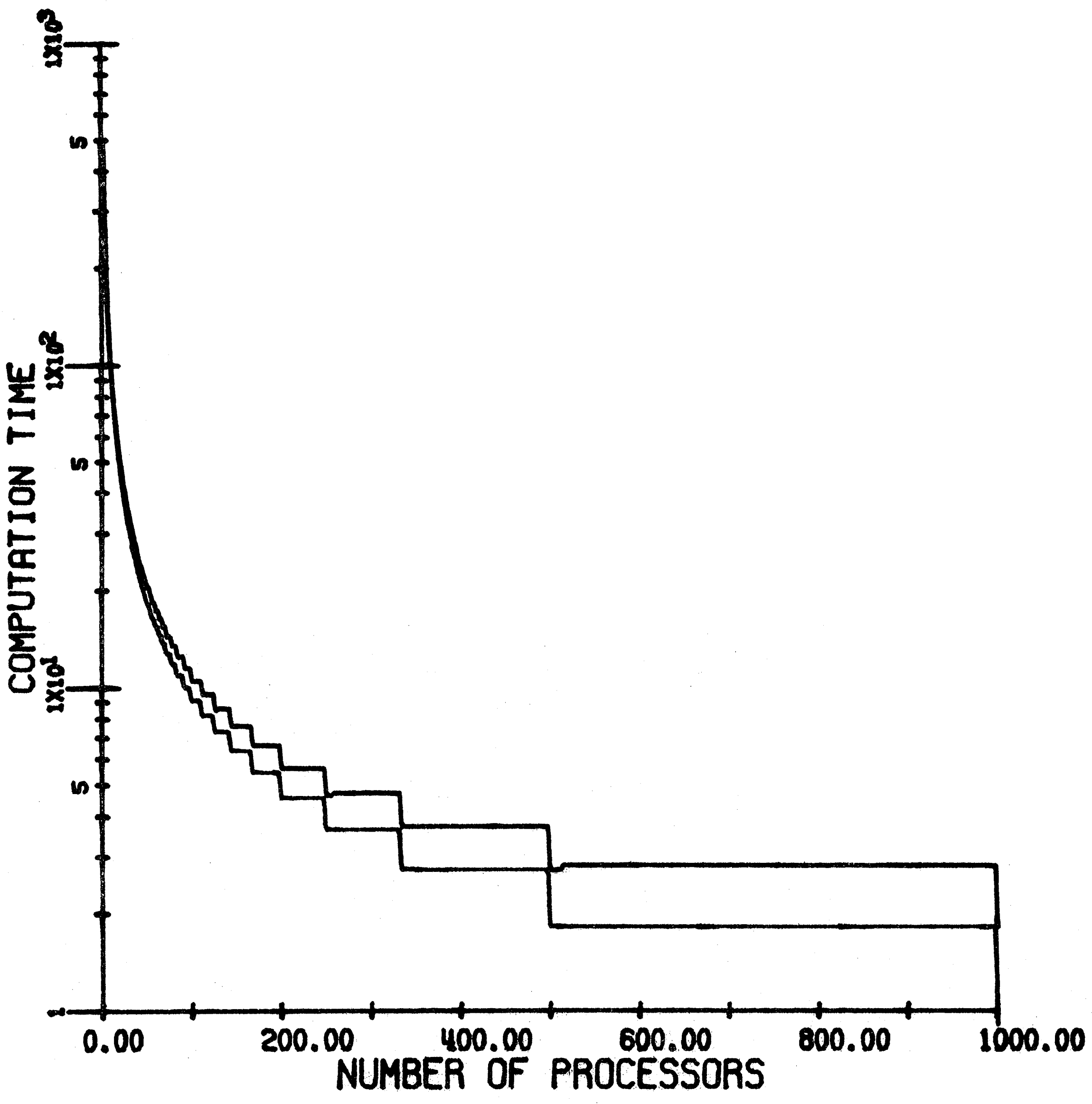

FIGURE 8

COMPUTATION TIME OF SIMPLE REPRODUCTIVE PLAN

