PERFORMANCE ANALYSIS OF TWO CLASSES OF DATAFLOW COMPUTING SYSTEMS*

by

Robert Eugene Thomas

Technical Report #120

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

May 3, 1978

*Master of Science Thesis

Preface to the Thesis

One need hardly mention to those familiar with dataflow that research in the field has progressed very rapidly in recent years. Although I have tried to keep the results of this thesis as general as possible, the sparsity of dataflow architectural models which were available at the time the study was being done had its influence in limiting the scope of the thesis. This is particularly evident in the classification of dataflow machines into two types -- the preallocated and dynamically allocated machines. This classification has been found to be too rough and it is now felt that attempts at classification are too preliminary in this rapidly changing field.

As a result of this thesis and subsequent follow-up studies, it has been found that the dynamic allocation model (i.e., as described herein using allocation tokens) suffers from a serious lack of efficiency. For this reason, the allocation token model should not be taken as being advocated by members of the UCI Dataflow Architecture Project. Although limited space prevents description of our current (and ongoing) ideas on allocation, we have found that it is more appropriate to raise the "granularity" of allocation while perhaps retaining a smaller granularity of computational steps. A scheme such as this is actually a mixture of a preallocated and a dynamically allocated

machine (as described in the thesis) and we feel that it has most of the advantages of both.

The subject of allocation aside, I feel that some of the other ideas and results presented in the thesis are worthy of serious consideration.

3 May 1978

Bob Thomas

UNIVERSITY OF CALIFORNIA

Irvine

Performance Analysis of Two Classes of Dataflow Computing Systems

A thesis submitted in partial satisfaction of the requirements for the degree Master of Science in Information and Computer Science

by

Robert Eugene Thomas

Committee in charge:

Professor Kim P. Gostelow, Chairman

Professor Arvind

Professor Frederic M. Tonge

1978

CONTENTS

LIST OF FIGURES

LIST OF GRAPHS

Acknowledgments

From the scope of this study, it is apparent that a great deal of effort was required to bring it to completion. Since only a part of this effort was mine, I would like to acknowledge aid I received from other members of the UCI Dataflow Architecture Project. Wil Plouffe implemented an ID to base language compiler (with output suitable for the simulator) which greatly eased the production of dataflow programs. Kim Gostelow designed and coded a preliminary version of the simulator. (From this 20 page version, the simulator grew to 67 pages.) I also wish to thank Arvind for the suggestion to use a recursive description of computation time. Thanks to all of the above for many critical discussions.

A final thanks to the UCI Computing Facility for PDP-10 support and to the Swedish National Defense Research Institute for a SIMULA bug fix.

ABSTRACT OF THE THESIS

Performance Analysis of Two Classes of Dataflow Computing Systems

by

Robert Eugene Thomas
Master of Science in Information and Computer Science
University of California, Irvine, 1978
Professor Kim P. Gostelow, Chairman

Dataflow may be thought of as a language-oriented approach to the design and organization of computing machines. A particular reason for considering dataflow as a basis for the design of a machine is that it offers theoretically sound semantics for achieving highly parallel deterministic computation. Whether or not dataflow becomes a viable alternative to conventional (von Neumann) machine organization may well depend on the level of performance obtained by hardware based on dataflow principles. Since the fundamental concepts of dataflow are different from those of the von Neumann model, it is believed that conventional predictors of performance are unsuitable for dataflow machines. As a beginning for understanding dataflow performance, this thesis develops a method to analyze the execution of dataflow programs to determine how time is utilized. In order to substantiate the analytical

results a simulation model was constructed on which dataflow programs can be executed to observe relative time and resource utilization. The results of the study are a set of parameters and equations which roughly approximate the computation time required for certain types of dataflow programs on two different classes of dataflow architectures - the preallocated and the dynamically allocated machines.

1.0 INTRODUCTION

Advances in LSI technology have made it economically feasible to employ large numbers of identical logical units to perform computational tasks. The usual motivations for doing so include the desire to increase speed without relying on additional speed at the hardware level and to provide greater reliability through modular system design. Dataflow appears to offer theoretically sound programming language semantics for achieving deterministic asynchronous computation [D73, AG75, AG77]. However, many approaches may be taken to implement dataflow principles on various types of hardware logic presenting a significant challenge to machine designers. Although much has been written on the characteristics of conventional sequential programs, little is known about the behavior of dataflow programs.

Although dataflow program behavior could be studied on a purely abstract level, the inclusion of certain constraints implied by conceivable hardware implementations would make the results more interesting to designers. Several methods could be used to study dataflow program behavior including analysis of program graphs, simulation, and queueing theory. Since some implementation details were to be included, a combination of graph analysis and simulation were chosen as tools which could be used to study programs of interesting complexity. Some natural

questions about a proposed machine architecture include:

1. What is the expected performance in terms of time and resources required to compute various classes of algorithms?

2. What is the behavior when only a subset of the requested resources are available to a computation?

3. What are the implications of schemes for deadlock avoidance?

4. What is an appropriate balance of resources?

5. Do programs exibit a "locality of reference" which can be exploited?

This study addresses questions one through four above with a somewhat "architecture independent" point of view. This view requires that the study focus on dataflow behavior on an abstract machine rather than on a detailed simulation of one of the proposed dataflow architectures [DA77, DM75, AG76].

2.0 DATAFLOW PRINCIPLES OF OPERATION

Two basic principles of dataflow are [AGP76]:

1. An instruction executes when and only when all operands needed for that instruction become available, and
2. Instructions, at whatever level they might exist, are purely functional and produce no side effects.

Dataflow computation is therefore "data driven" as opposed to "control driven" as exemplified by the conventional von Neumann machine. A dataflow program may be represented as a directed graph with certain restrictions on interconnections between nodes. The nodes of the graph represent instructions and the directed arcs represent paths for operands. The value of an operand is contained within a packet of bits referred to as a token. Tokens may contain other information maintained for control, protection, etc. but for the most part this information is outside the scope of this study. To ensure determinancy and freedom from deadlock while still providing the full power of loops, conditionals, and procedure calls, dataflow programs are written in a language such as those described by Dennis [D73], by Weng [W75], or by Arvind, Gostelow, and Plouffe [AGP76]. A program written in a dataflow base language such as Dennis's dataflow language (DDF) is a graph representation of an algorithm (or procedure). This graph will be referred to as a "program graph" in this

paper. Figure (1) is an example of a program represented in DDF. When a program graph is executed (on any available interpreter), it defines a new graph which can be constructed by adding a new arc for each token output and a new node (called an "activity") for each instruction executed. This expanded graph, which will be referred to as an "execution graph", differs from the program graph exactly where arcs and nodes appeared in a loop (directed cycle) or conditional construct. Figure (2) is the execution graph of figure (1) with input equal to one and n equal to two. The execution graph is a complete acyclic description of the finished computation. The expansion of the graph can be accomplished dynamically (i.e. dependent only on the dynamic availability of operands and processors) by a straightforward and mechanical interpreter described by Arvind and Gostelow [AG75]. This interpreter is referred to as the "Unfolding Interpreter" and will be discussed later in this paper. It is important to realize that different inputs to the same program graph will in general produce different execution graphs.

3.0 AN ABSTRACT DATAFLOW MACHINE (ADM)

An abstract machine is convenient in order to isolate interesting behavior while disregarding distracting detail. Although the ADM cannot be realized in practice due to its unbounded nature, my goal is to show that it incorporates enough reality to remain interesting. The ADM consists of an unlimited collection of processing elements (PEs) and token collectors (TCs) drifting in an unlimited token storage pool. For the purpose of this paper, PEs will be regarded as abstract computing machines which can produce output as a function of inputs; the complexity of the function and the internal structure of PEs is of no concern. The machine starts execution by preloading the program definition into each PE and by introducing the appropriate input tokens. Token collectors gather together all of the tokens destined for a PE (which is going to execute a particular instance of an instruction) and when all are present, the TC seeks an idle PE where the tokens are deposited. From the input packet, the PE computes output tokens which in turn become operands for other PEs. PEs derive the logical destination address for these tokens from the preloaded program description. Whether all PEs are identical or of special types is of no concern since TCs could seek out specific PEs from control information written on tokens, or universal PEs could assume the

necessary identity from information in the program specification with the same effect.

3.1 Performance Analysis Of Dataflow Machines

The decision to build a dataflow machine may well depend upon its expected level of performance. Performance is a complex issue which has been studied concerning conventional machines since their inception. Although we might like to know what speedup might be achieved when an algorithm is executed in dataflow verses on a conventional machine, such a comparison is clearly dependent upon the specific machine used for comparison and the nature of the algorithm. For the proposed dataflow machine, it furthermore involves many assumptions about the speed of hardware, the utilization of available resources, and the effects of limitations of resources. The burden then falls on the machine designer to justify values for parameters which describe the performance of his machine. However, what are the parameters that appropriately describe dataflow machine performance? With a conventional von Neumann machine implemented with current technology, the speed of the memory is most often the bottleneck in compute-bound jobs. Although dataflow programs could be executed on machines similiar to conventional multiprocessor architectures, we are taking the approach

that decentralized control and memory will allow greater speed and increase machine modularity. For example, much of the data normally residing in conventional memory would be carried on tokens flowing on high bandwidth buses in a dataflow machine. Also, conventional machines are usually implemented with one or a small number of fast processors while dataflow machines may employ hundreds or thousands of relatively slow processors. For these reasons, one might expect that new parameters are needed to properly describe dataflow machines. Furthermore, comparisons of parameters with conventional machines may be misleading. Currently the only comparison known to be reliable is the total computation time required for specific algorithms. The first part of this study will be concerned with the development of parameters which describe aspects of dataflow machine performance.

3.1.1 Performance Analysis Of The ADM -

The approach to performance measurement taken here is to execute a program on a hypothetical dataflow machine and then to observe where time is consummed. The presentation will begin with an informal introduction and proceed with more formality as the machines become more complex. However, in most cases the results are verified not from mathematical origins but by experimental evidence from

simulation.

In the ADM time may be consummed in at least three ways (other overhead such as loading the program and input/output will be ignored):

1. Time for a token to proceed from the output of a PE to a TC.
2. Time for a packet of tokens to proceed to an idle PE.
3. Time for a PE to produce output after absorbing the input packet.

One might note that the situation could be simplified by allowing PEs to collect tokens directly thus doing away with TCs. The Simplified Abstract Dataflow Machine (SADM) is defined as this simplification to ADM. The discussion which follows primarily concerns SADM. However, it can be expanded to ADM in an obvious way. Time in a SADM may be consummed in at least two ways:

1. Time for a token to proceed from one PE to another. Let this time be denoted c_{ji} for the communication time from PE j to PE i.
2. Time for a PE to produce output after absorbing all input tokens. Let this time be denoted e_i for the execution time of PE i.

Note that time 2 for SADM could be considered to be the combination of times 2 and 3 for ADM (i.e. the definition

of e_i could be extended to include time 2 of ADM). Another time which could be defined is the time a PE waits for additional operands after receiving its first token. As we shall see below, this PE waiting time can be ignored in one definition of total computation time.

If one observes a dataflow program during execution (think of tokens flowing on arcs on the directed program graph), one would expect that of the numerous paths proceeding concurrently, one path could be traced which is the longest in terms of time. This is the intuitive notion of the "longest" path. Of course in the program graph, various arcs may be included in this trace more than once because of loops. Also different input may produce different traces and may change the length of the longest path as well as the path itself. However, if the program has already been executed and the instance of the acyclic execution graph recorded, the longest path is fixed. Here we may assume that during execution (say on the SADM), the times e_i and c_{ji} are recorded as part of the execution graph. (Of course, it may happen that more than one path in a particular execution graph may take exactly the same time to complete execution indicating that the longest path may not be unique even for an execution graph. However, in the discussion which follows, we may arbitrarily select one such path and ignore others even if their lengths differ.)

Let us suppose that the longest path has been

determined for an execution graph and that the labels along the longest path have been renamed so that c_1 is the first communication delay, e_1 is the first execution delay and so on, ending with an execution delay (for convenience). Let L_{Lp} be the number of arcs in the longest path. Then the total time for computation, t_c, is

$$t_c = \sum_{i=1}^{L_{Lp}} (c_i + e_i)$$

If we let $\bar{C}_{Lp}$, and $\bar{E}_{Lp}$ be the mean communication and execution delays respectively along the longest path then

$$t_c = L_{Lp} (\bar{C}_{Lp} + \bar{E}_{Lp}) \qquad (1)$$

Note that waiting time is not a factor in the equation because waiting time does not appear on the longest path. (Otherwise, the path on which we would be waiting becomes the new longest path.)

Equation (1) is of limited use because it involves detailed analysis of a program after it has executed. In order to avoid some bookkeeping, we would like to replace $\bar{C}_{Lp}$ and $\bar{E}_{Lp}$ with $\bar{C}$ and $\bar{E}$ respectively, where $\bar{C}$ and $\bar{E}$ are the mean communication and execution times for all arcs and nodes in the graph. This yields

$$t_c \approx L_{Lp} (\bar{C} + \bar{E}) \qquad (2)$$

Although equation (2) is inexact, it has value as a

predictive tool since the values for $\bar{C}$ and $\bar{E}$ can be measured easily. Since the longest path was not chosen randomly, the precision of equation (2) is unknown. Obviously under certain circumstances, such as a long sequence of time-consumming operations executing concurrently with numerous short lived operations, equation (2) is not accurate. However, if the execution and communication times are uniformly distributed in the execution graph, then equation (2) describes the behavior of the system more closely. Furthermore, equation (2) does not take into account various conditions which may affect computation time. One such factor is variation in communication and processing time due to competition for resources. For example, if a dataflow machine were being shared by a number of processes, one might expect that competition for resources would increase computation time for an individual process. Therefore, values of computation time can also be considered to form a distribution. As yet the nature of this distribution and its exact relation to equation (2) is unknown although it seems to be related to the variance in the C and E probability distributions, hereafter referred to simply as distributions.* Variance in dataflow graphs may have an

---

* A single capital letter will be used in the following to indicate a probability distribution; the value of the corresponding mean is represented with an overbar (e.g. C is a distribution with mean $\bar{C}$).

effect similiar to that in queueing systems. In simple queueing models it is well known that the mean time to pass through the system increases as the variance in the service distribution increases [K75]. As yet the quantitative effects of variance in dataflow graphs have not been analyzed. Another reason equation (2) is inexact as a predictive tool derives from the property that L_{Lp} may be different for different values of c_i and e_i on the same execution graph due to variations in communication and execution times for distinct executions of the same program and input data. Figure (3) shows a simple example of this possibility. The comments above on variance also apply to other equations below involving probability distributions. Although the equations are somewhat incomplete, I submit that they are useful since they indicate (collaborated by simulation results) that

1. Total computation time is a linear function of $\bar{C}$ and $\bar{E}$ derived from an execution graph.

2. Variance in total computation time is small (for a fixed computation) when variance in C and E is small.

3.1.2 Analysis Of A Dynamically Allocated Machine -

The ADM and SADM tacitly assume that all required resources (TCs and PEs) are available to a computation prior to the start of execution. Such machines will be referred to as preallocated machines. A more interesting class of machines capable of executing programs with more instructions than the number of physical PEs (or TCs) available are the dynamically allocated machines. These machines allocate PEs (or TCs) during execution based on some policy such as demand scheduling.

The discussion which follows concerning dynamically allocated machines is valid when exactly one of the inputs to an activity is determined as the allocation port. The production of a token destined for an allocation port initiates the allocation of a PE (or TC) for the intended activity. In this discussion, the allocation port will be marked on the dataflow program graph for ease of description. However, the results also apply to machines which dynamically determine the allocation port (e.g. those which initiate a resource request when the first token for an activity is produced). This discussion does not apply to allocation schemes which initiate a request for resources at some time other than when one of the tokens intended for the activity is produced, e.g. random allocation in time. (However, if somehow all activities are allocated resources before the last token destined for

them is produced* even for one operand instructions, then equation (2) applies in a meaningful way - i.e. the machine behaves as a preallocated machine for the purposes of time performance.)

The following is a general discussion of dataflow machines which use a logical addressing mechanism. In many cases, the results apply to simpler machines by disregarding or combining appropriate parameters. However, in other cases, changes in basic assumptions may require some modification in the method and results.

Let us suppose that dataflow programs are marked by a compiler such that for each instruction one of the input ports is marked as the allocation port according to an "allocation policy" as described above. These programs will be executed on a Dynamically Allocated SADM (DASADM) which operates as follows. When a token destined for the allocation port of an instruction is produced, it is marked as the allocation token for the intended activity. Allocation tokens released to the token pool search for an idle PE and allocate one when found. The PE then waits for other operands, if any, destined for the activity. When

---

*The critical time here may really be the time of token output plus some normal communication time to the vicinity of where the PE is being allocated since in logically addressed communication the PE need not be allocated prior to output of a token destined for it.

all operands are present, the PE produces output exactly the same as the SADM.

The performance analysis for DASADM contains a subtle difference from SADM when a probability distribution is applied to the communication delay. The reason is that token waiting time may occur in some communications. This waiting occurs when a non-allocation token is produced which is destined for a PE not yet allocated. Even though equation (2) applies to DASADM (i.e. it applies to the resulting execution graph), the desired goal is to identify parameters which are as close to hardware parameters as possible. It is therefore desirable to eliminate token waiting time from the distributions in the equation since waiting time is dependent on the allocation scheme. In order to develop an equation similiar to (2) but more appropriate for DASADM, it is necessary to alter slightly the notion of longest path as described below.

If an instruction has more than one operand and it is executed on DASADM, the allocation token would never be on the longest path. The reason of course is that the PE (or TC) must be allocated prior to the arrival of other operands. (Here I am assuming that the act of allocation includes absorption of the allocation token since the size of a token is relatively small.) Now if the allocation token is produced last, it would have been on the longest path but for the collection of other tokens which forced

the longest path to shift to one of the other operands. The communication delay for these operands necessarily contains some waiting time which is undesirable in the equation. Therefore, I will define a new path called the "critical path" which shares elements of the longest path but actually determines the longest path in situations where the allocation is late. Figure (4) is an arbitrary segment of an execution graph which will be used to recursively define the critical path and total computation time for DASADM. Let the function MAX(list of real numbers) return the maximum real number from the enclosed list. The communication time on the left of figure (4), c_{ji}, is directed to an allocation port (denoted by the letter "A") so it may include some communication time plus time to allocate a PE (these two could be separated, but doing so only complicates the current discussion). The communication times on the right are all non-allocation times which may include waiting time for allocation. Let w_{ki} be the time (possibly zero) that a non-allocation token spends waiting for its destination PE to be allocated. The communication delay in the execution graph for a non-allocation token then is

$$c_{ki} = w_{ki} + s_{ki} \qquad (3)$$

where s_{ki} is the time used to find the destination PE from the moment it was allocated. Note that s_{ki} is closer to a

hardware parameter than c_{ki}. Let t_i be the time at which the activity labeled i outputs its results. Following is an algorithmic description of the relation of t_i to its predecessors. If activity i has only one operand it must be an allocation token and

$$t_i = e_i + t_j + c_{ji} \qquad (4a)$$

For the case of more than one operand let m be the label of the previous activity such that

$$t_m + c_{mi} = MAX(t_k + c_{ki}, \ \ldots \ , t_n + c_{ni})$$

then

$$t_i = e_i + [\ \underline{IF}\ \text{(the PE is preallocated)}$$
$$\underline{THEN}\ t_m + c_{mi} \quad \text{(here } w_{mi} \text{ is defined to be 0)}$$
$$\underline{ELSE}\ t_j + c_{ji} + s_{mi}\ ] \qquad (4b)$$

(Terms with non-existent subscripts are to be ignored, e.g. t_0.) The predicate above is true when the activity is allocated prior to the arrival of the last token.* Note

---

*Equation (4b) is intentionally left informal so that the discussion may be more easily applied to a broader class of architectures. The basic premise is that some additional time overhead (i.e. s_{mi}) is incurred when the PE is not preallocated. Analysis of a specific architecture is required for more detailed definition of "preallocation" and "overhead". See Appendix I for an example definition of these terms for an implementation of DASADM.

that in (4a) and both parts of (4b), non-allocation token waiting time does not appear in the equation. This allows the approximation of total time with distributions which do not contain this waiting time.

Given an execution graph, equation (4) can be used to identify the critical path as follows. Compute t_i for each activity and with it associate the appropriate subscript (j or m) depending on which equation or THEN/ELSE part was chosen. For example, if the predicate in equation (4b) is true, the desired ordered pair for activity i is $(t_i = e_i + t_m + c_{mi}, m)$. Here m is the label of the preceeding activity which <u>determined</u> the longest path. The critical path is traced simply by starting with the last activity executed (maximum t_i) and following the chain of pointers (the second element of the ordered pair) back to the start of the graph (c_{0i}). Besides the number of arcs on the critical path, L_{cp}, three additional parameters are of interest. Let a_1 be the number of one operand activities along the critical path. In addition, let a_E be the number of "early" allocations defined by the number of times the <u>THEN</u> branch of (4b) is taken and correspondingly, a_L, is the number of "late" allocations from the <u>ELSE</u> branch. Note that

$$L_{cp} = a_1 + a_E + a_L$$

The above definition of the critical path allows

non-allocation token waiting time to be eliminated from the total time equation. Let $\bar{C}_T$ be the mean communication delay for non-allocation tokens between previously allocated PEs, $\bar{C}_A$ be the mean allocation token communication delay <u>including</u> waiting time to allocate, and $\bar{C}_S$ the mean search time for previously <u>waiting</u> tokens to find their destinations from the moment of allocation. In the simple case when $\bar{C}_T = \bar{C}_A = \bar{C}_S = \bar{C}$

$$t_c \approx L_{cp} (\bar{E} + \bar{C}) + a_L \bar{C} \qquad (5)$$

Equation (5) is derived from (4a) and (4b) by noting that an execution delay, e_i, and a communication delay (either c_{ji} or c_{mi}) appear for each two part segment of the critical path. In addition, another communication delay, s_{mi}, is required for each late allocation (a_L). The assumption in (5) is that all of the communication delays are taken from the same distribution, C.

On a machine which has a limited number of PEs (or TCs), the assumption above is unreasonable. In a manner similiar to the derivation of (5), a more general equation for dynamically allocated machines is

$$t_c \approx L_{cp} \bar{E} + a_1 \bar{C}_A + a_E \bar{C}_T + a_L (\bar{C}_A + \bar{C}_S) \qquad (6)$$

Note that equation (6) collapses to (5) when $\bar{C}_A = \bar{C}_T = \bar{C}_S$.

The derivation of (6) does not require unlimited resources. Moreover it is clear that the term, $\bar{C}_A$, can account for one effect of limited PEs (i.e. the mean waiting time to allocate a PE is included in $\bar{C}_A$). However, a possible effect of a limited communication medium, i.e. the waiting time for output access to a bus, has been hidden in $\bar{E}$. Figure (5) makes the bus waiting time, w_{bi}, more explicit. Since w_{bi} occurs in each segment of the critical path, it can be incorporated directly in (6) as

$$t_c \approx L_{cp} (\bar{E} + \bar{W}_B) + a_1 \bar{C}_A + a_E \bar{C}_T + a_L (\bar{C}_A + \bar{C}_S) \qquad (7)$$

where $\bar{E}$ has a more limited meaning. Of course $\bar{E}$ could be broken down into further terms such as code fetch, memory delays*, etc.

---

* Dennis has shown that a conventional memory organized as a "heap" can be utilized by dataflow programs where some of the functions performed by PEs are memory operations (i.e. select and append).

3.1.3 Effects Of Allocation Policy On Performance -

The terms a_L and a_E in equation (6) are directly dependent on the sequence of PE allocation in the execution of the dataflow program. Suppose that all of the terms in (6) are independent. (The equation is true for an execution graph even when terms are interdependent.) Assuming that $\bar{C}_A > \bar{C}_T$, then an obvious way to shorten execution time is to allocate such that $a_L = 0$ in the resulting execution graph.* In order to accomplish this in general, some activities must be allocated before any of their operands are produced. For example, suppose that two operands for a binary activity are produced at the same time. Then for the time that it takes to allocate a PE (or TC), the other token must wait for the allocation. Since the total time is dependent on the allocation time, the ELSE branch is taken in (4b) and a_L is incremented indicating that, in general, $a_L \neq 0$ even when the first token produced for an activity initiates allocation (i.e. demand scheduling).

In addition to having $a_L = 0$, if one-operand activities could also be allocated prior to the production

---

*Another obvious way is to shorten the critical path (and the longest path) which is intimately related to the amount of parallelism available in the task and the specific scheduling of resources for its execution. These subjects are beyond the scope of this study.

of that operand, then $a_E = L_{cp} = L_{Lp}$ and equation (6) would collapse to

$$t_c \approx L_{cp} (\bar{E} + \bar{C}_T)$$

which is essentially equation (2), showing the equivalence of $\bar{C}$ and $\bar{C}_T$ for preallocated machines.

3.1.4 Interdependence Of E And C Distributions -

This section is an attempt to relate previously stated results to realizable dataflow machines. Since many relationships are currently open questions, only intuitive arguments are presented. My hope is that the discussion will clarify differences between the abstract machine described above and realizable machines.

When $\bar{E}$, $\bar{C}_T$, $\bar{C}_A$, and $\bar{C}_S$ are derived from an execution graph, it is clear that the values are independent of one another (i.e. none of the values is used to compute another). However, in a realizable machine, a condition that fundamentally affects only one distribution may have compound effects in other distributions. Waiting time (such as the waiting time to allocate a PE that is included in $\bar{C}_A$) is particulary sensitive to other machine parameters. As an example, imagine a machine composed of a small number of very fast PEs such that the machine is effectively limited by communication delays. Furthermore,

assume that the machine has a sophisticated allocation scheme which manages to keep the PEs productive a large percentage of the time such that the addition of more PEs would not reduce the execution time of a certain group of programs. Now if the speed of the PEs became slower (larger $\bar{E}$), a larger number of PEs might be utilized. However, given the same number of PEs, the demand for increased processing could easily cause an increase in waiting time to allocate PEs. Therefore in this example an increase in $\bar{E}$ causes an increase in $\bar{C}_A$. An examination of equation (7) indicates that computation time will not increase linearly with $\bar{E}$ under these conditions unless $\bar{C}_A$ is a linear function of $\bar{E}$. Other examples of non-linear behavior which do not involve waiting time also can be surmised. Suppose that a decrease in $\bar{E}$ causes a computation to grow quickly in space thus increasing the mean distance between PEs and the time for inter-communication. In this situation a decrease in $\bar{E}$ causes an increase in $\bar{C}_T$ and possible non-linear changes in computation time.

Although the analysis presented in the previous section does not account for interrelationships between distributions, I feel that the results serve as a basis for further study in this area, and are particularly useful in intuitively understanding dataflow progam behavior.

4.0 SIMULATION OF A DYNAMICALLY ALLOCATED DATAFLOW MACHINE

Section three above discussed the relation of total computation time to the mean of a number of probability distributions for communication and execution delays. However, the dependence of each of these means on various factors such as resource limitations, spatial distribution of PEs (an obvious attribute of physical machines), and class of algorithm was not included. In addition, the amount of resources required to carry out a computation was unspecified. In many instances simulation is an appropriate tool to use when other types of analysis are intractable in complex situations. In this section a description and analysis of results concerning some of the above questions will be given for the Irvine Dataflow Simulator. The simulator is coded in SIMULA, a discrete simulation language [DN66].

4.1 Specification Of The Simulated Machine

Although a general purpose simulator is desired, the following items must be specified in detail in order to simulate a dataflow machine even at the level of the DASADM:

1. The base language in which dataflow algorithms are expressed.

2. The interpreter which executes the semantics of the base language.

3. The addressing mechanism which directs tokens destined for a particular activity to a processor.

It should be noted that the results given in the remainder of this paper may be dependent on choices made regarding the above items. These choices are described below.

4.1.1 The Base Language -

The simulator actually accepts a number of base languages, but only the most relevant will be described here. One of these is the Dennis Data Flow language (DDF) [D73]. Figure (1) is an example of a DDF program. (In the remainder of this section, the reader is assumed to be familiar with DDF.) A second base language (referred to simply as BL1) is identical to DDF except that a new switch actor is included, and in certain circumstances a merge actor may be replaced simply by joining two lines. (When constrained by the semantics of a higher level language (e.g. ID [AGP76]) and executed under the unfolding interpreter, joining two lines can be shown to result in determinate programs [AGP77].) The firing rules and a sample code segment for a switch are given in figures (6a) and (6b). A third base language (BL2) is very similiar to BL1 except that the switch may have more than one set of value input/output lines and not all values on input lines

need be present for firing. Figures (7a) and (7b) illustrate this "vectored" switch. For all switches, if an output for V_i is not defined for the boolean value input, then V_i is simply destroyed.

The vectored switch is distinguished from all other dataflow operators in DDF and BL1 since it essentially fires n times for one boolean input. In the simulator implementation of this switch, a PE remains allocated until all n + 1 inputs have been absorbed and all defined outputs produced. Note that the vectored switch produces the same results as n unary switches supplied with n copies of the boolean value if the boolean value is an allocation token for the vectored switch.

Whenever a loop is implemented with two lines joined together in BL1 or BL2, the possibility exists that successive tokens which are initiating separate instances of the loop could be confused with tokens internal to previous instances of the loop. This is the same problem that exists for procedure applications and it is solved in a similiar manner - i.e. by changing the context for each instance of a loop. A particular instance of a loop will be referred to as a "loop domain". Loop domains have a context distinct from the enclosing program as well as from each other. The change of context can be implemented with colored tokens [D73] or by a stacking mechanism and is not of concern here. Although other definitions could be

given, the operating rules for loop domains in the simulator are exactly the same as procedure application - i.e. all inputs to the domain must be available before any part of the loop begins execution and the loop must terminate before any output is produced. Figure (8) illustrates a general loop construct in BL1 or BL2.

4.1.2 The Dataflow Interpreter -

The choice of interpreter is an important one since the operation of the interpretation mechanism can impose sequentiality in dataflow programs. Dennis has proposed a "feedback" interpreter for dataflow schemas [D73]. It is our feeling that the feedback interpreter places an unnecessary constraint on the potential asynchrony of dataflow programs. Arvind and Gostelow have proven [AG77] that the "unfolding" interpreter produces the same results as the feedback interpreter except in certain circumstances when the unfolding interpreter is more defined (produces more results) in the sense of the theory of fixpoint semantics [MA74]. For this reason, the unfolding interpreter was chosen as the basis for implementation of a DASADM.

Although the unfolding interpreter is described in detail in [AG75], the basic idea is that each activity is tagged with a distinct identifier referred to as an

"activity name". An activity name is of the form u.p.s.i where u is an outer context, p is the procedure in which this activity is defined, s is the statement number, and i is the ith instance of the statement. (The prefix u.p is actually the context for all statements in p but these are separated for the purpose of exposition.) Since activities have distinct identifiers, these can be used for addressing purposes by including the u.p.s.i identifier tag on each token. This tag is extended to include one more field q, where q is the destination port number of the operand. All tokens destined for a particular activity are given its activity name (u.p.s.i) and the appropriate port number, q. Mechanical rules for generating these names are given in [AG75]. With the unfolding interpreter it is possible to have distinct instances of the same statement, loop, or procedure active at the same time.

4.1.3 The Addressing Mechanism -

Specification of the addressing mechanism and the associated bus structure of the dataflow machine is necessary because different structures may impose different restrictions on concurrency. In addition, the choice to implement a dynamically allocated machine (and a particular method of allocation) verses a static machine has implications on the bus structure (i.e. it introduces the

possibility of waiting time at least for non-allocation tokens). The activity name required by the unfolding interpreter provides a convenient logical address for PEs. In order to remain as close as possible to the definition of DASADM (to avoid being locked to a specific architecture) the following bus structure is implemented in the simulator (the distributions mentioned below are computed from input parameters):

1. The activity names of all allocated PEs are maintained in a set.

2. When a token is ready to be put on the bus, the status of the bus is checked. If the bus is full (the size is defined by an input parameter), the PE must wait until an empty spot is available ; otherwise the token is put on the bus.

3. When a non-allocation token is put on the bus, it is given a communication delay from the C_T distribution. When this bus delay has elapsed, the set of PEs is searched to find its destination. If the destination is present, the token is absorbed by the destination PE; otherwise the token is put in a waiting set.

4. On each allocation of a PE the set of waiting tokens is searched for tokens destined for this activity. Each token that is found is given a communication delay from

the C_S distribution.

5. When the communication delay for a previously waiting token has elapsed, the token is absorbed by the destination PE.

6. A token destined for an allocation port (specified on the program graph), is marked as an allocation token. These tokens are given communication delays from the C_A distribution. On the expiration of this delay, if a free PE is available (the total number of PEs is an input parameter) then a PE is allocated and the token is absorbed. Otherwise, the token goes into a set waiting for allocation. Each time a PE becomes free this set is checked. If the set is not empty, the first element is given a new communication delay from C_A.

The bus structure above is implemented so that the simulation of bus conflicts is minimal (i.e. conflicts arise only when the bus is full since the tokens have no physical position) and the time to search the sets and other bookkeeping is not counted as part of the total computation time for the dataflow program being executed. Although the above structure with this constraint would be impossible to implement in hardware, I will attempt to show that it is useful for studying the behavior of the class of

dynamically allocated dataflow machines.

4.2 Simulator Input

The input to the simulator consists of the following:

1. A linear representation of the dataflow algorithm to be executed: Sample dataflow programs were chosen from the following algorithms where the ".extension" indicates the implementation base language: QUICKSORT.BL1, QUICKSORT.BL2, MATRIX MULTIPLICATION.BL1, GAUSS-ELIMINATION.BL1 (for simultaneous linear equations), GAUSS-SEIDEL.BL1 (approximation for simultaneous linear equations), and NEWTON-RAPHSON.DDF (root finder);

2. The values for $\bar{C}_A$, $\bar{C}_T$, $\bar{C}_S$, and $\bar{E}$: Real numbers for communication and execution delays are drawn from a probability distribution with specified means using a random number generator and various distribution functions provided by SIMULA (or from scientific packages such as the International Mathematical and Statistical Library). The seed for the random generator is saved at the end of each simulation and is restored at the beginning of each run since the repetitive use of a particular sequence of values may color the results. Each run of the simulator is

therefore a Monte Carlo experiment and many runs are required to verify results. Experiments are conducted such that sufficient values are drawn from each distribution so that the resulting mean is within ten per cent of the desired value. The following distributions were used in this study: uniform, normal, exponential, hyper-exponential (using two exponential stages in parallel), and determinate (constant value) distribution;

3. The values for the number of PEs and the total bus capacity;

4. Various other input parameters described below.

Due to the addressing mechanism described above the values input for $\bar{C}_A$, $\bar{C}_T$, $\bar{C}_S$, and $\bar{E}$ produce execution graphs with corresponding <u>independent</u> (from each other) distributions with approximately the desired mean when <u>unlimited</u> resources are available. However, when resources are <u>limited</u>, the execution graph distributions in general differ from input distributions due to reassignment of resources, bus conflicts, etc. In the following, the word "input" (e.g. input $\bar{C}_A$) will be used to differentiate between the two types of distributions. Note that the equations of section three directly apply only to execution graph distributions.

4.3 Simulator Output

The output of the simulator is described in some detail not only to clarify this study but also to provide potential metrics for others contemplating dataflow simulation or evaluating hardware performance.

Let A be the number of PEs allocated at any moment in time. Then the value of A verses time is a discrete (experimental) function which changes each time a PE is allocated or deallocated. These state changes are noted by the simulator and the value of A is integrated (summed) to compute the area under the function. This area is defined to be the allocated processor resource (APR) in PE-seconds.* APR may be thought of as the cost of processing much like CPU-seconds is used in conventional machines. In a similiar manner, let E be the number of PEs executing** at any moment in time, and let EPR be the

---

*The time unit is arbitrary and irrelevant since the discussion following is based on relative changes in these values. I will use seconds for its mnemonic value.

**In this study the term executing may include a wide range of actions such as code fetch and performing arithmetic/memory operations which may furthermore include waiting time for memory conflicts. However, memory conflicts are not modeled in this simulator, although in the model described memory delays could be considered as part of the E distribution.

executing processor resource in PE-seconds. Since EPR does not include token or bus waiting time and since $\bar{E}$ is an input parameter, EPR is a close measure of computation effort (ignoring memory conflicts) which remains constant for a specific computation independent of allocation policy, limited resources, etc. Since EPR is always part of APR, the EPR/APR ratio is one measure of processing efficiency. A related metric is the wasted processor resource which is defined as APR - EPR. However, both of these measures have limited value unless a detailed architecture and optimization goals are specified. A third experimental function is the total number of tokens on the bus, T, over time and its integral, token bus resources (TBR) in token-seconds. In addition, the mean value for each resource is computed by dividing total resource usage by the total computation time. For example, the mean number of PEs allocated is given by APR/total time.

Data from the three experimental functions above may be graphed for each run of the simulator. As can be seen in graphs (1) and (2), these functions display the peaks and valleys of resource utilization and (in this example) the expansion and contraction of resource utilization in a simple matrix multiplication. In order to limit the amount of data output for these graphs, the simulator "averages" the data over a time interval specified as an input parameter. Therefore, in certain cases, sharp peaks and

valleys may be lost on the graph. However, this has no effect on the actual computed resource measurements since these values are computed without averaging. Note that additional averaging may be incurred for graphs when the data is further compressed to conform to the discrete constraints of a printer.

Other metrics for dataflow performance include the experimental distributions for PE waiting time and token bus time. It is important to note that in general these distributions are not input but are experimentally determined. Experimental distributions generally include waiting time of some sort and will be denoted in the following with the subscript X appended to previously defined distributions of a similiar type. The probability mass function (PMF) is a convenient method to graph these distributions. (The probability mass function is the discrete case of the more familiar probability distribution function.) PE waiting time is defined as the time spent from allocation (time at which the allocation token was absorbed by a PE) until the last token required by that same PE is absorbed. The PMF gives the (experimental) probability that the waiting time was less than or equal to a given time along the x axis. Also of interest is the first derivative of the PMF which is referred to as the mass density function (mdf). Graph (3) is a sample PMF and mdf for PE waiting time. Although these functions are

plotted together, the probability scale along the y axis is applicable only to the PMF.

PMF and mdf are also suitable functions for graphing token bus time. However, depending on the architecture, it is generally useful to separate bus times similiar to the distributions described in section three. In particular, the allocation token communication time distribution, C_{AX} (which is the same as C_A with unlimited resources since there is no waiting time to allocate) and the non-allocation token communication distribution (including waiting time), C_{NX}, were separated in this study since it is expected that these distributions will be quite different in practice. Graphs (4) and (5) are examples of token bus time distributions for allocation and non-allocation tokens respectively. Also of interest would be the waiting time distribution to enter the bus, W_{BX} (of equation (7)), which as yet has not been included in simulator output.

As part of the graphs for the experimental distributions described above, standard statistical parameters are produced for each distribution including the mean, variance, standard deviation, and coefficient of variation.

In addition to the simulated time to execute each dataflow procedure, various other values are printed for each run of the simulator such as the total number of PEs

allocated and total tokens transmitted. Also printed are the maximum number of PEs allocated and executing at one time, and the maximum number of tokens concurrently on the bus. Many of these values are useful mainly for cross checking simulation results.

4.4 Experimental Time And Resource Utilization

In this section, verification of analytic results presented in section three will be given. In addition, requirements for resources determined by simulation will be presented.

Recall from equation (6) that the computation time for a dataflow program is approximately equal to a linear combination of the means of various distributions for execution and communication delays. Since one of the goals of this study was to determine the relative effects of execution and communication delays, most runs of the simulator were made with input distributions $C_T = C_S = C_A = C$. Therefore, equation (5) is an appropriate description of time performance for these runs. The experimental method in this section involves holding the input value for $\bar{E}$ (or $\bar{C}$) constant for various values of $\bar{C}$ (or $\bar{E}$) and plotting total simulated computation time, allocated processor resource (APR), executing processor resource (EPR), and token bus resource (TBR) verses $\bar{C}$ (or $\bar{E}$).

4.4.1 Unlimited Resources -

Graphs (6) through (10) indicate the results when unlimited PEs and bus capacity are available to a computation. Note that not only total computation time but also APR, EPR, and TBR are approximately linear for various base languages, algorithms, and probability distribution functions for C and E indicating that APR and TBR may be thought of as linear functions of computation time. Furthermore, when C and E are determinate (variance = 0, graph (6)), the graph is virtually straight indicating that the terms L_{cp} and a_L in equation (5) are essentially constant for the simulated dataflow program. This is not always the case since in obscure instances the sequence of allocation can depend on the relative values of $\bar{C}$ and $\bar{E}$ as illustrated in figure (9). However, note that this effect is caused by a "mixed" allocation policy and the effect is relatively small. When the C and E distributions are not determinate, graphs (7) through (10) indicate that the average behavior is linear and deviations seem to be proportional to the amount of variance in the C and E distributions.

Graphs (11) and (12) also illustrate linear results when the input distributions for $\bar{C}_A$ and $\bar{C}_T$ are separated. In this case equation (6) is applicable to the total computation time. As might be expected, APR, EPR, and TBR are also linear with $\bar{C}_A$ and $\bar{C}_T$.

4.4.2 Limited PEs, Unlimited Bus -

The derivations of equations (1) through (7) were independent of the resources available to a computation. Note that even if deadlock occurs (perhaps from lack of resources), the equations hold for an execution graph by producing an infinite computation time. However, more interesting results are obtained when deadlock (caused by lack of PEs (or TCs)) is avoided by reassigning PEs to other activities. One scheme to prevent deadlock on a DASADM when as little as one PE is available is presented in [AG76].* The scheme involves deallocation of a PE after it has waited a set amount of time and all operands have not arrived. Let this time be MAXWAIT after which the PE outputs any tokens it may have absorbed and then becomes available for another activity. To prevent the same activity from being reallocated too soon, an "activation" time referred to as TIMEIN is written on the allocation token. After returning to the bus, the allocation token may not begin to allocate a PE until TIMEIN has elapsed. Although the values of these two parameters could be different for each activity, the simulator currently uses

---

*This scheme was chosen because of its simplicity and ease of implementation. It is not suggested that the unmodified scheme is appropriate for dataflow machine implementation.

the same value for all activities. Note that restrictions must be placed on the values of TIMEIN and MAXWAIT to ensure deadlock free operation as follows:

1. The value of TIMEIN must be finite in most cases.*
2. The value of MAXWAIT must be finite (with limited resources) and yet long enough so that eventually all of the operands for each activity will have time to reach the PE before deallocation.

Note that the previous description of DASADM is equivalent to setting MAXWAIT to infinity and TIMEIN to zero and is guaranteed to be deadlock free only when unlimited PEs are available.