POPULATION SIZE = 10000

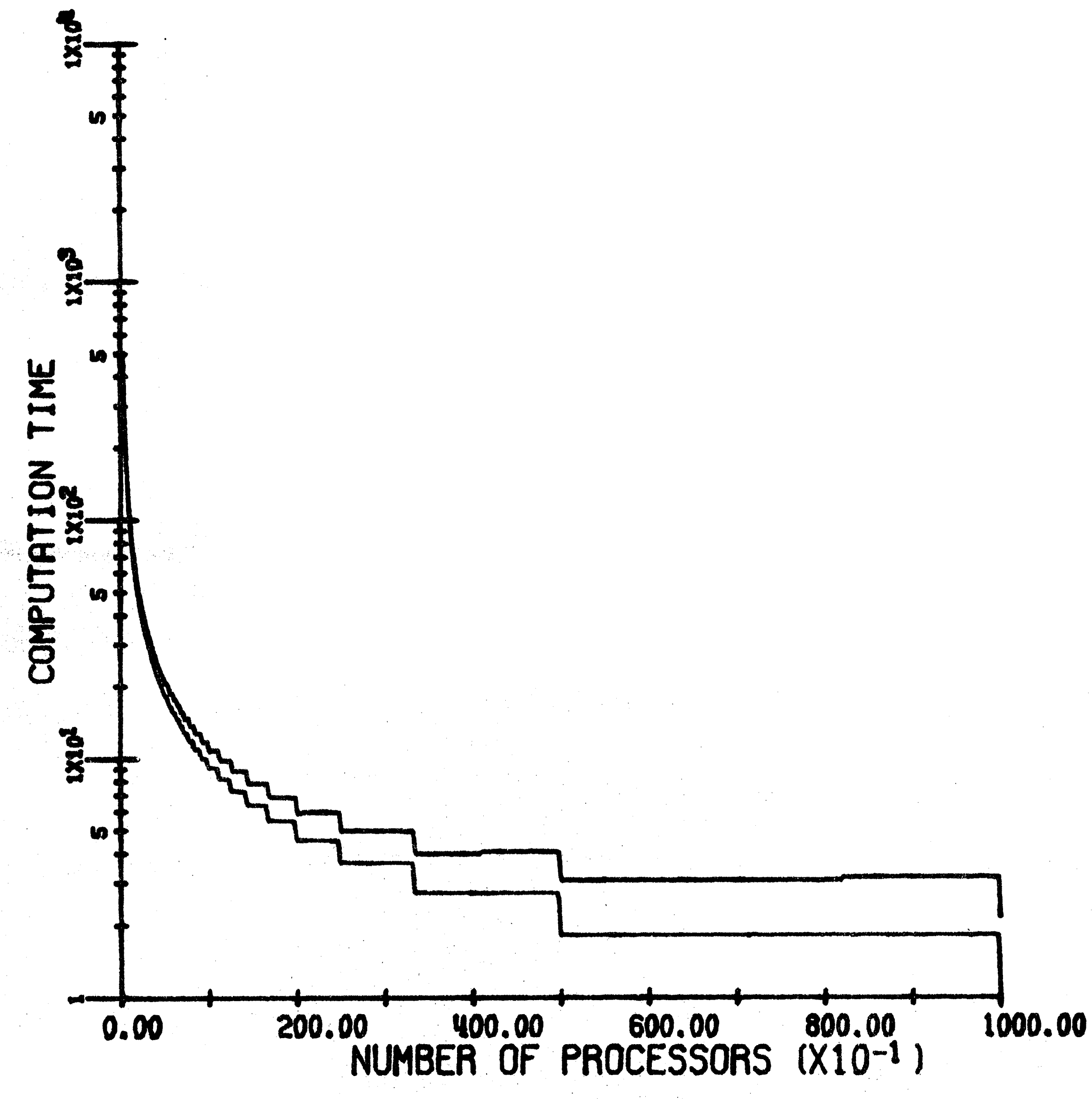

FIGURE 9

COMPUTATION TIME OF