The presence of this deallocation scheme has side effects on resource and time utilization even when unlimited PEs are available. Simulation results indicate that the scheme generally increases computation time and bus resources while decreasing allocated PE resources (APR). In addition, the input distribution C_A is no longer in direct correspondence to the execution graph distribution for C_A since the final allocation of a PE may

---

*Some activities need never execute for a computation to terminate. For example a gate actor in DDF which happens to absorb every value input. However, execution of all activities may be desireable for other reasons such as adherence to the "well behaved" property [D73].

have proceeded through several deallocation and reallocation sequences. However, graphs (13) and (14) indicate that computation time, EPR, APR, and TBR remain linear with input distributions E and C when deallocation is present with unlimited resources. In addition, comparison of graphs (10) and (14) indicates that EPR remained the same, APR decreased, and TBR and computation time increased when the dellocation scheme was used and no other changes were made.

The mean of the maximum number of PEs allocated at once for the nine runs plotted on graph (14) was twenty-six PEs (i.e. the mean number of PEs which this computation consummed at its highest point). Graphs (14), (15), and (16) are a family of plots for a machine with unlimited PEs, a maximum of seven PEs, and a maximum of three PEs respectively. Again the results indicate that computation time, EPR, APR, and TBR are linear with the input distribution for $\bar{E}$. Note that these results do not imply that computation time degrades linearly with the number of PEs available. On the contrary, graph (17) indicates the result of limiting PEs from the maximum usable (seventy in a larger matrix multiplication) to approximately one tenth this number. When viewing graph (17), it is important to remember that the abstract machine being simulated imposes no penalties (say in extended communication time) when unlimited numbers of processors are participating in the

computation. It is clear that a realizable dataflow machine would not have this property for large computations and that the optimal number (for fastest execution) of PEs for a computation could be less than the maximum allowed by the dataflow program graph. Determining the optimal number of PEs is architecture dependent and is not pursued in this study. However, experience with this simulation model and results such as graph (17) as well as for other algorithms, indicates that from twenty-five to forty per cent of the maximum number of PEs seems to produce good performance without undue waste of processor resources.

Graphs (18) and (19) are sample graphs of resource utilization with the number of processors limited to seven. These graphs may be roughly compared with graphs (1) and (2) which are examples with unlimited resources.

4.4.3 Limited Bus, Unlimited PEs -

Experiments with a limited bus were hindered due to lack of a simple mechanism to prevent deadlock. However, graph (20) illustrates the linear results when the total bus capacity was reduced (from a maximum mean utilized of sixty-two tokens when unlimited) to a maximum of forty tokens. In addition, graph (21) indicates the results when bus capacity is reduced when other parameters are constant. However, this graph is not as revealing as graph (17) since

the computation deadlocks when the bus capacity is less than thirty-five tokens.

4.4.4 Limited Bus, Limited PEs -

Although results with these constraints are most interesting, in my opinion a specific architecture is required in order to study this area.

4.5 Experimental Effects Of Allocation Policy

In this section, the effect of changing the allocation port markings on a program graph will be presented (more dynamic allocation policies are not discussed). The method of selecting the allocation port (i.e. the allocation policy) may induce variations in time and resources required by a computation. All allocation policies are restricted by the following: each and every activity must be allocated by exactly one allocation token. There appears to be an unbounded number of such allocation policies since each permutation of the allocation port assignment for an entire graph could be thought of as the result of a particular allocation policy. One interesting group of these policies attempts to minimize the value of a_L from equation (6) for all execution graphs of dataflow programs. Implementation of this policy involves graph

search of the dataflow program and assumptions about the probable execution and communication times of the implementation architecture; hence automatic implementation of this policy was not attempted. However, two simple allocation policies were implemented for automatic production of dataflow programs. The first policy is implemented by a compiler which translates from the Irvine Dataflow (ID) language to BL1 and always selects port number one as the allocation port. The second policy randomly selects one port from the input ports present for each instruction of a dataflow program graph. In effect the random policy is a generic policy which randomly generates instances of dataflow programs equivalent to those produced by particular allocation policies. The program which implements this policy is initialized so that each execution in general produces different output for the same dataflow algorithm input. Graphs (22) and (23) indicate the resources utilized by the same algorithm with three different allocation policies. Note that the approximately straight lines with increasing $\bar{E}$ (or $\bar{C}$) is a result of equation (5) and the previously stated experimental result that TBR and APR are linear functions of computation time. Graphs (24) and (25) are the corresponding computation times required by the three allocation policies which in this example are relatively the same.

4.6 Experimental Effects Of Variance In C

Some effects of variance in the C and E distributions were alluded to in section 3.1.1 . Graph (26) illustrates the effect of changing the variance in the input distribution C while $\bar{C}$ remains constant. The coefficient of variation (the standard deviation divided by the mean of a distribution) is a convenient normalizing parameter for variance. Increased variance in the input distributions was introduced by lowering the probability of selecting the first stage of a parallel two stage exponential system (where the mean service time of each stage is also a function of the probability). When the probability is greater than zero and less than one half, this configuration models a hyper-exponential distribution in which the variance can be made unbounded (for an unbounded number of sample points). For the sample program plotted in graph (26), it is clear that computation time (and variance in t_c) increases as variance in the communication time distribution increases even though the mean communication time is constant.

4.7 Substantiation Of Analytical Results

In preceding sections, both analysis of dataflow program graphs and simulation data indicated that

computation time for a dataflow program is a linear function of the means of the E and C distributions. In this section, a brief comparison of results from the two methods is presented.

Since most of the simulation results were obtained with $\bar{C}_A = \bar{C}_T = \bar{C}_S$, it is convenient to compare these results to those predicted by equation (5) from section three which is reproduced below:

$$t_c \approx L_{cp} (\bar{E} + \bar{C}) + a_L \bar{C}$$

Rewriting equation (5) yields

$$t_c \approx L_{cp} \bar{E} + (L_{cp} + a_L) \bar{C} \qquad (8)$$

It is easily seen from this equation that if $\bar{C}$ is held constant (assuming that L_{cp} and a_L are also constant), $\bar{E}$ is an independent variable which defines a linear function with slope = L_{cp} and intercept = $(L_{cp} + a_L) \bar{C}$. Similiarly, if $\bar{E}$ is held constant, $\bar{C}$ becomes the independent variable and the slope = $(L_{cp} + a_L)$ with intercept = $L_{cp} \bar{E}$. Once the slope and intercept are determined, the values of L_{cp} and a_L may be readily determined by algebra. The values of L_{cp} and a_L may also be computed from two boundary cases if the dataflow computation time for these cases can be determined. The structure of the simulator accomodates these conditions and the results may be used by substitution into the following

equations derived from equation (8)

$$t_c \approx L_{cp} \bar{E} \qquad \text{for } \bar{C} = 0$$

$$t_c \approx (L_{cp} + a_L) \bar{C} \qquad \text{for } \bar{E} = 0$$

In order to substantiate the results of the simulator, the dataflow program graph of matrix multiplication was analyzed for the longest path according to the rules of the unfolding interpreter (the definition of path length used here is simply the number of dataflow instructions in the path). For matrix multiplication the length of the longest instruction path was found to be

$$L_{LIP} = 4 (k + m + n) + 25$$

for a matrix of size k by m multiplied by a matrix of size m by n. Therefore, for the two by three matrix multiplied by a three by two matrix used in the experiment below, the longest instruction path equals fifty-three. This program with three different allocation policies was run on the simulator for the boundary conditions and with $\bar{E}$ and $\bar{C}$ respectively as the independent variable. Port 1 indicates that port number one was the allocation port for all instructions, and RA2 and RA3 indicate two different instances of random allocation. A summary table of the computed values of L_{cp} and a_L is presented in Appendix II. In the cases when $\bar{E}$ and $\bar{C}$ were independent variables,

nine points were taken (for each) and a least squares regression was used to obtain the slope and intercept with each point given equal weight. As indicated by the table, perfect agreement (note that this data was collected with deterministic E and C distributions) with analytical results for critical path length were obtained in the case of allocation to port one. Although the program graph was not analyzed to obtain the value of a_L (a laborious and error prone task), the data appears to be consistent. Also deviations in values for RA2 and RA3 are surmised to be caused by path length changes of the type described by figure (9).

5.0 SUGGESTIONS FOR FURTHER RESEARCH

This study has focused primarily on dataflow program behavior at an abstract level rather than in terms of a specific architecture. Additional study along the same lines may include analysis of the effects of variance in dataflow execution graph distributions. However, there are many open questions concerning dataflow machines and program behavior which seem to be dependent on specific machine architectures. The following is a partial list of performance questions which this study has raised:

1. How will memory delays (or conflicts) affect dataflow performance?
2. What are the interrelationships between the distributions E, C_A, C_T, and C_S?
3. What are the relationships between these distributions and hardware parameters?
4. Can heuristics be developed which approximate an optimum allocation policy?
5. Can decentralized scheduling policies be developed which lessen the impact of limited resources?

6.0 CONCLUSIONS

The performance analysis of computing machines is a complex matter involving relationships between hardware parameters, hardware and software organization, and the nature of the job mix. It is unlikely that complete knowledge of these relationships will ever be known considering the increasing complexity of computing systems and the rapid advance of computer technology. However, detailed information on performance is required by machine designers who must carefully balance tradeoffs in machine cost and performance. This study is an attempt to provide machine designers with parameters and relationships which describe the performance of dataflow machines. Since dataflow machines are fundamentally different in operation from conventional von Neumann machines, the study of dataflow performance is just beginning.

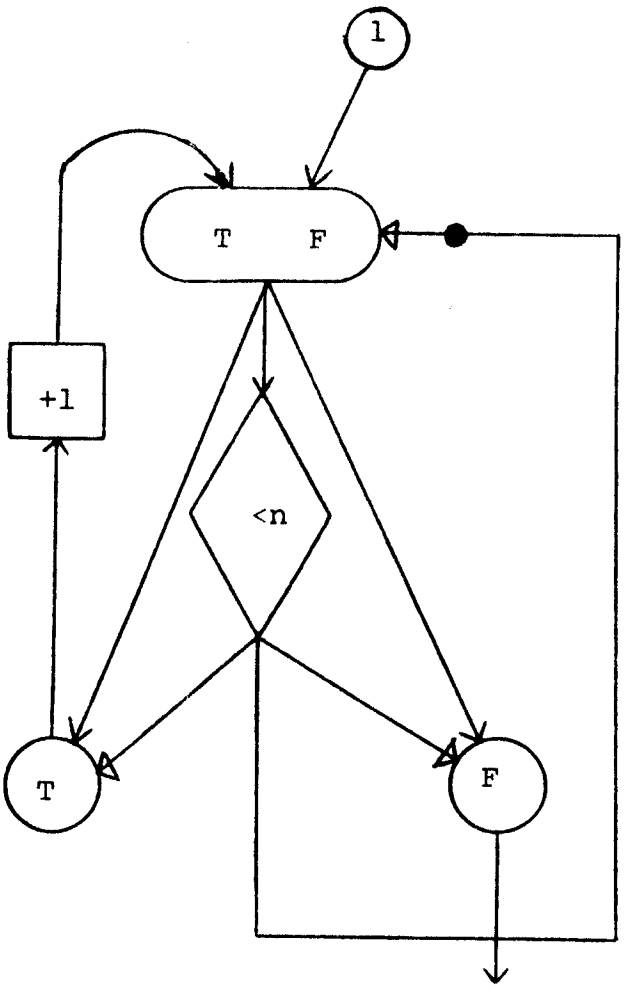
In order to reduce the number of variables inherent in performance analysis, a number of simplifications were made to facilitate the study. Input/output and memory were not considered and the dataflow machine was abstracted into processing elements interconnected by a communication medium. In section three, a method of analysing dataflow programs after execution was presented. The method resulted in several expressions which approximate computation time in terms of the mean allocation,

processing, and communication delays for dataflow instructions. The approach taken in that section was to attempt to eliminate waiting time in the equations. The attempt was successful for non-allocation token waiting time but not for allocation or bus access waiting time. The equations are approximate because the quantitative effects of variance in the distributions above are not known. Furthermore, use of the equations to predict performance under various conditions may be difficult since changes in one term may propagate significant changes in other terms. Future studies may make such interrelationships better known and the equations more explicit. Also, many of these relationships may be applicable only to specific machine architectures.

In section four, a dataflow simulator was described where allocation, processing, and communication delays are computed from independent probability distribution functions. The simulated computation time of dataflow programs in this context is a linear function of the means of these distributions. When resources are unlimited, simulation results closely follow the equations of section three for computation time with slopes and intercepts predicted by the equations (for deterministic C and E distributions). Moreover, when resources are limited the results are linear but the slopes and intercepts are not predicted by the equations presented since the input

distributions do not correspond directly to the execution graph distributions. In all cases, the resources used by a computation were found to be a linear function of computation time. Also presented in that section was the effect of allocation policy. The effect on computation time was found to be minimal when the mean time to allocate a processing element was small relative to other delays. However, when resources are limited, it is very likely that the mean time to allocate will increase. In this event, the allocation policy can have a severe effect on perfomance. In all cases, the allocation policy seemed to have a pronounced effect on resource utilization.

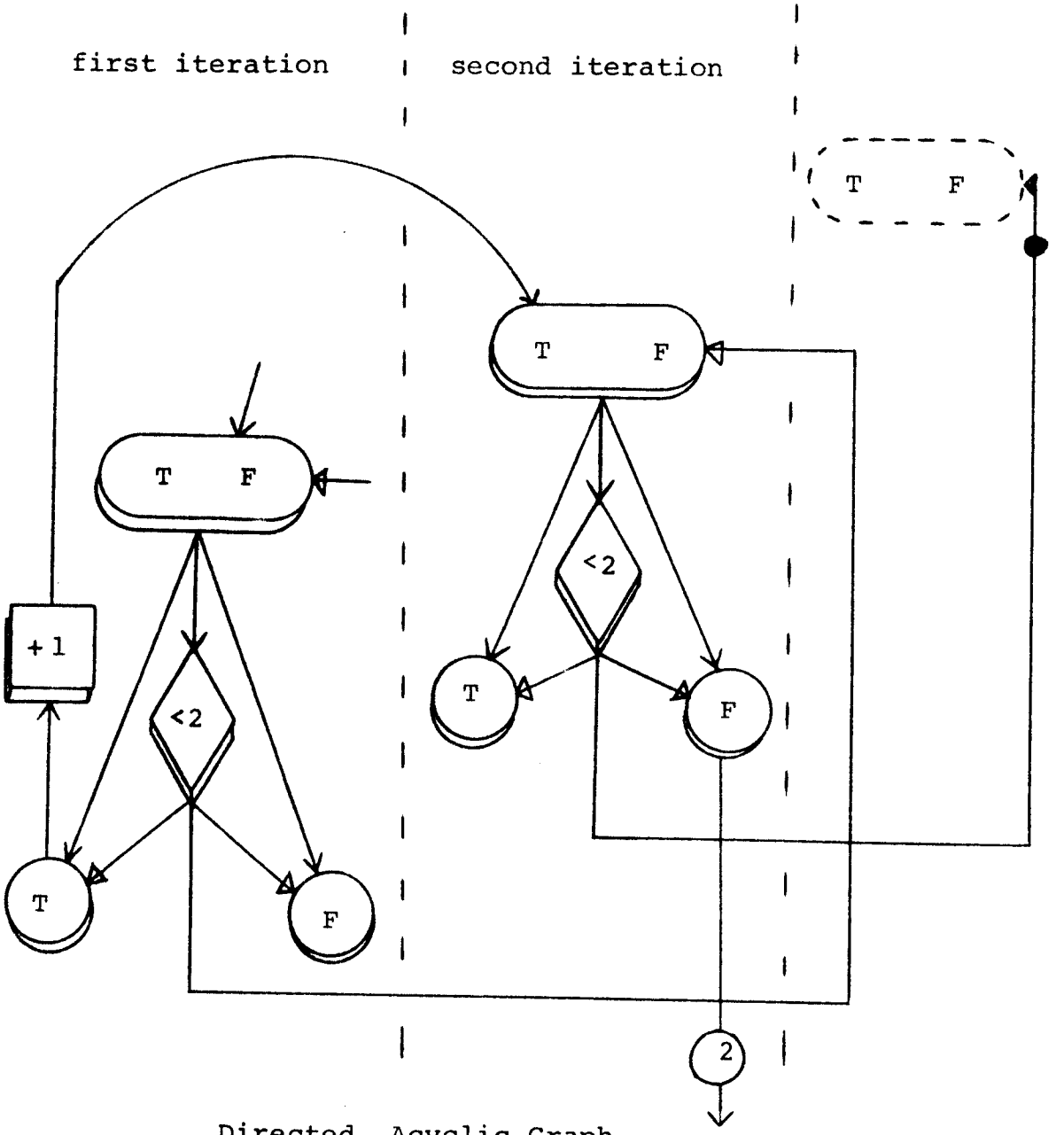
Dataflow Program Graph



Directed, Cyclic Graph
each node is a dataflow base language instruction

Figure (1)

Dataflow Execution Graph

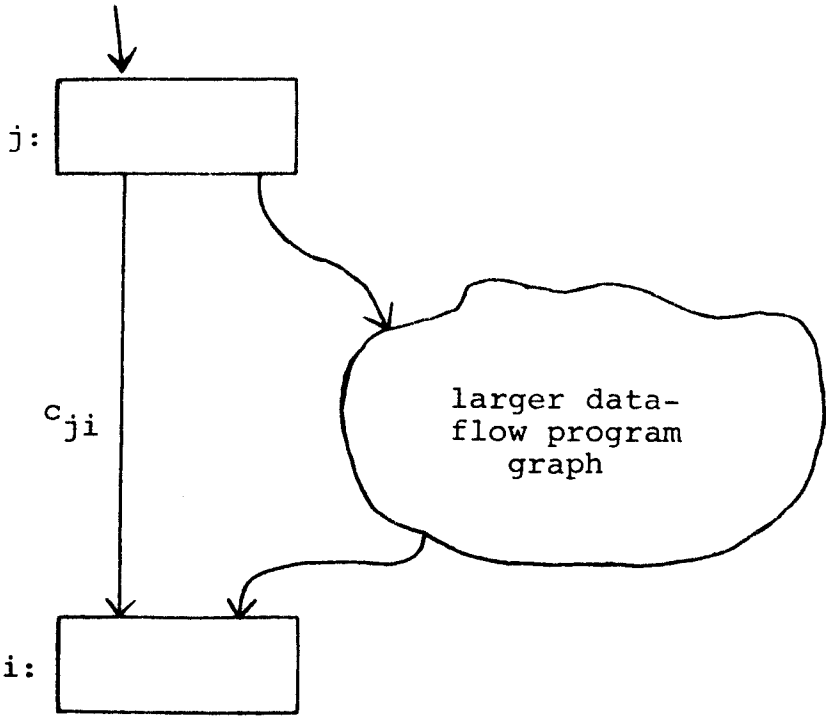


Directed, Acyclic Graph
each node is an "activity"

Figure (2)

Path Length

Change due to High Variance in C (or E)



If $(c_{ji} >>> \bar{C})$

then the longest path is leftmost
else the longest path is rightmost

Figure (3)

Execution Graph Segment

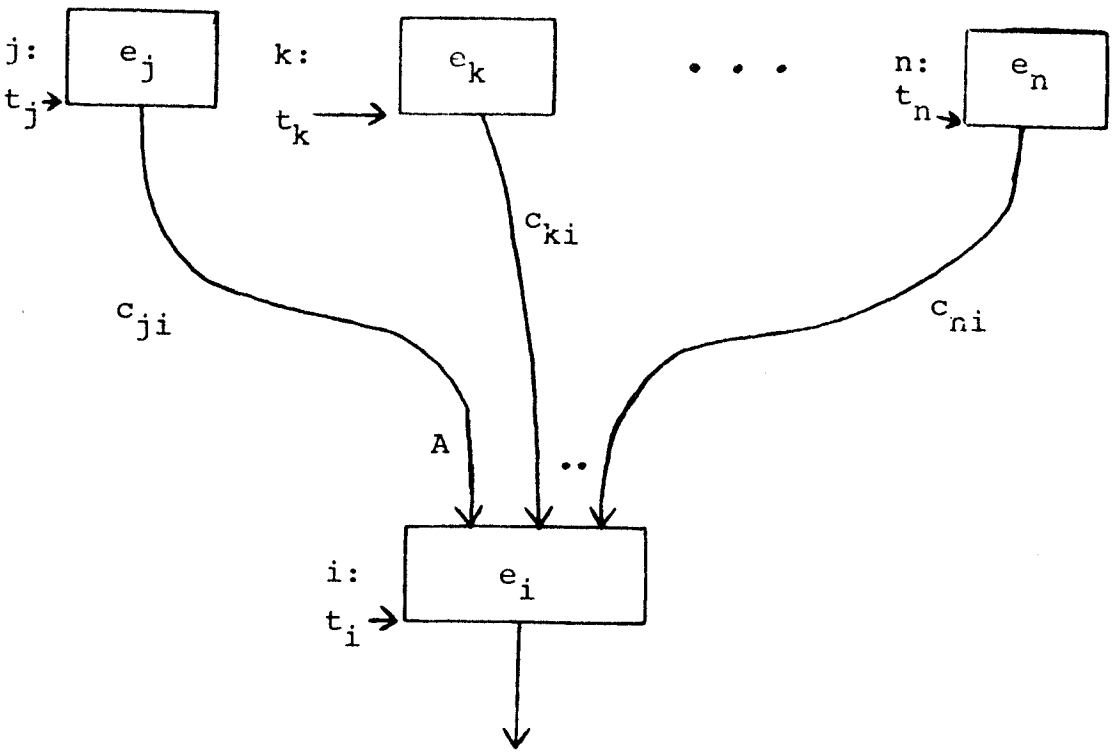


Figure (4)

Execution Graph Segment
with Bus Waiting time, W_B

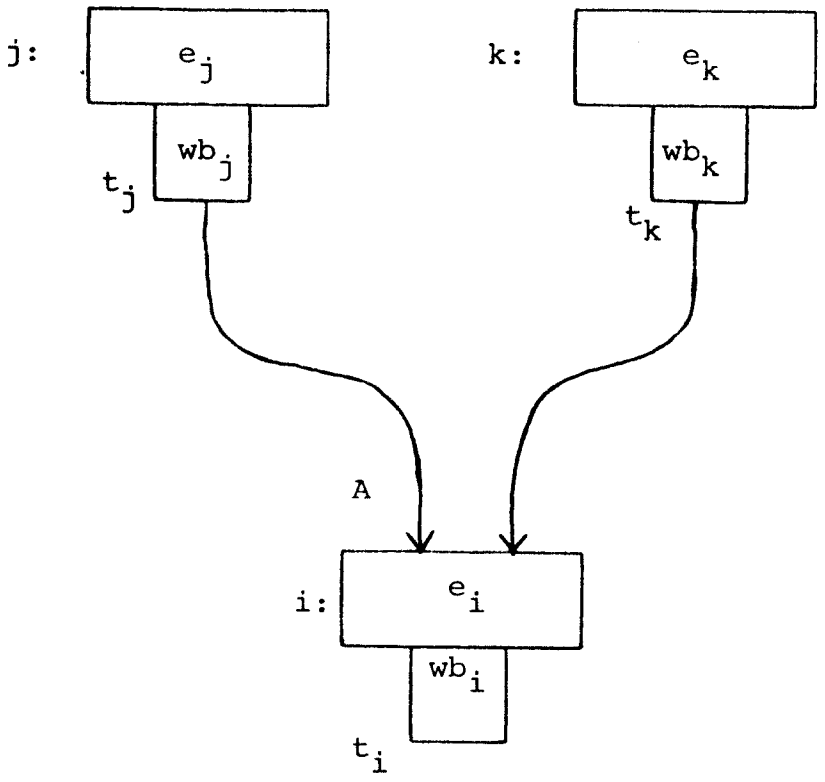
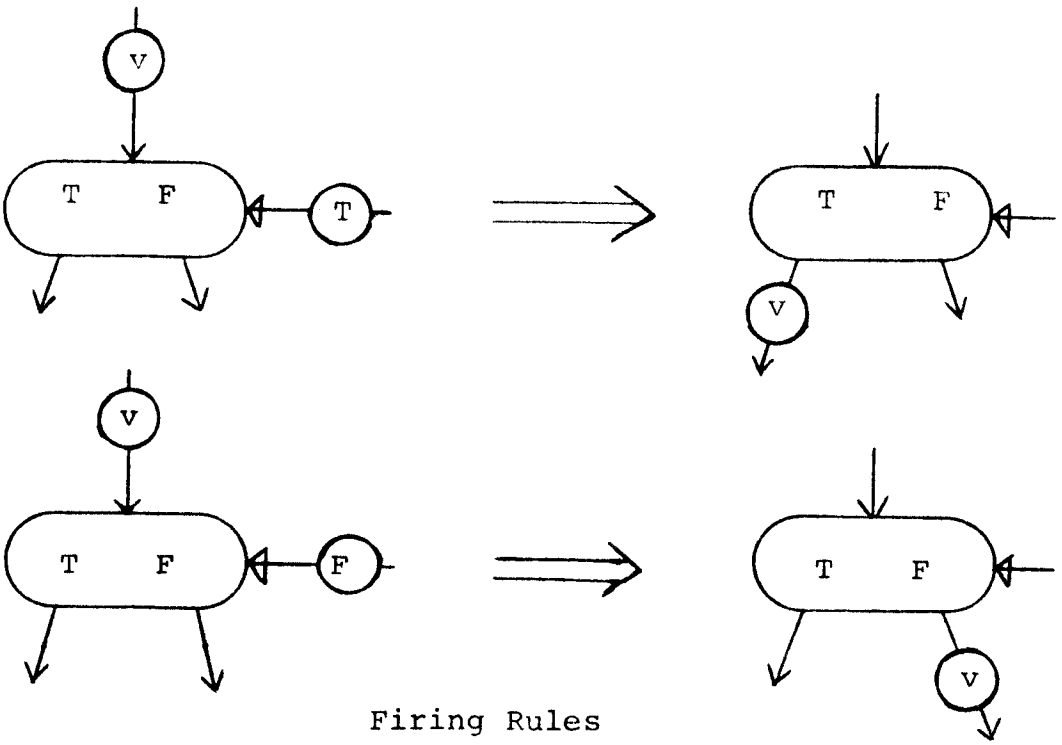


Figure (5)

Simple Switch Actor



Firing Rules

Figure (6a)

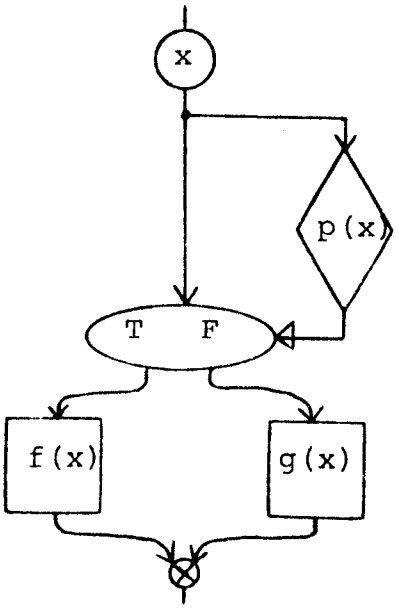


Figure (6b) If then/else Construct

Vectored Switch Actor

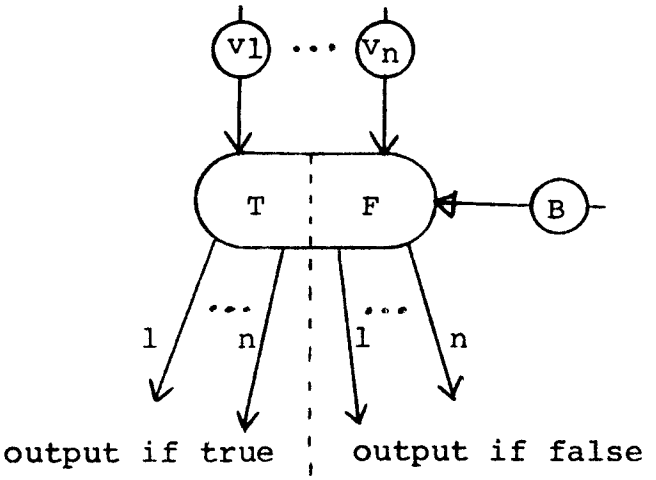
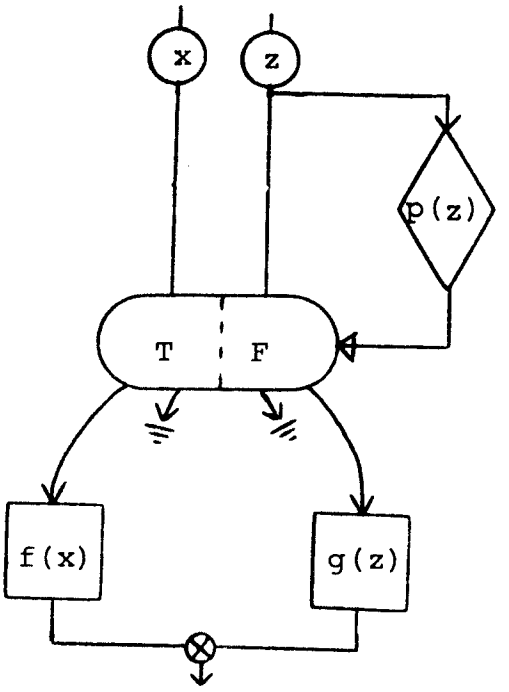


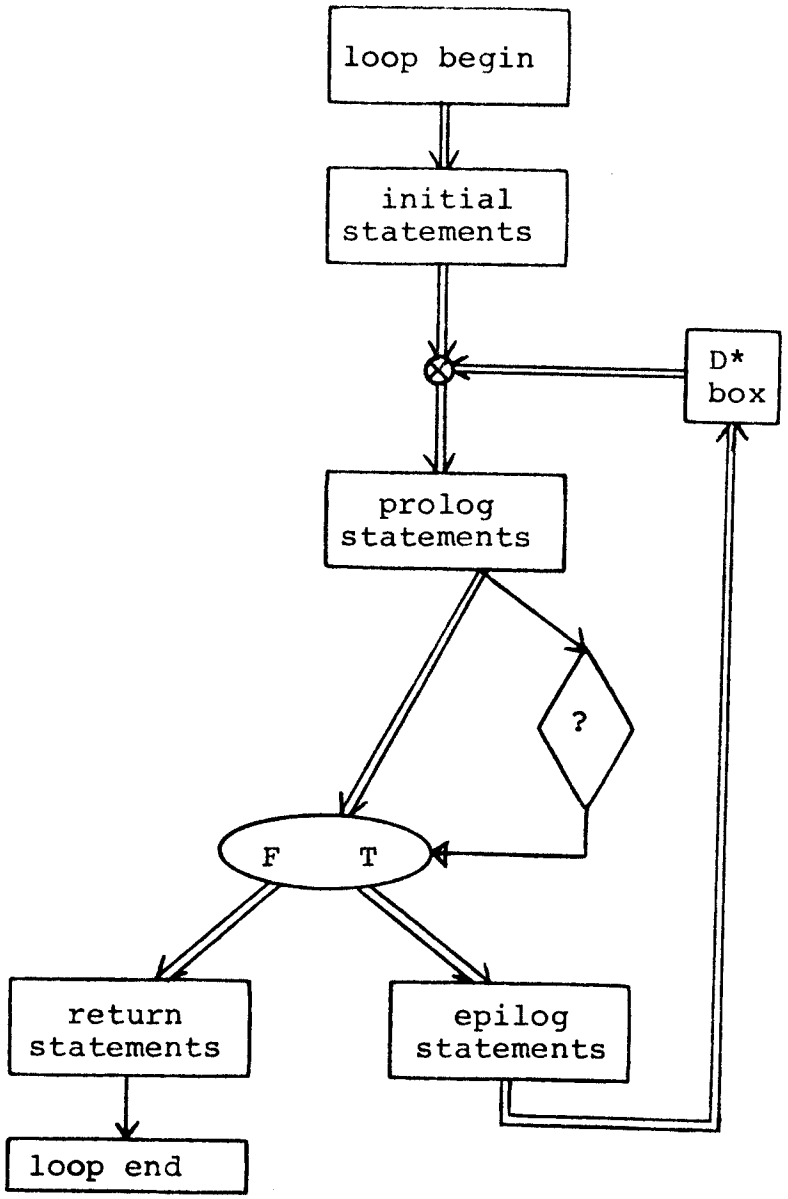
Figure (7a)



If p(z) then f(x) else g(z)

Figure (7b)

General Loop Construct

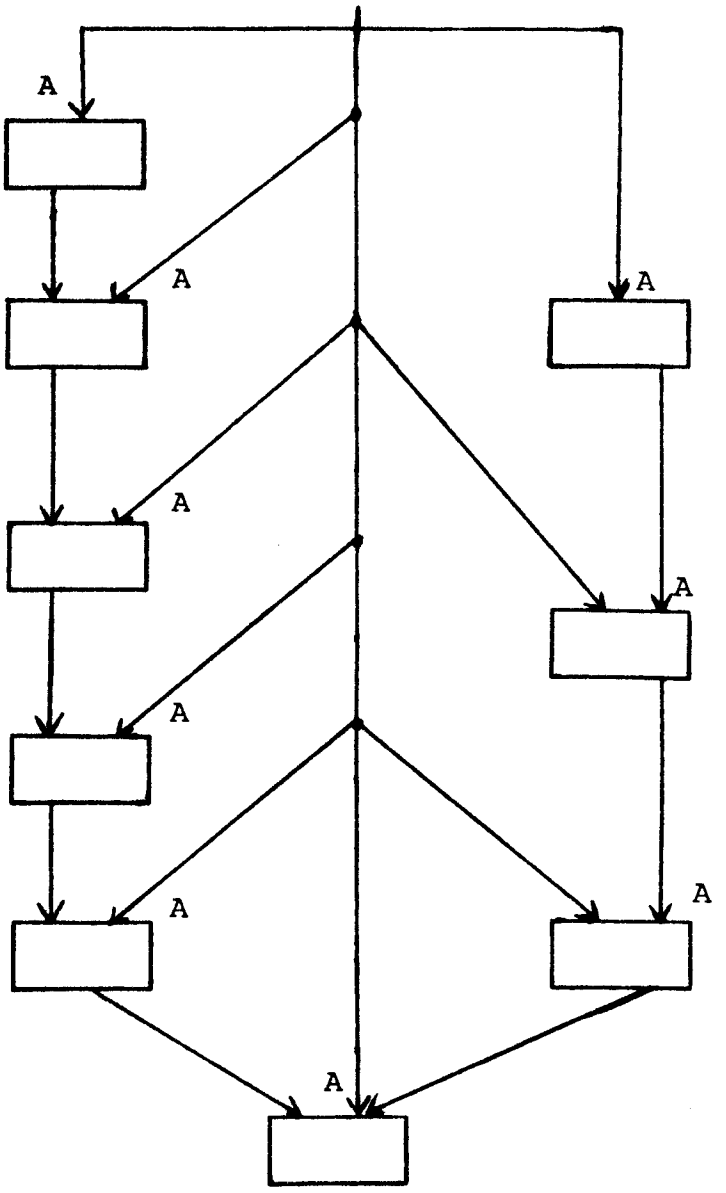


*The D box ensures that each iteration of the loop produces distinctly tagged tokens. A method for accomplishing this is given in [AG75].

Note: Vectored lines indicate that in general more than one actor of the type indicated exists in parallel.

Figure (8)

Path Length Change Due to
Mixed Allocation Policy

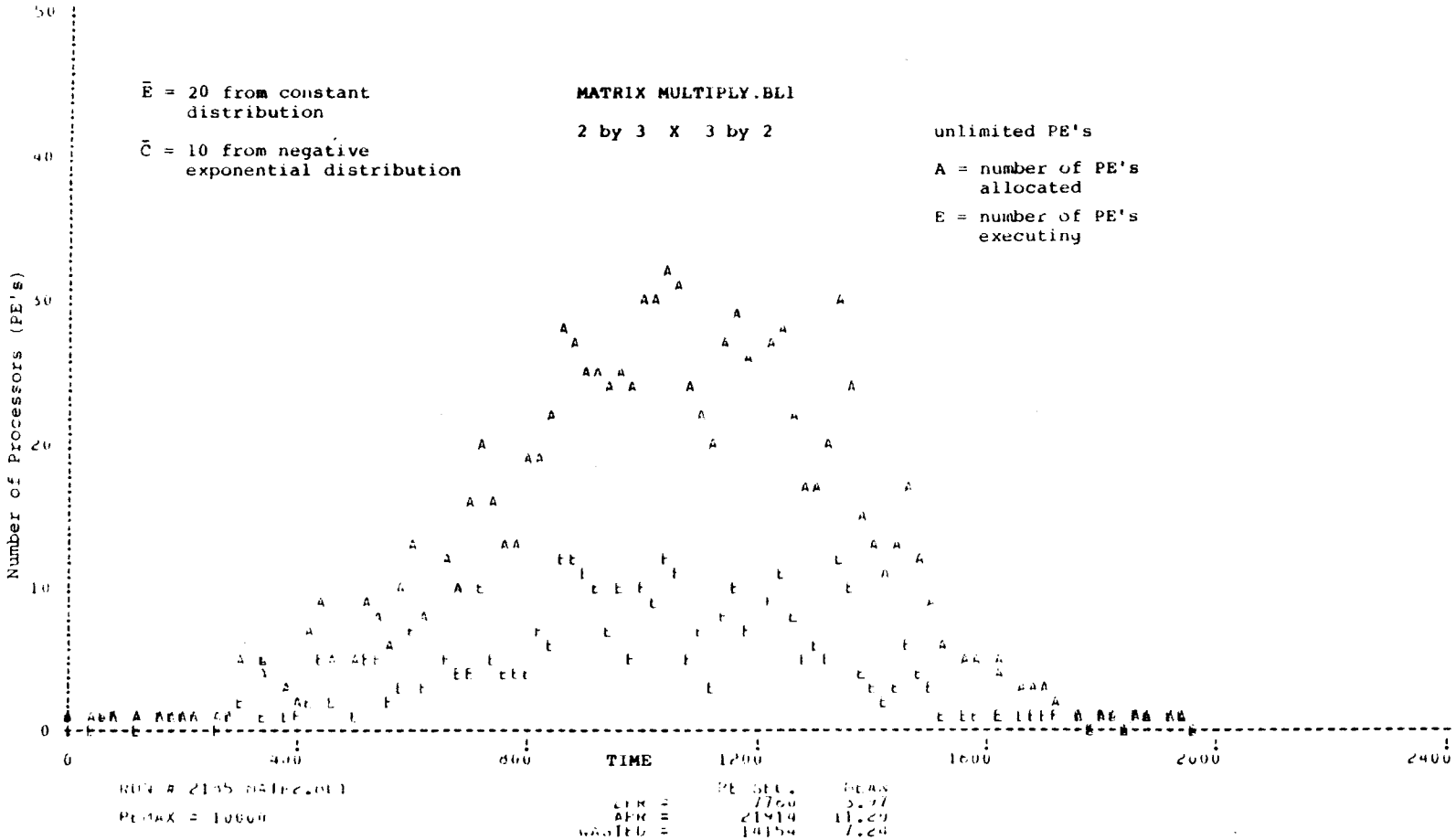


Critical path is leftmost when $\bar{C}_A \leq \bar{E}$
(parallel allocation)

Critical path is rightmost when $\bar{C}_A >> \bar{E}$
(sequential allocation)

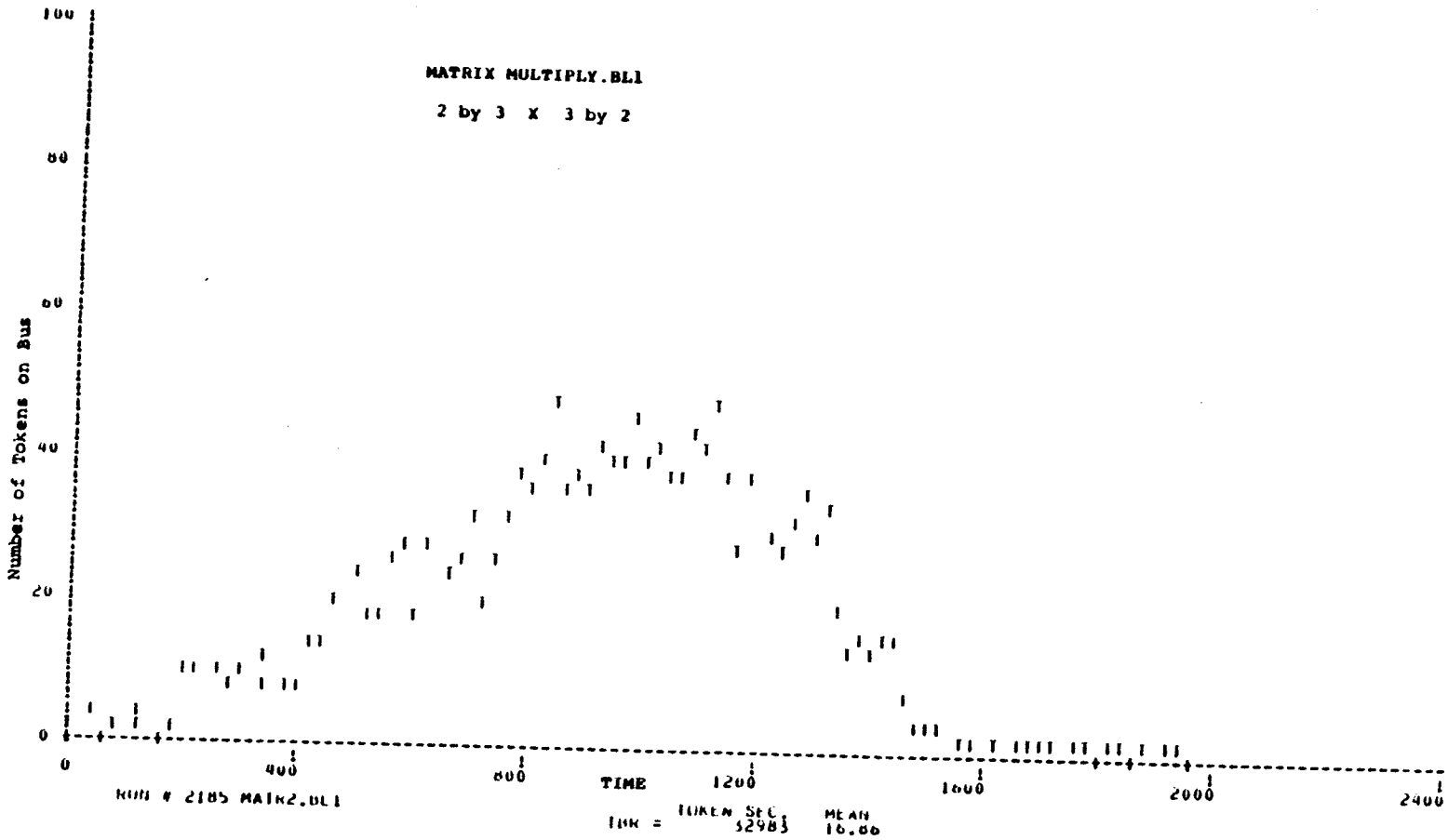
Figure (9)