SIMPLE REPRODUCTIVE PLAN

POPULATION SIZE = 100000

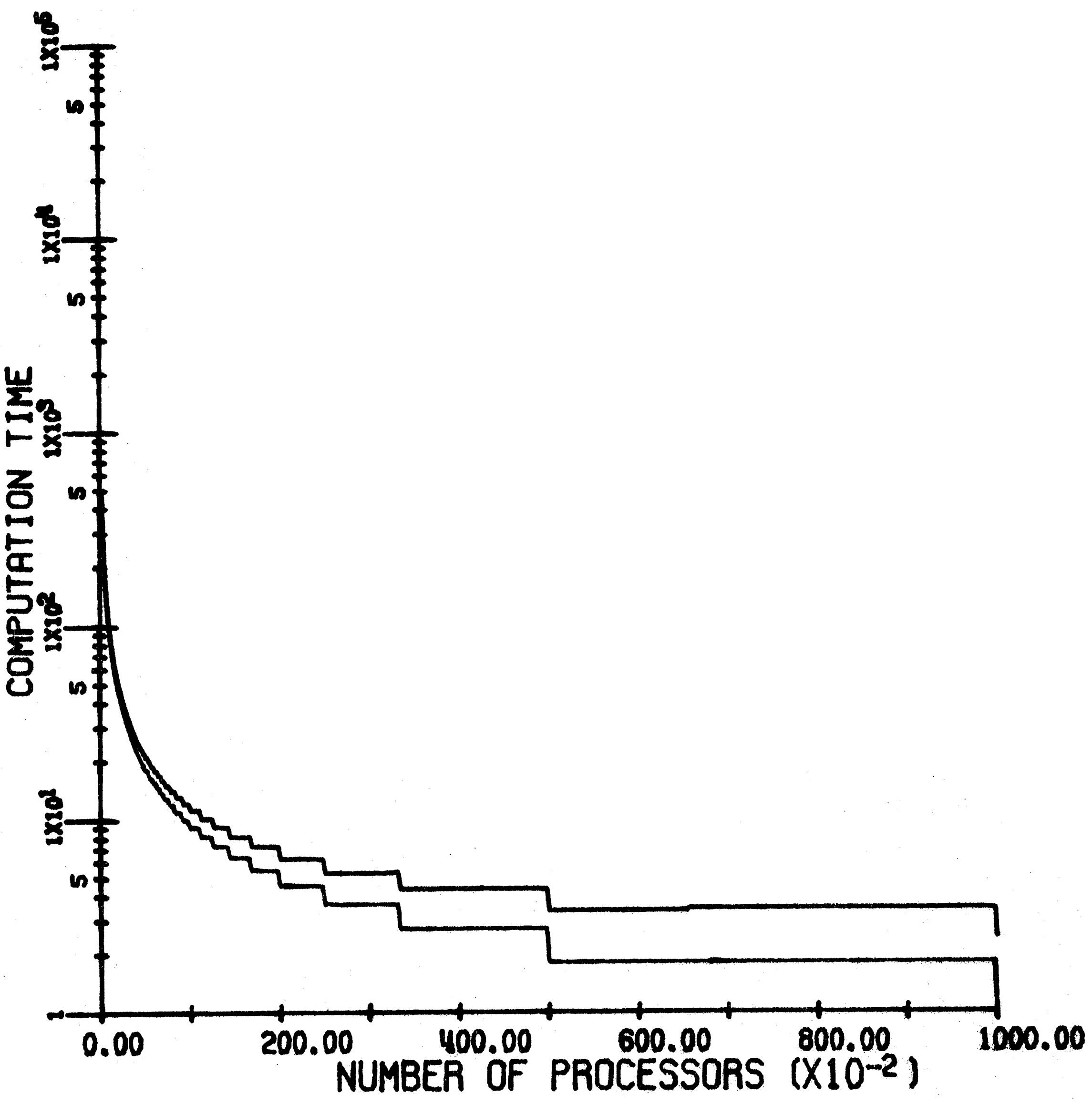

FIGURE10