ALLOCATED AND EXECUTING PE's



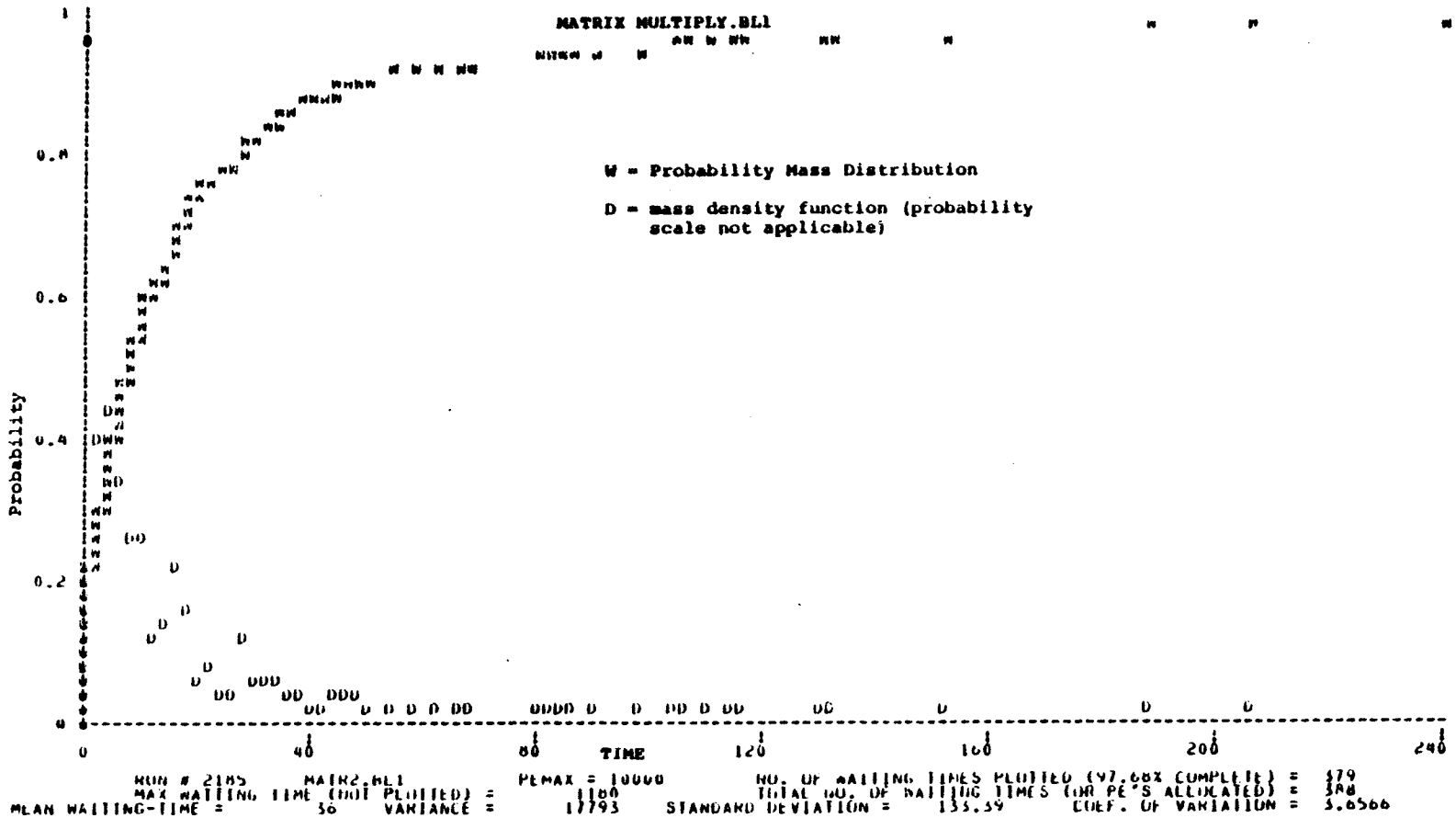
Graph (1)

TOKEN BUS LOAD (unlimited PE's)



Graph (2)

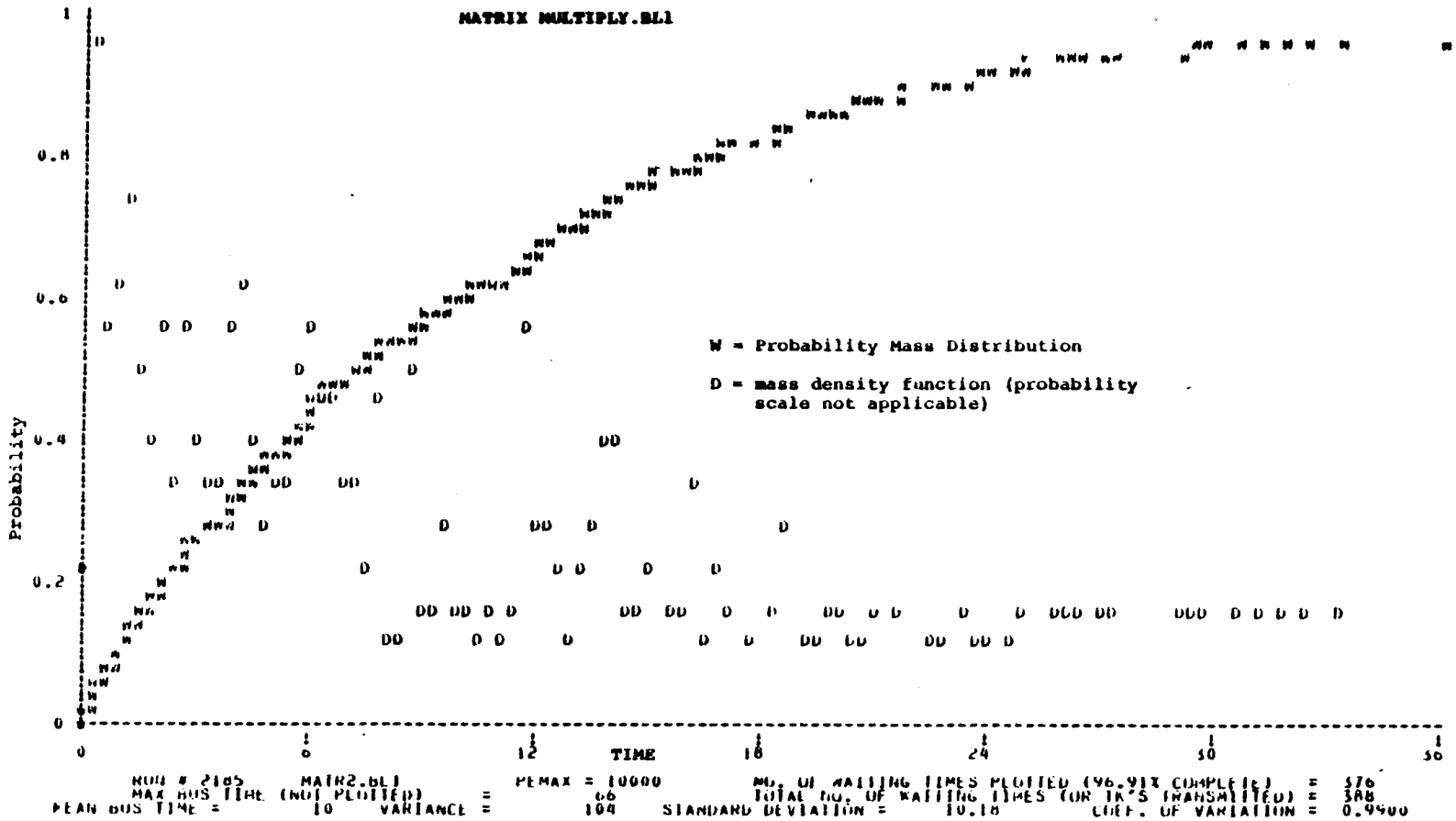
PE WAITING-TIME (PROBABILITY MASS) DISTRIBUTION



Graph (3)

ALLOCATION TOKEN BUS-TIME (PROBABILITY MASS)

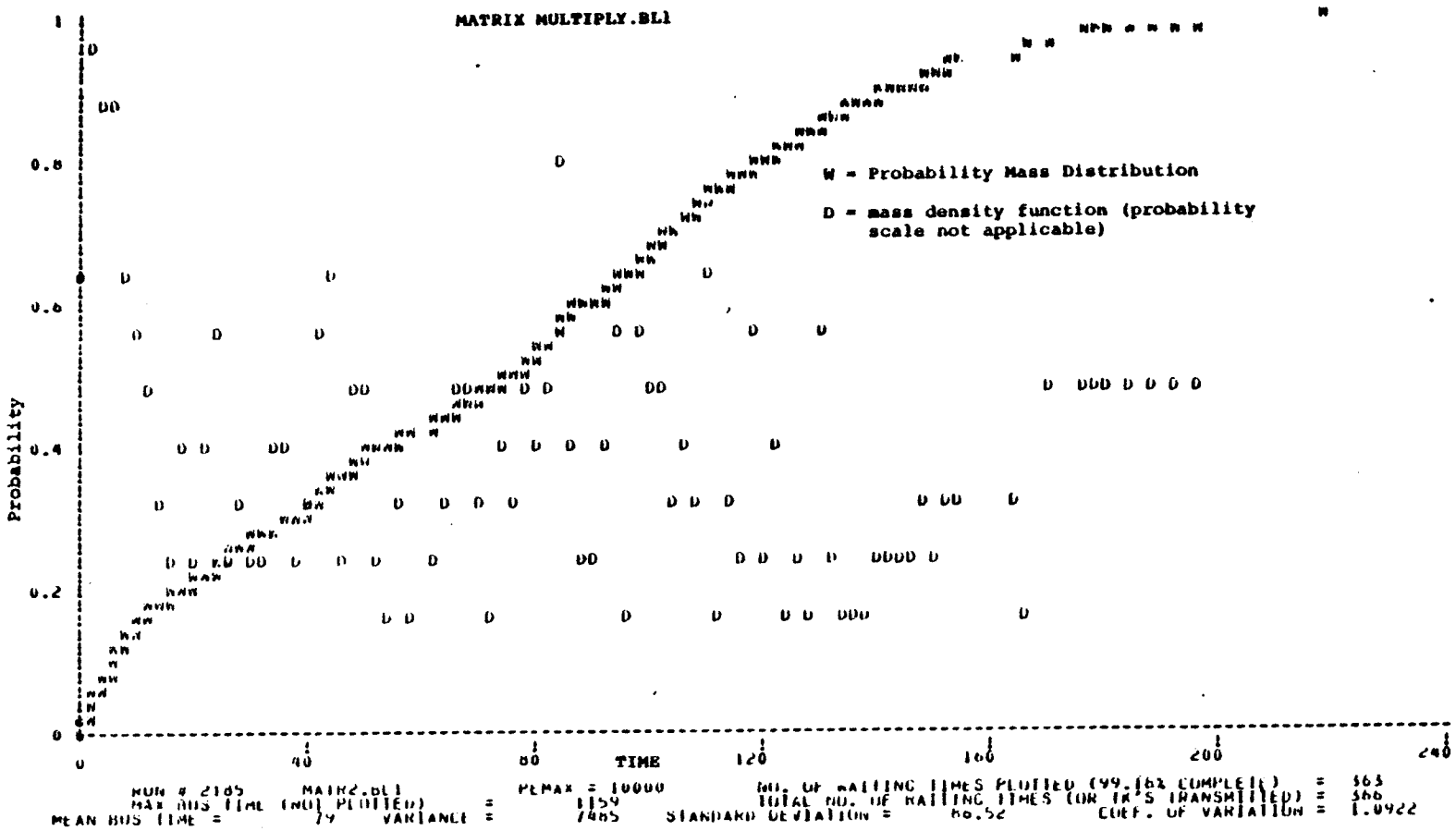
DISTRIBUTION



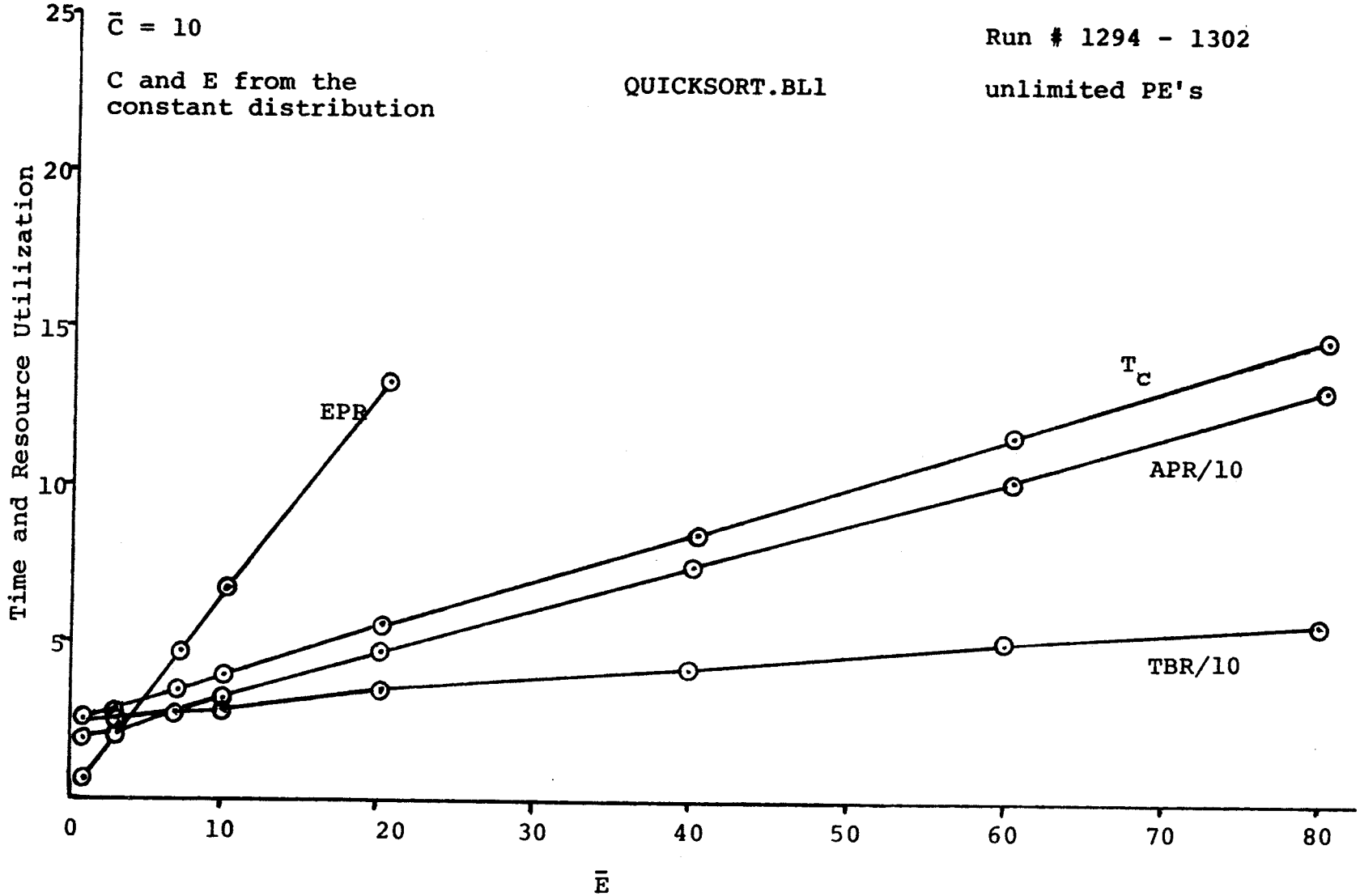
Graph (4)

NON-ALLOCATION TOKEN BUS-TIME (PROBABILITY MASS)

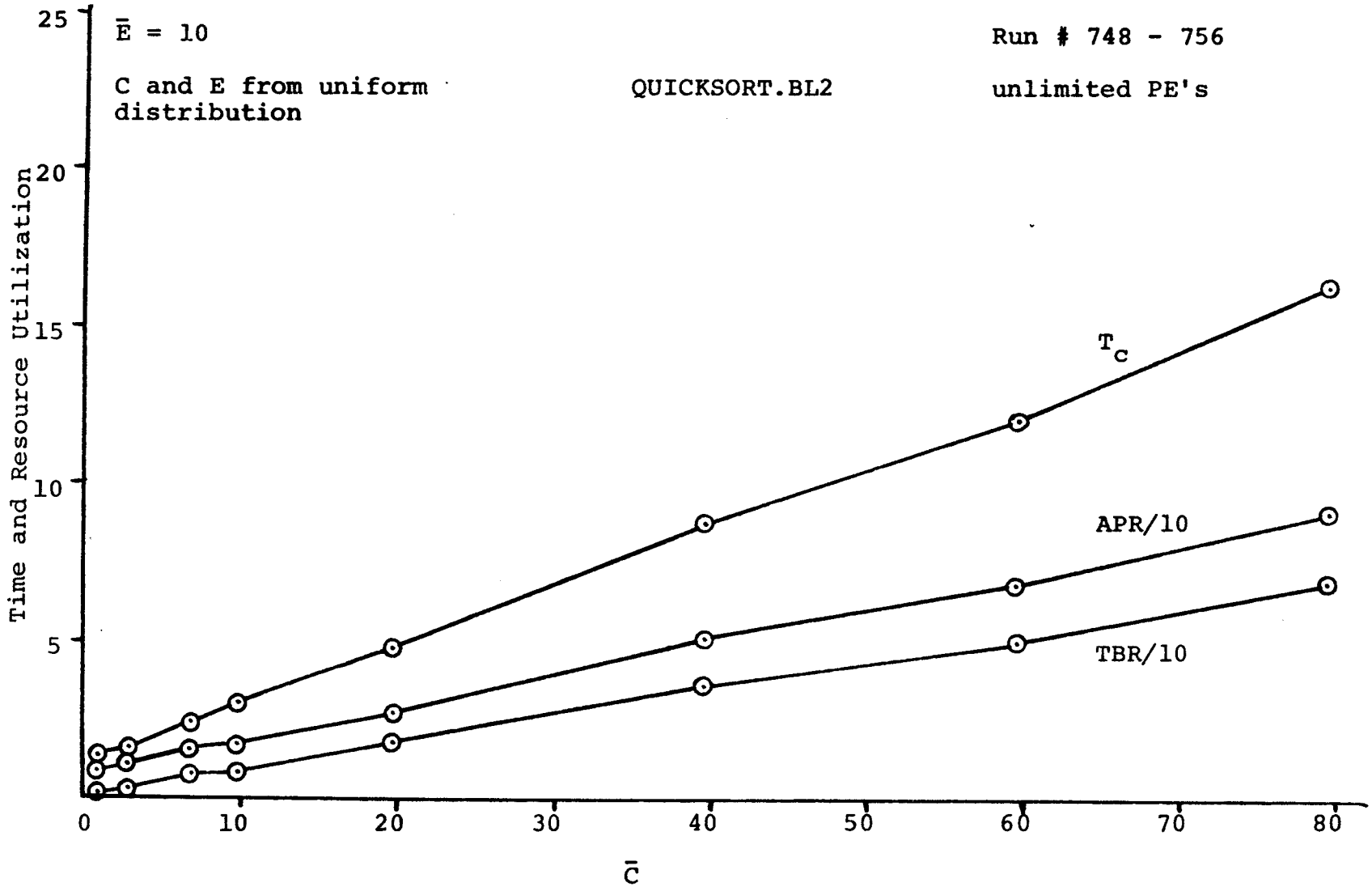
DISTRIBUTION



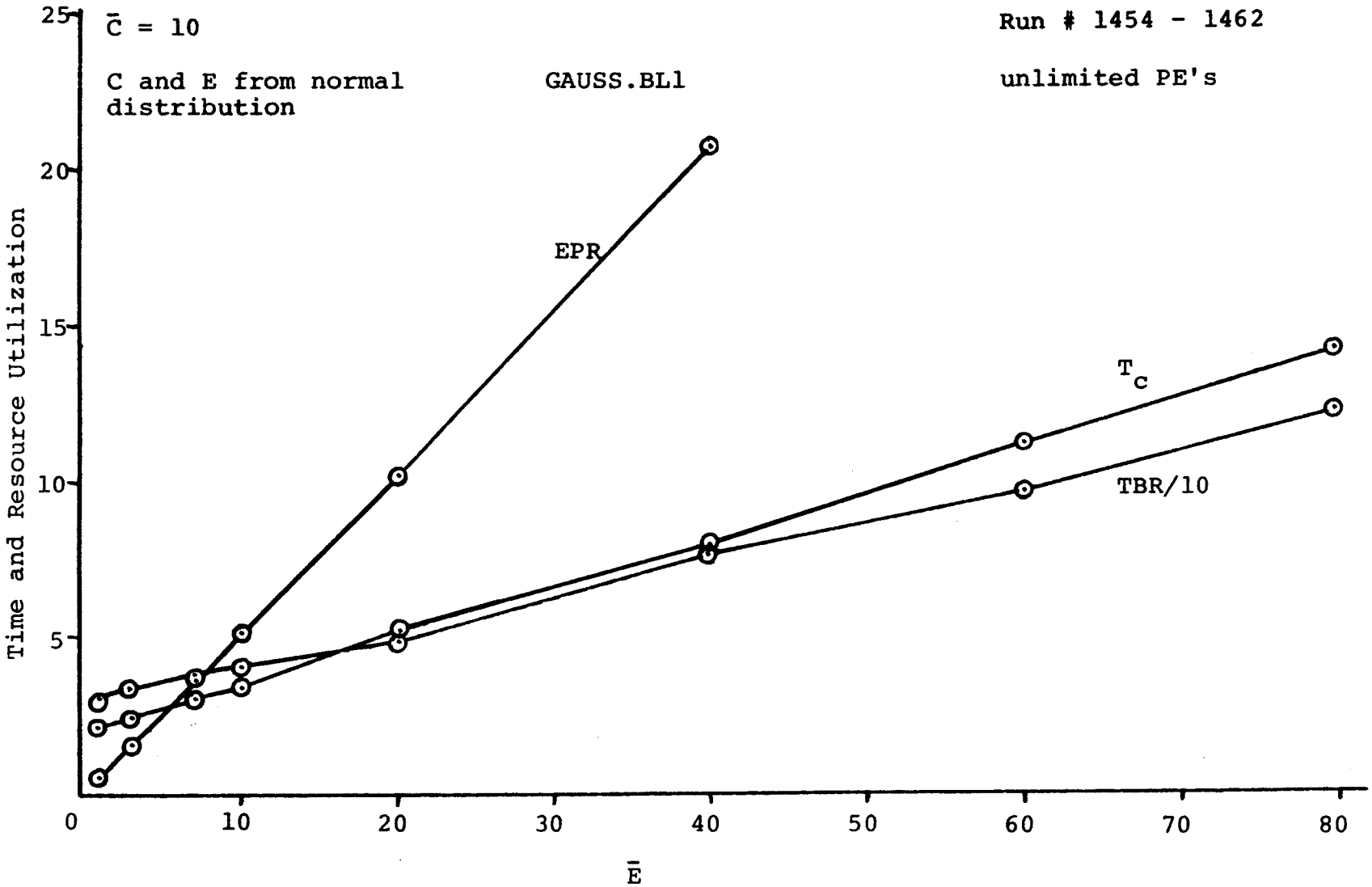
Graph (5)


$\bar{C}$ = 10
C and E from the constant distribution
QUICKSORT.BL1
Run # 1294 - 1302
unlimited PE's
Time and Resource Utilization
25
20
15
10
5
0
10
20
30
40
50
60
70
80
$\bar{E}$
EPR
T_C
APR/10
TBR/10

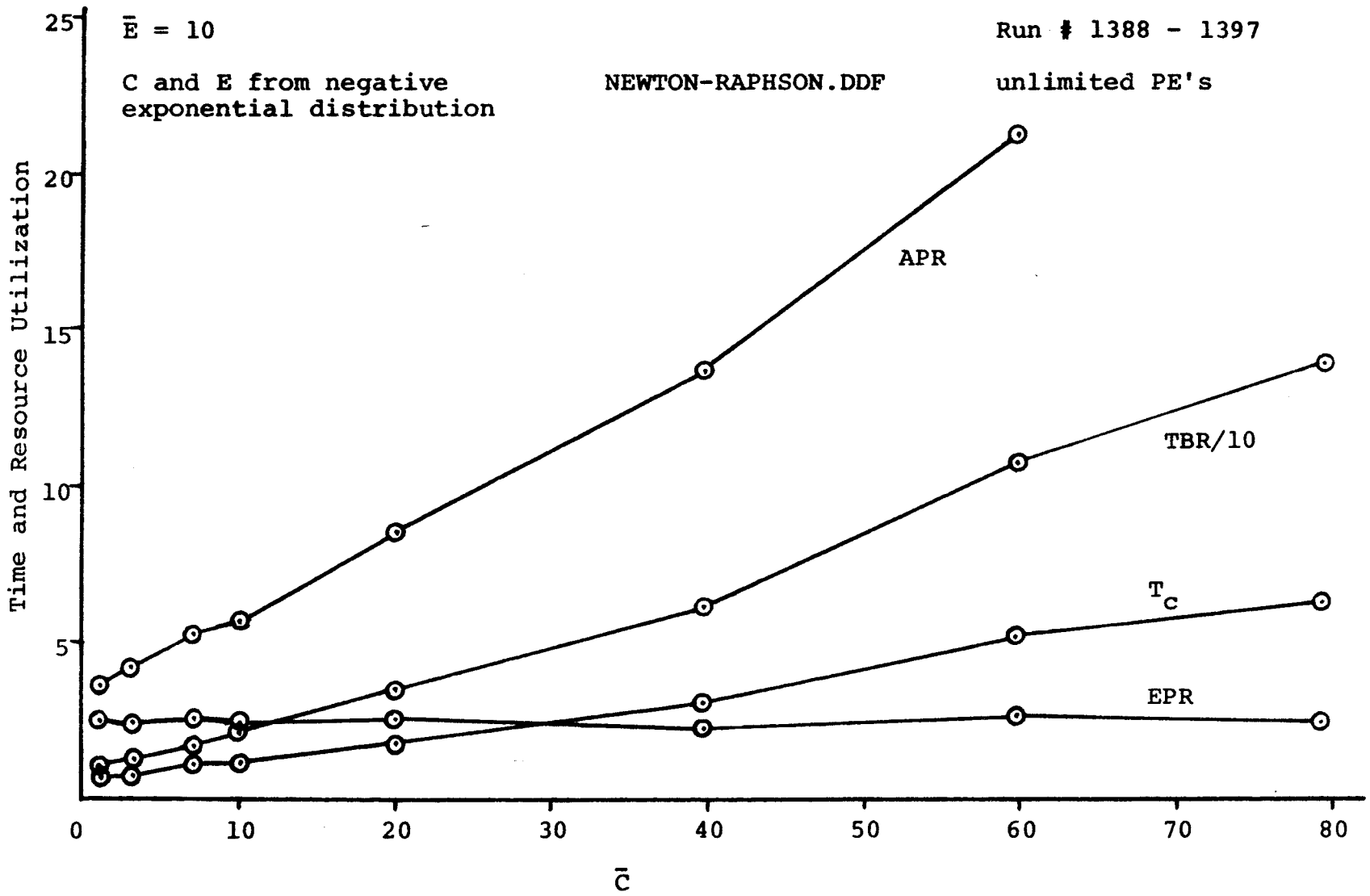
Graph (6)


$\bar{E}$ = 10
C and E from uniform distribution
QUICKSORT.BL2
Run # 748 - 756
unlimited PE's
Time and Resource Utilization
25
20
15
10
5
0
10
20
30
40
50
60
70
80
$\bar{C}$
T_C
APR/10
TBR/10

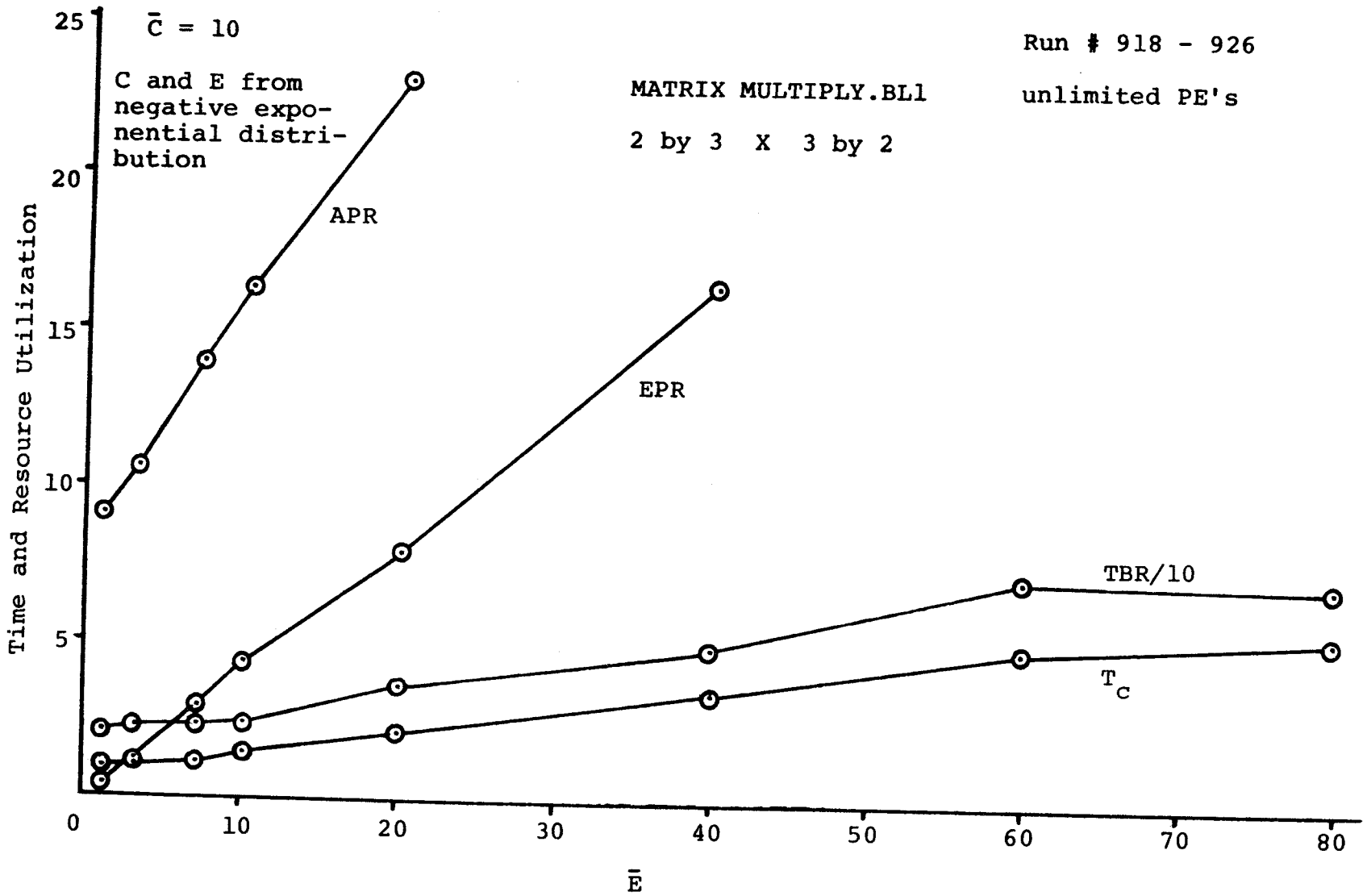
Graph (7)


$\bar{C}$ = 10
C and E from normal distribution
GAUSS.BL1
Run # 1454 - 1462
unlimited PE's
EPR
T_C
TBR/10
Time and Resource Utilization
$\bar{E}$
0
10
20
30
40
50
60
70
80
5
10
15
20
25

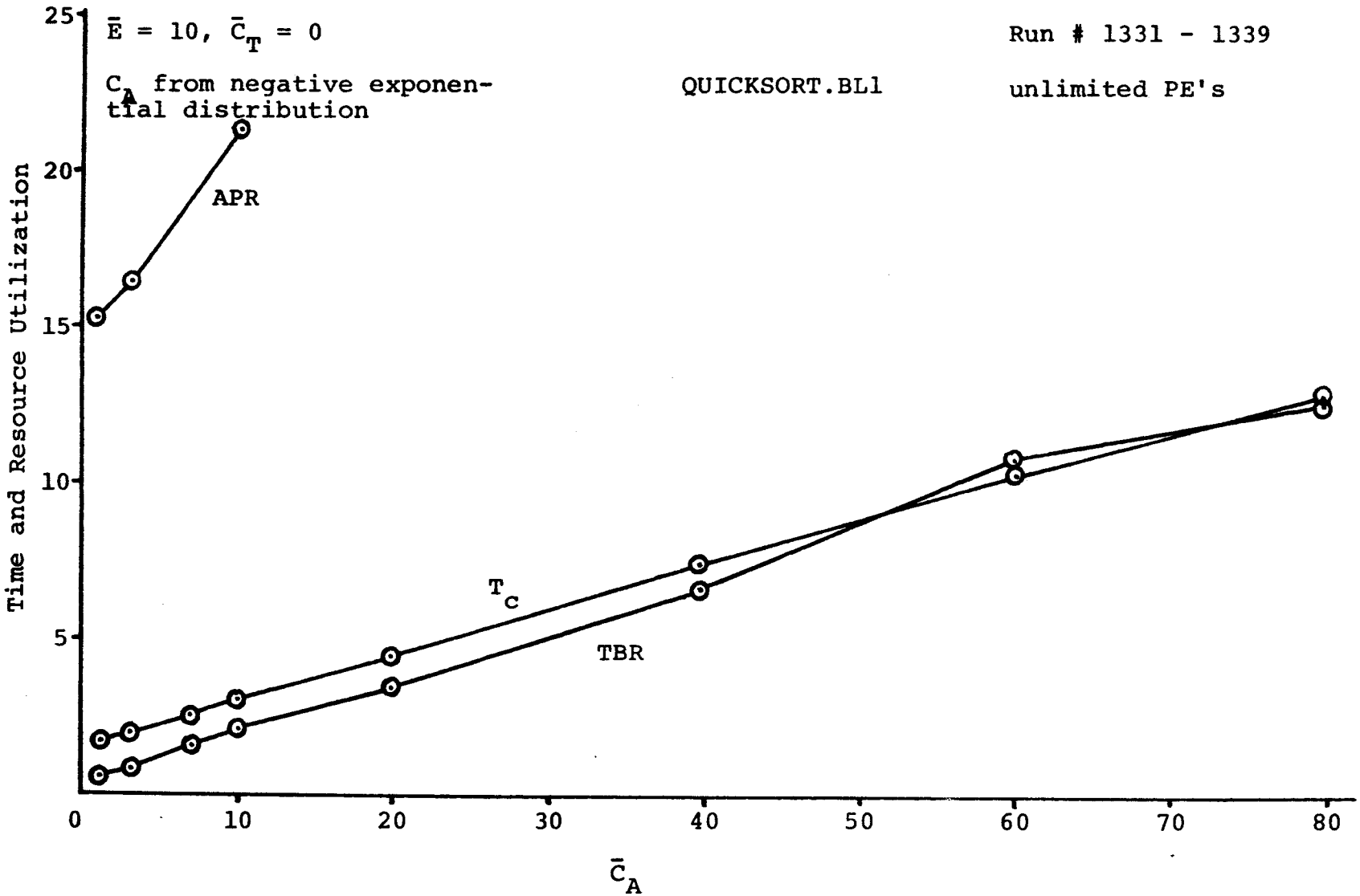
Graph (8)


$\bar{E}$ = 10
C and E from negative exponential distribution
NEWTON-RAPHSON.DDF
Run # 1388 - 1397
unlimited PE's
Time and Resource Utilization
25
20
15
10
5
APR
TBR/10
T_C
EPR
0
10
20
30
40
50
60
70
80
$\bar{C}$

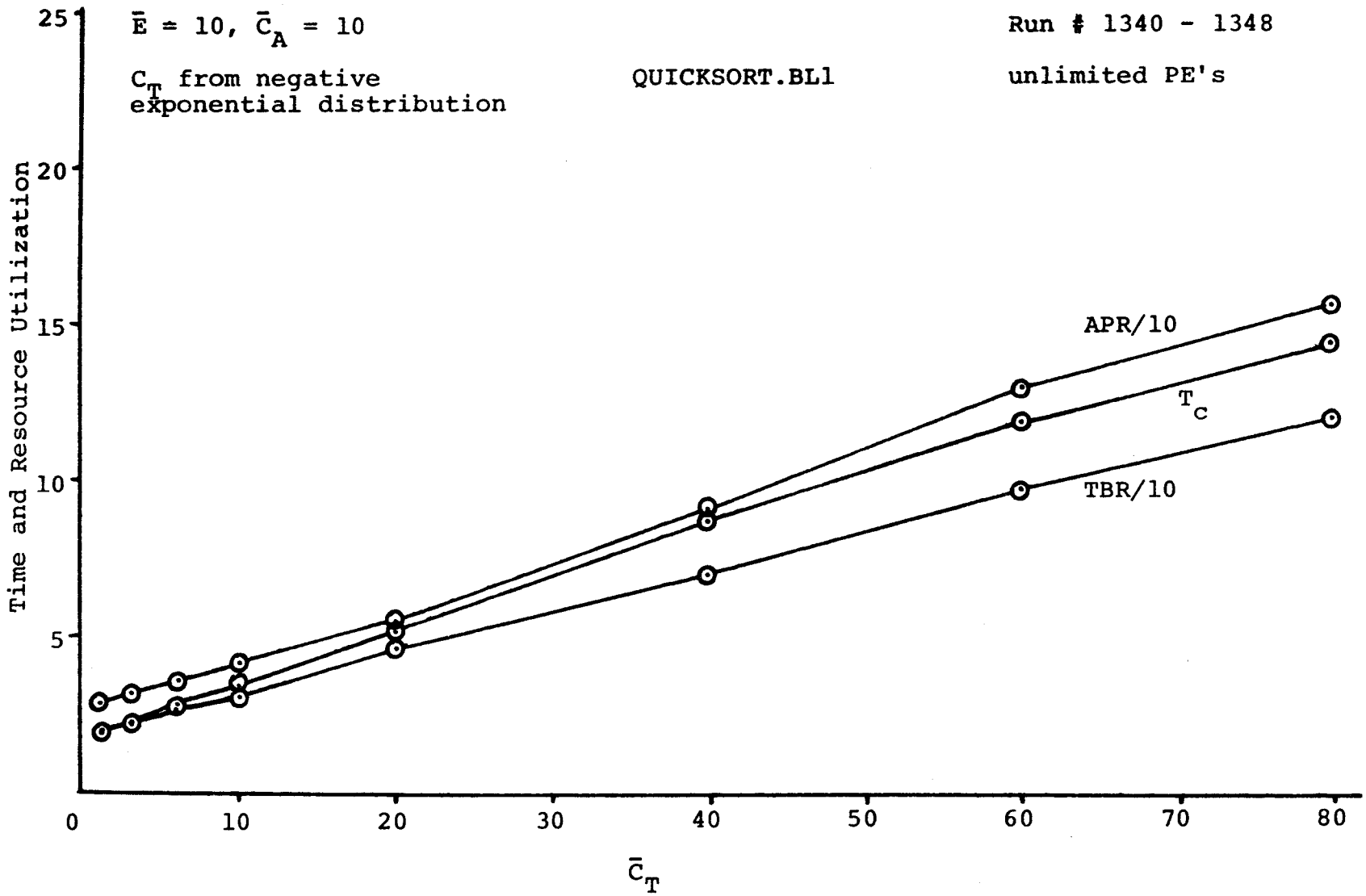
Graph (9)


$\bar{C}$ = 10
C and E from negative exponential distribution
MATRIX MULTIPLY.BL1
2 by 3 X 3 by 2
Run # 918 - 926
unlimited PE's
APR
EPR
TBR/10
T_C
Time and Resource Utilization
$\bar{E}$
0
5
10
15
20
25
10
20
30
40
50
60
70
80

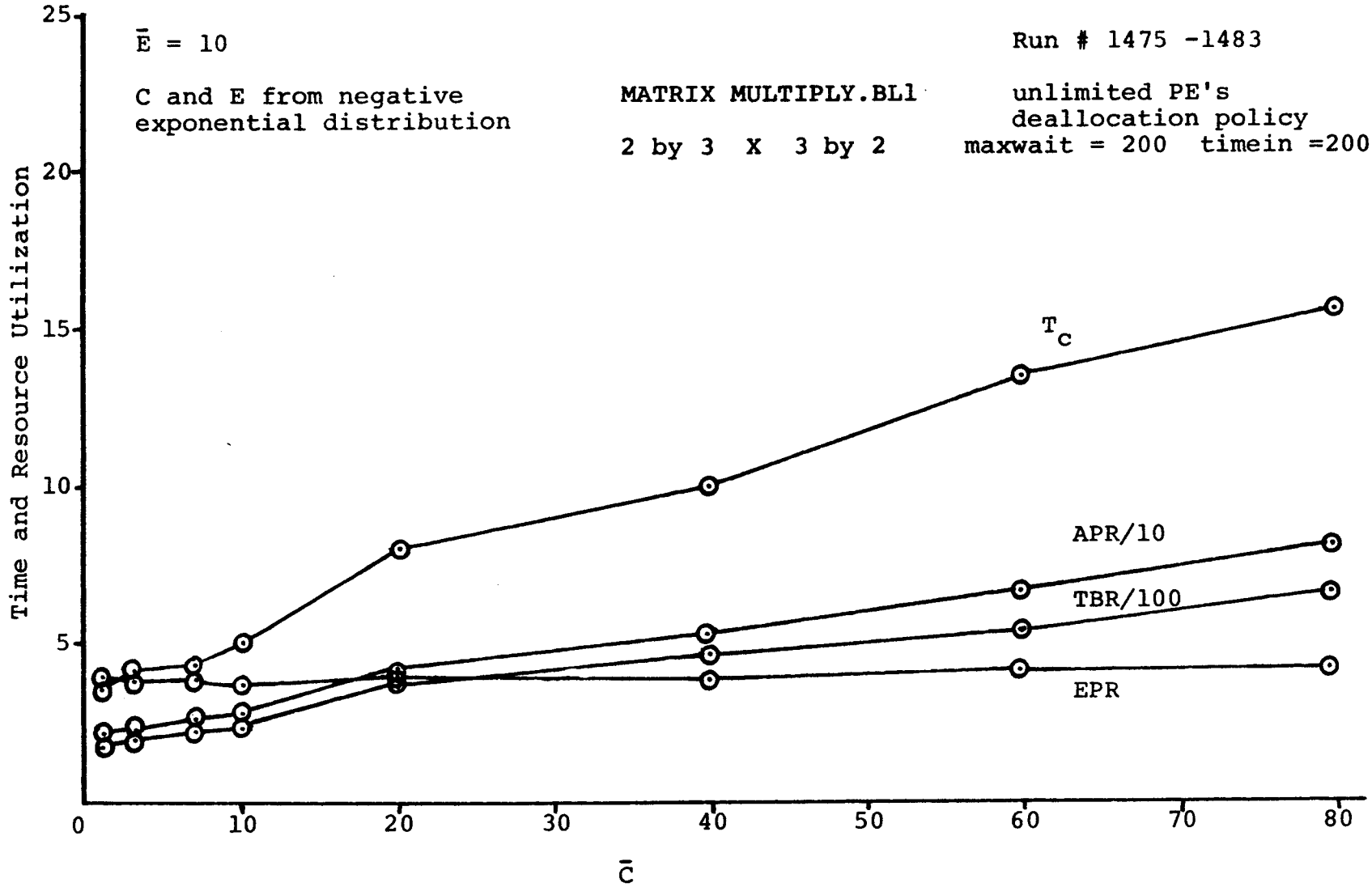
Graph (10)


$\bar{E}$ = 10, $\bar{C}_T$ = 0
C_A from negative exponential distribution
QUICKSORT.BL1
Run # 1331 - 1339
unlimited PE's
APR
T_C
TBR
Time and Resource Utilization
25
20
15
10
5
0
10
20
30
40
50
60
70
80
$\bar{C}_A$

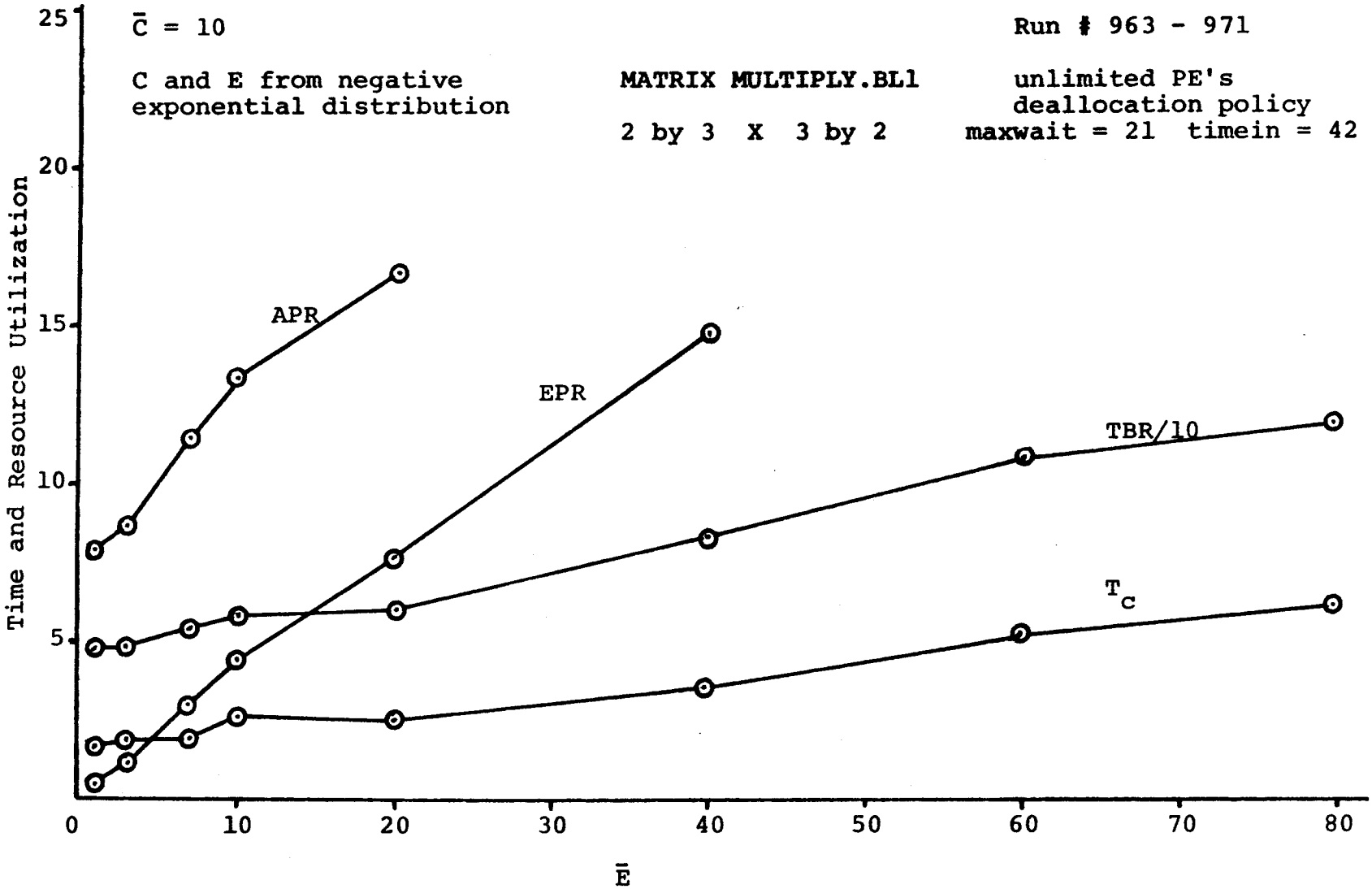
Graph (11)


$\bar{E}$ = 10, $\bar{C}_A$ = 10
C_T from negative exponential distribution
QUICKSORT.BL1
Run # 1340 - 1348
unlimited PE's
Time and Resource Utilization
25
20
15
10
5
0
10
20
30
40
50
60
70
80
APR/10
T_C
TBR/10
$\bar{C}_T$

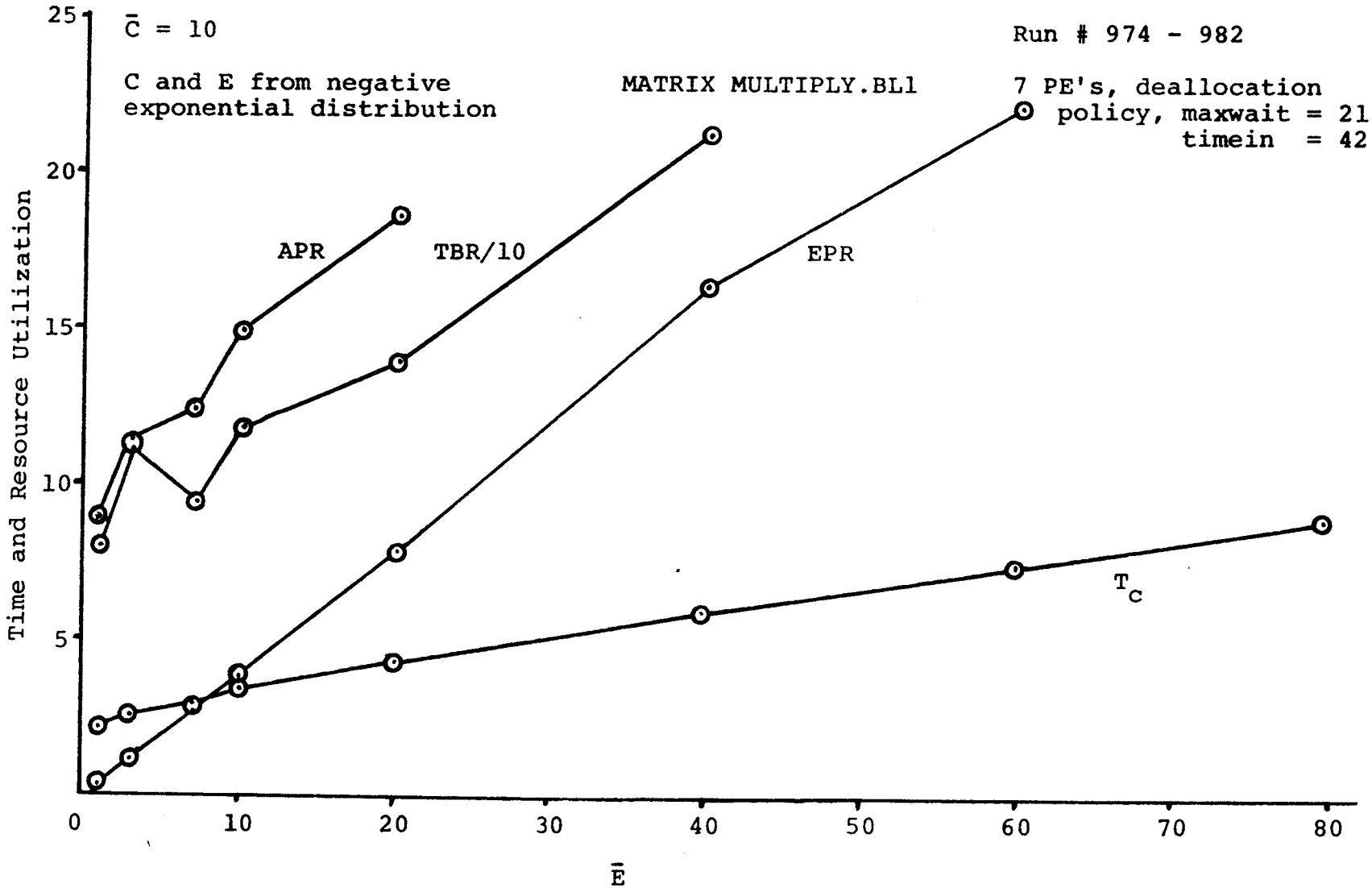
Graph (12)


$\bar{E}$ = 10
C and E from negative exponential distribution
MATRIX MULTIPLY.BL1
2 by 3 X 3 by 2
Run # 1475 -1483
unlimited PE's
deallocation policy
maxwait = 200 timein =200
Time and Resource Utilization
25
20
15
10
5
0
10
20
30
40
50
60
70
80
$\bar{C}$
T_C
APR/10
TBR/100
EPR

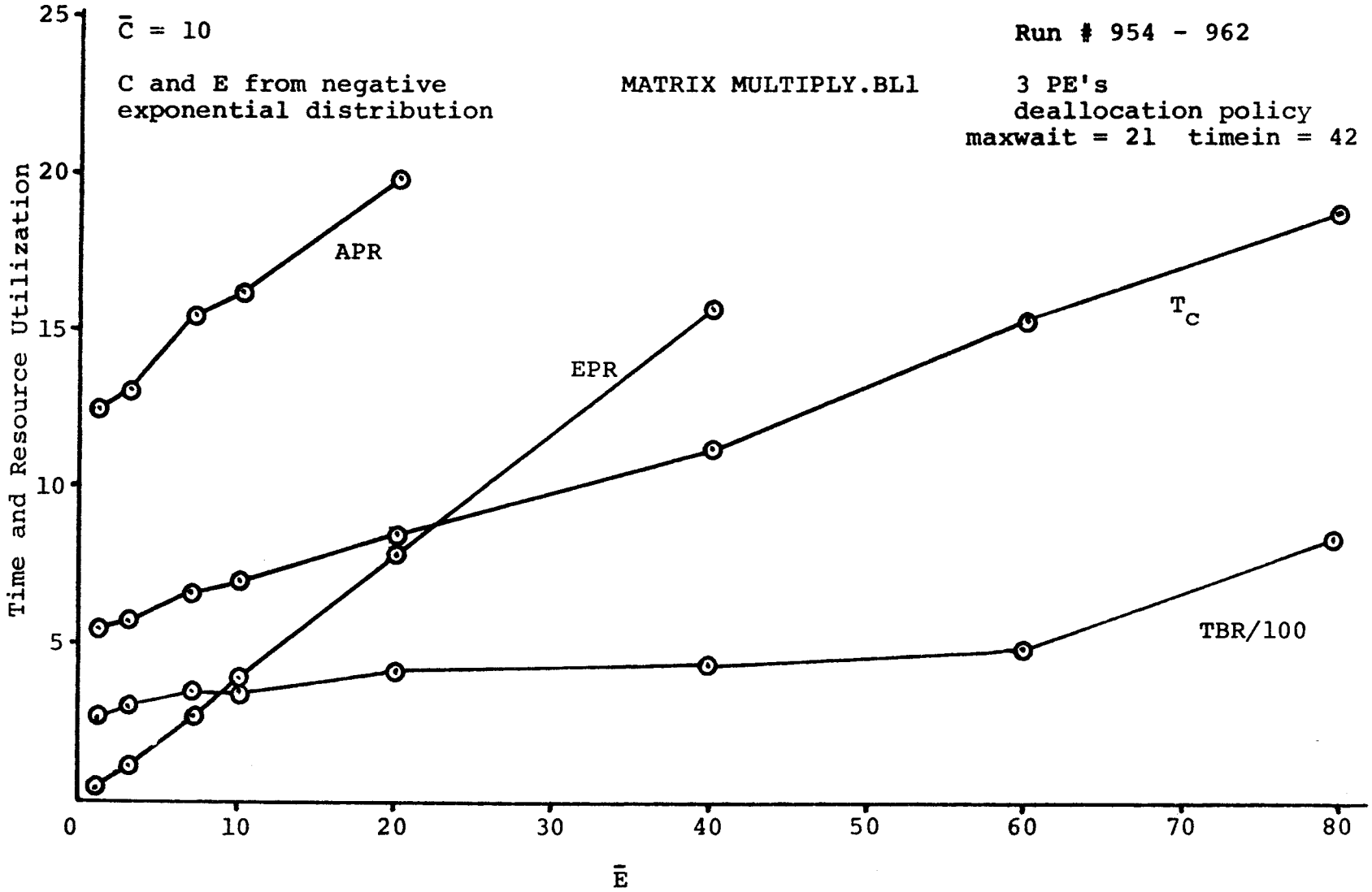
Graph (13)


$\bar{C}$ = 10
C and E from negative exponential distribution
MATRIX MULTIPLY.BL1
2 by 3 X 3 by 2
Run # 963 - 971
unlimited PE's
deallocation policy
maxwait = 21 timein = 42
Time and Resource Utilization
25
20
15
10
5
0
10
20
30
40
50
60
70
80
APR
EPR
TBR/10
T_C
$\bar{E}$

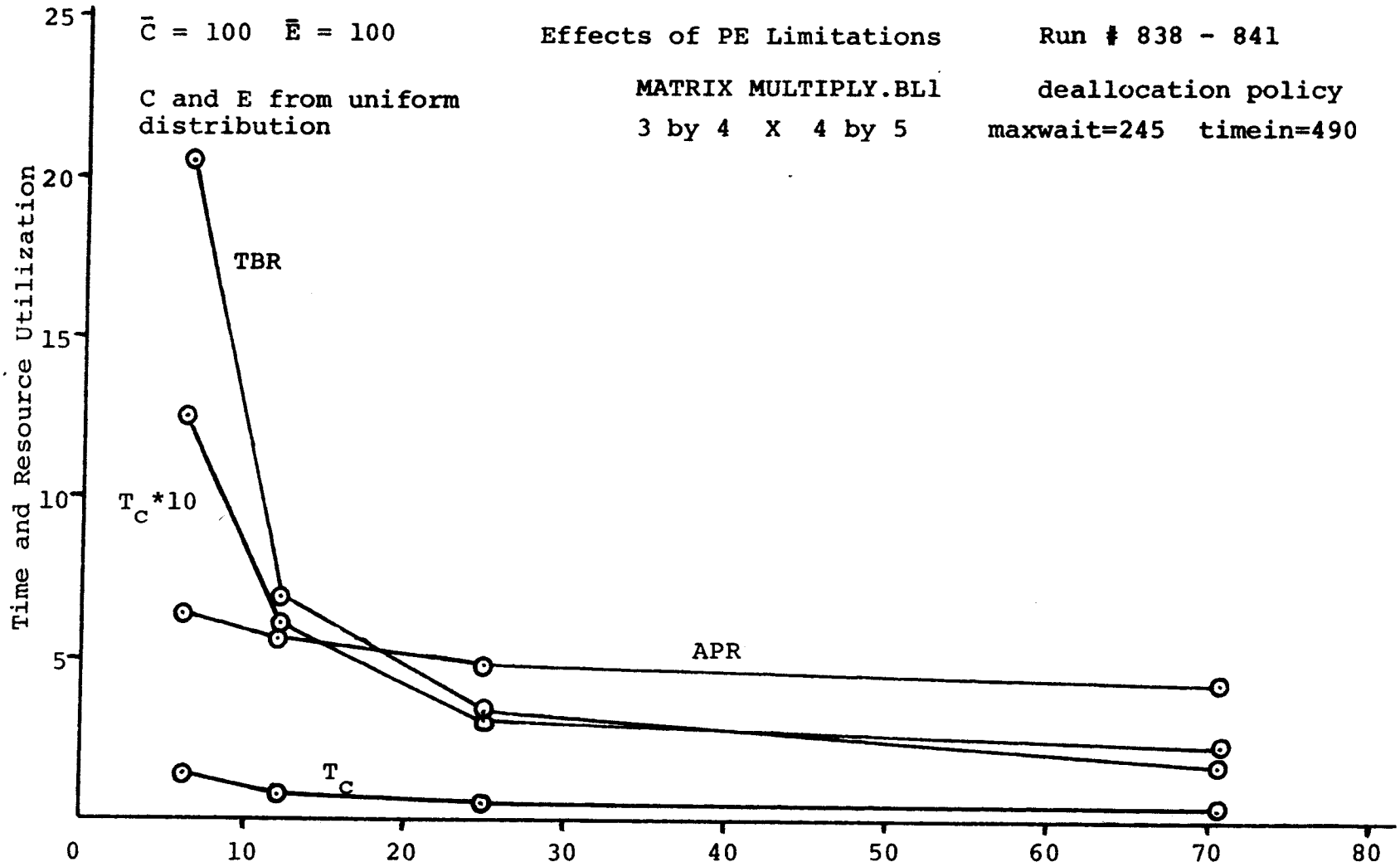
Graph (14)


$\bar{C}$ = 10
C and E from negative exponential distribution
MATRIX MULTIPLY.BL1
Run # 974 - 982
7 PE's, deallocation policy, maxwait = 21
timein = 42
APR
TBR/10
EPR
T_C
Time and Resource Utilization
25
20
15
10
5
0
10
20
30
40
50
60
70
80
$\bar{E}$

Graph (15)


$\bar{C}$ = 10
C and E from negative exponential distribution
MATRIX MULTIPLY.BL1
Run # 954 - 962
3 PE's
deallocation policy
maxwait = 21 timein = 42
Time and Resource Utilization
APR
EPR
T_C
TBR/100
$\bar{E}$
0
5
10
15
20
25
10
20
30
40
50
60
70
80

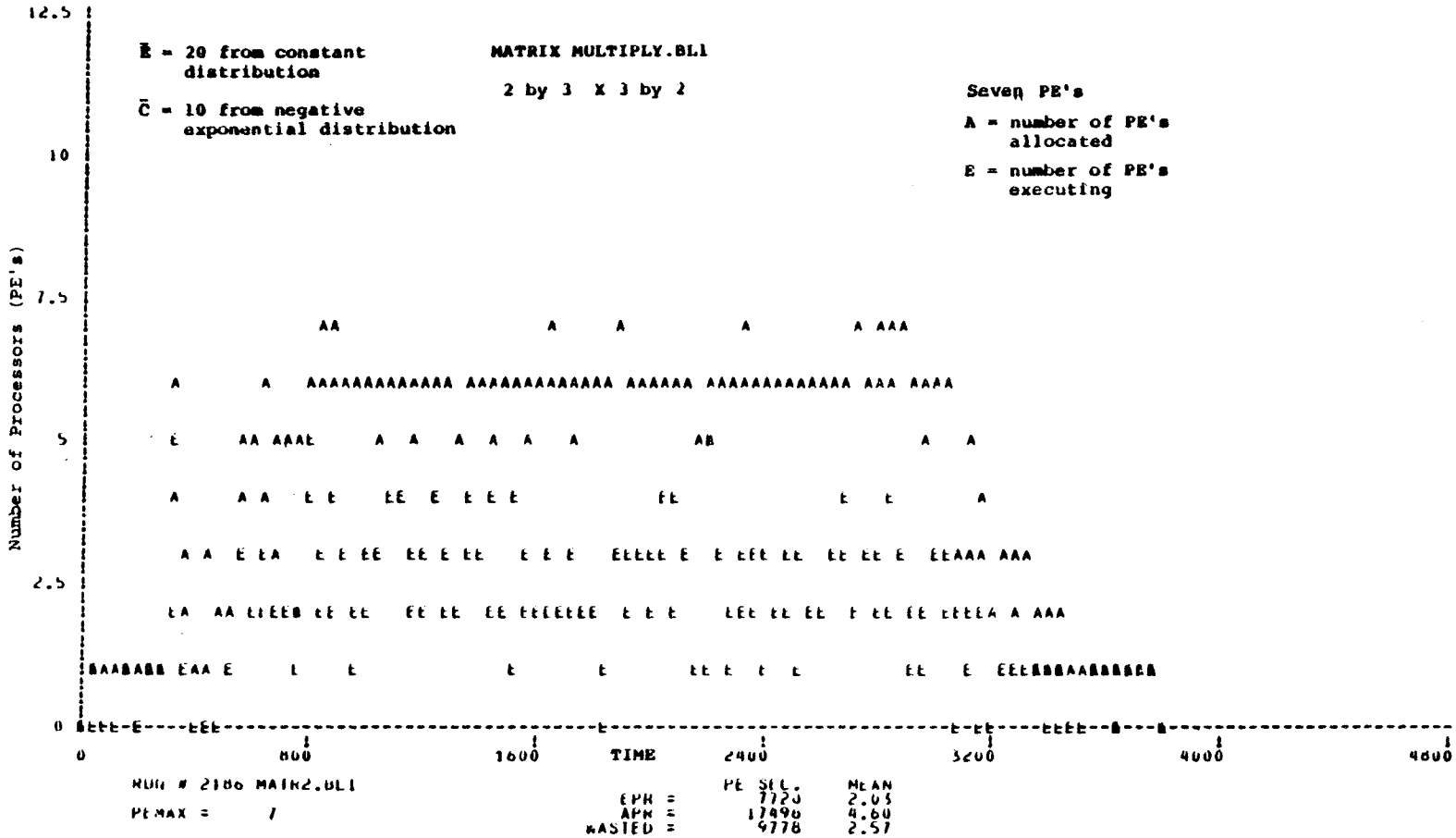
Graph (16)


$\bar{C}$ = 100 $\bar{E}$ = 100
Effects of PE Limitations
Run # 838 - 841
C and E from uniform distribution
MATRIX MULTIPLY.BL1
3 by 4 X 4 by 5
deallocation policy
maxwait=245 timein=490
Time and Resource Utilization
25
20
15
10
5
0
TBR
T_C*10
APR
T_C
10
20
30
40
50
60
70
80
Number of Processors (PE's)

Graph (17)

ALLOCATED AND EXECUTING PE'S

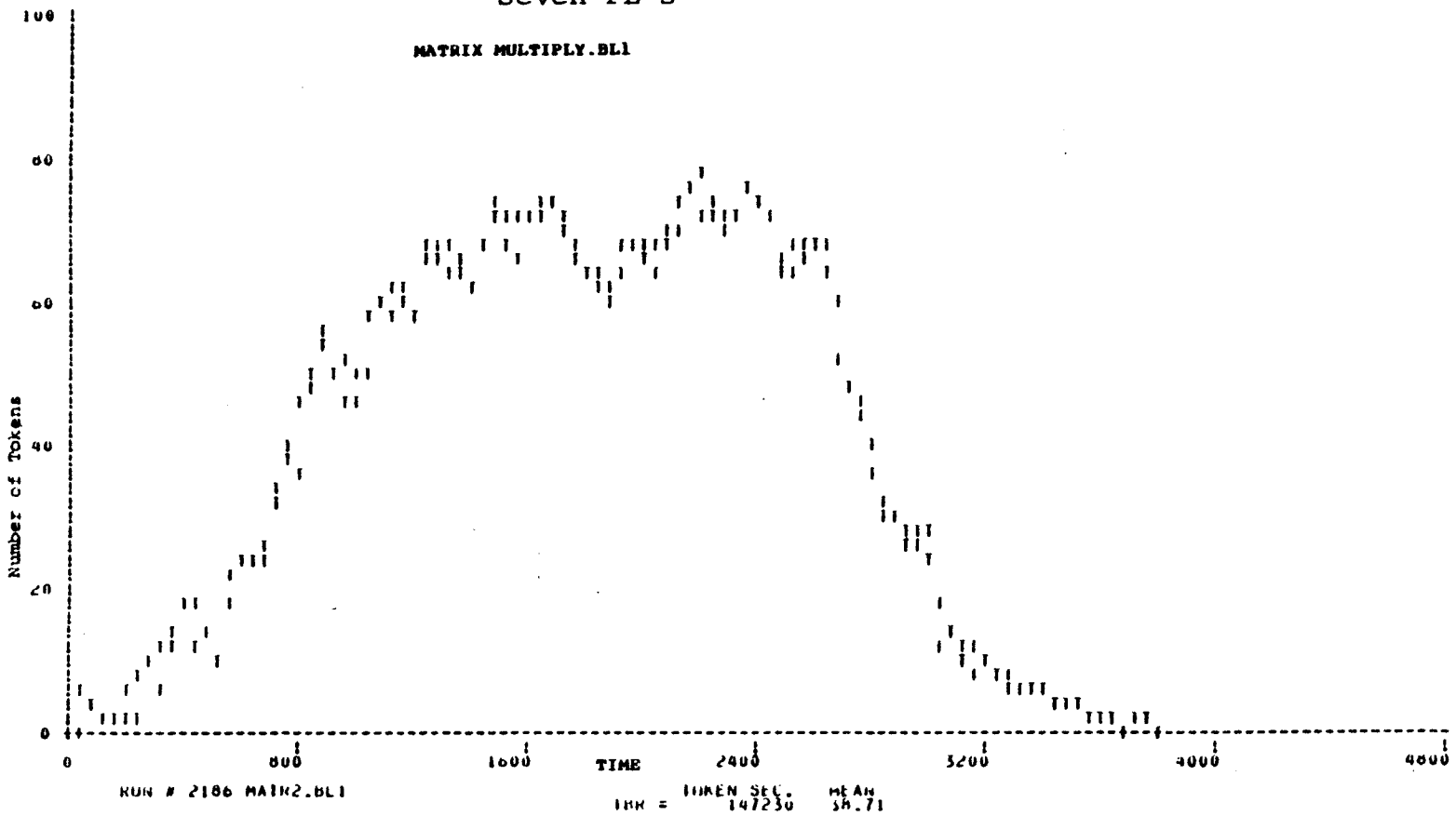
Seven PE's



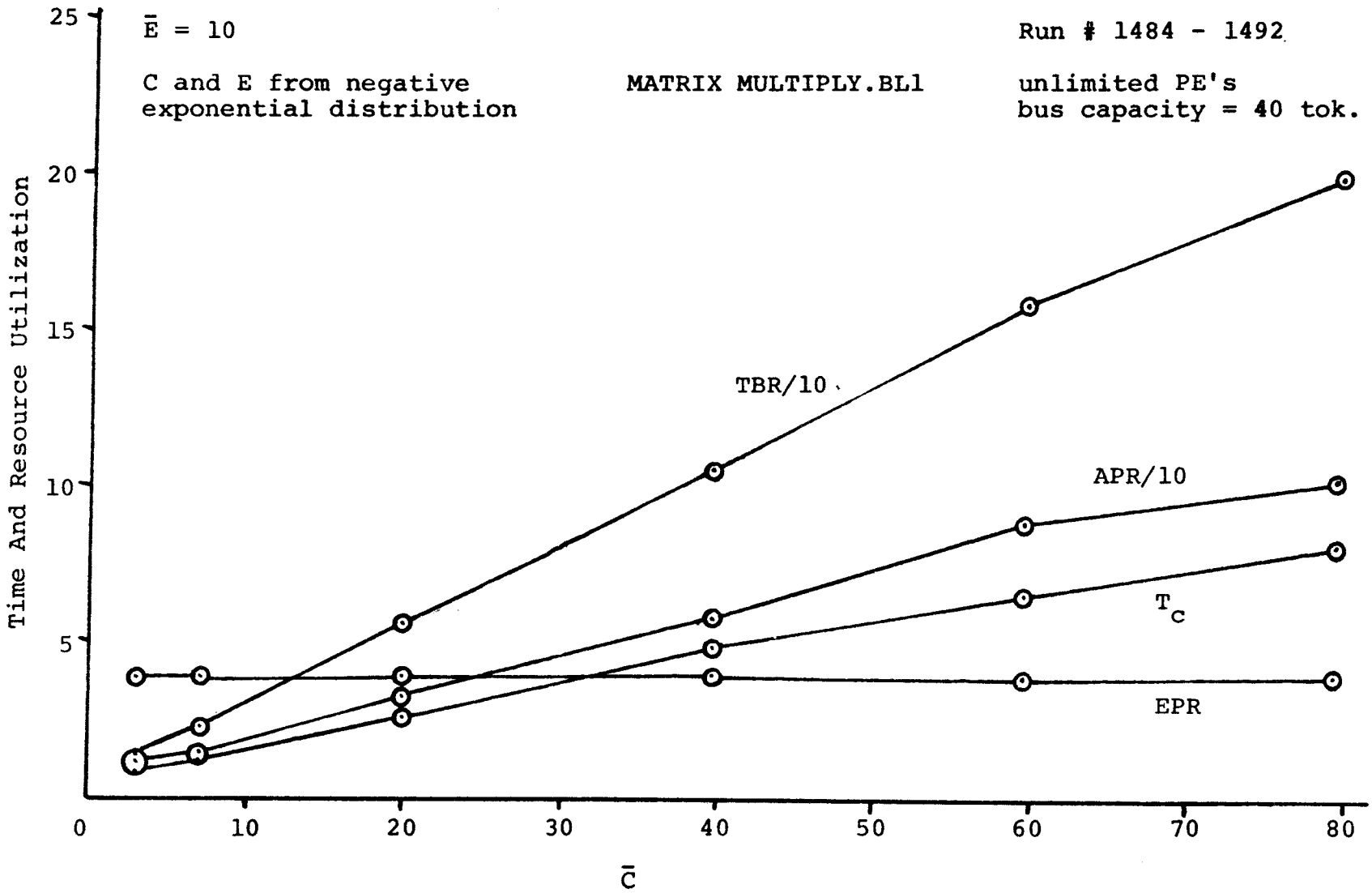
Graph (18)

TOKEN BUS LOAD

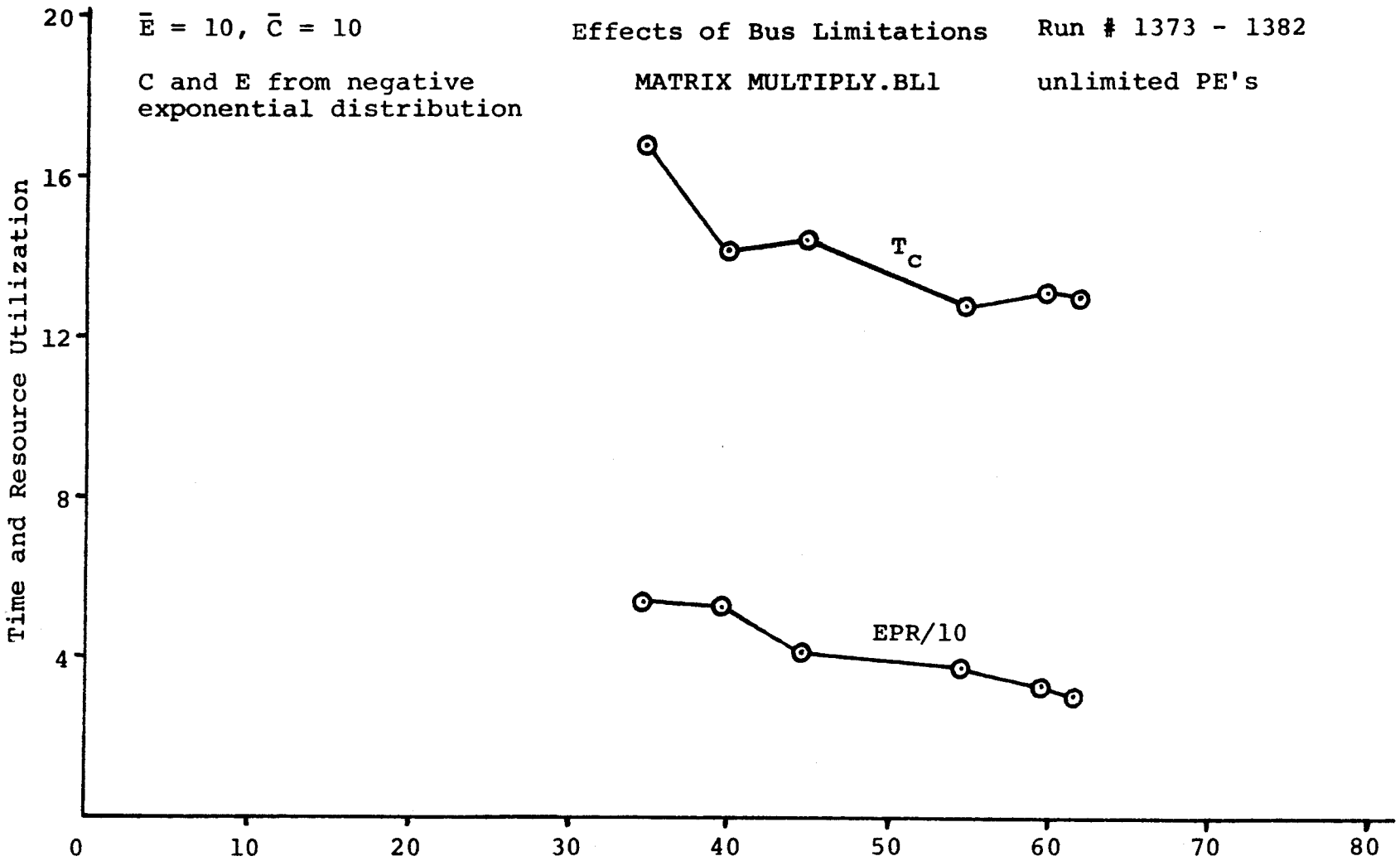
Seven PE's



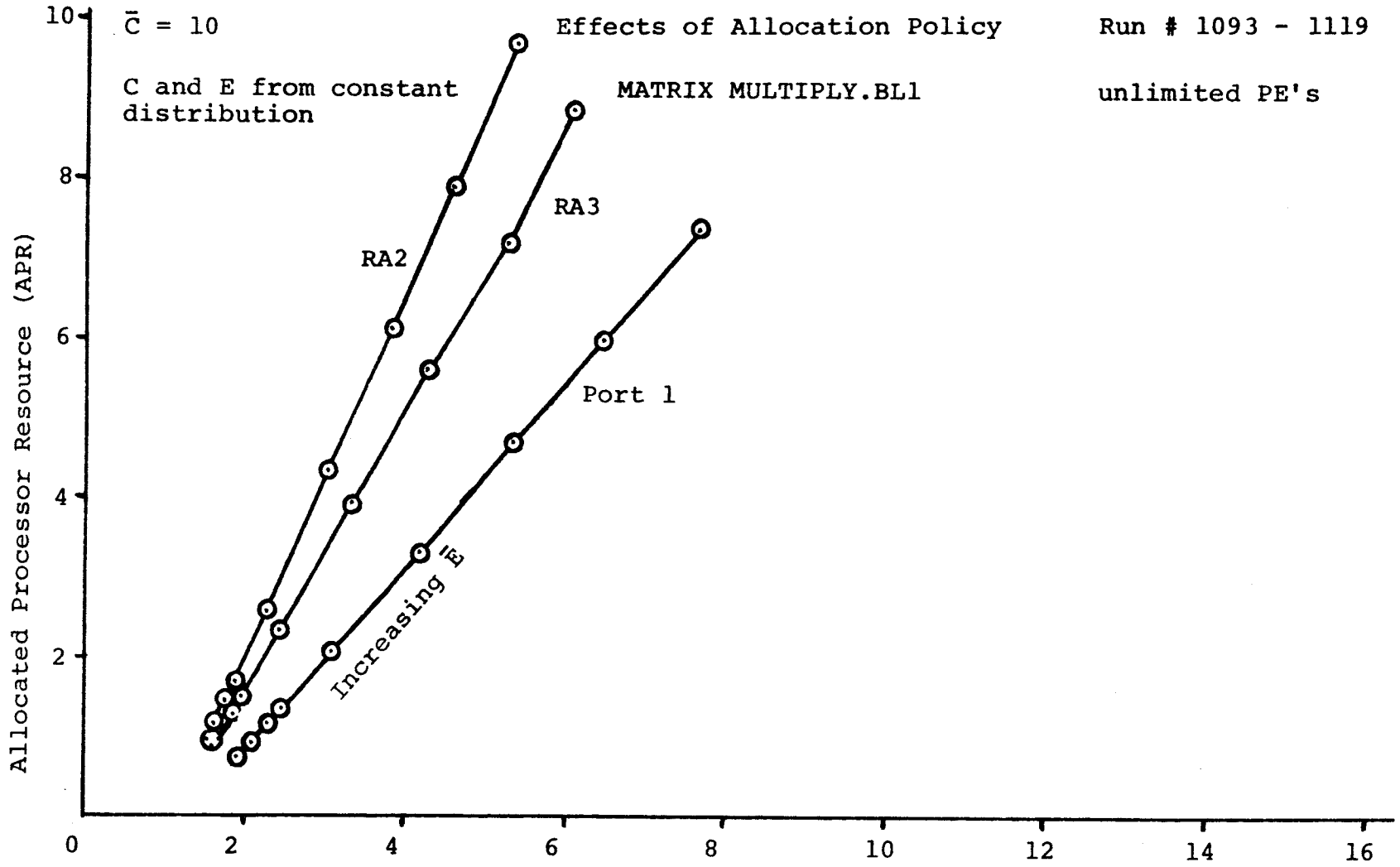
Graph (19)


$\bar{E}$ = 10
C and E from negative exponential distribution
MATRIX MULTIPLY.BL1
Run # 1484 - 1492
unlimited PE's
bus capacity = 40 tok.
Time And Resource Utilization
25
20
15
10
5
0
10
20
30
40
50
60
70
80
$\bar{C}$
TBR/10
APR/10
T_C
EPR

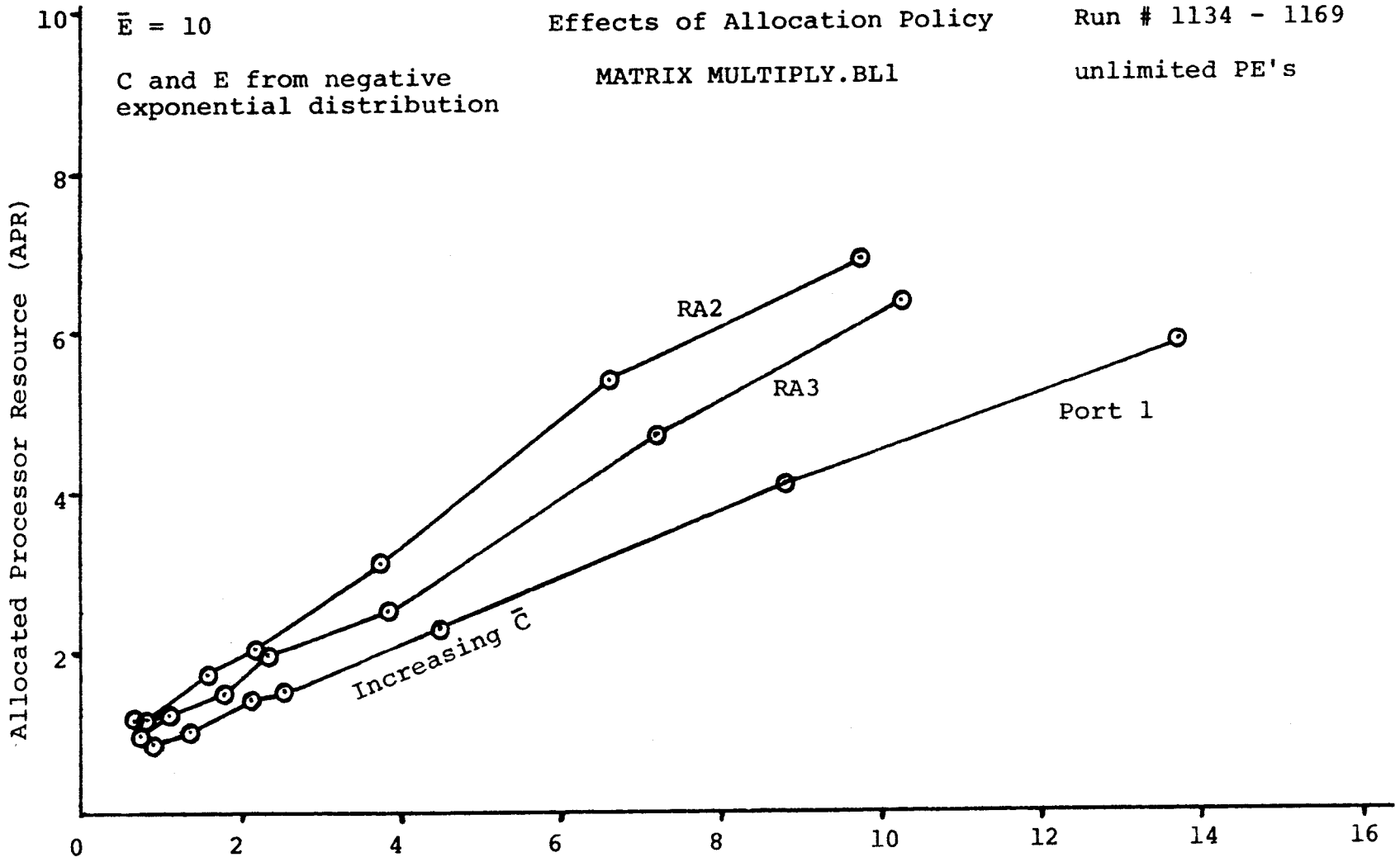
Graph (20)


$\bar{E}$ = 10, $\bar{C}$ = 10
C and E from negative exponential distribution
Effects of Bus Limitations
MATRIX MULTIPLY.BL1
Run # 1373 - 1382
unlimited PE's
T_C
EPR/10
Time and Resource Utilization
20
16
12
8
4
0
10
20
30
40
50
60
70
80
Maximum Capacity of Bus (tokens)

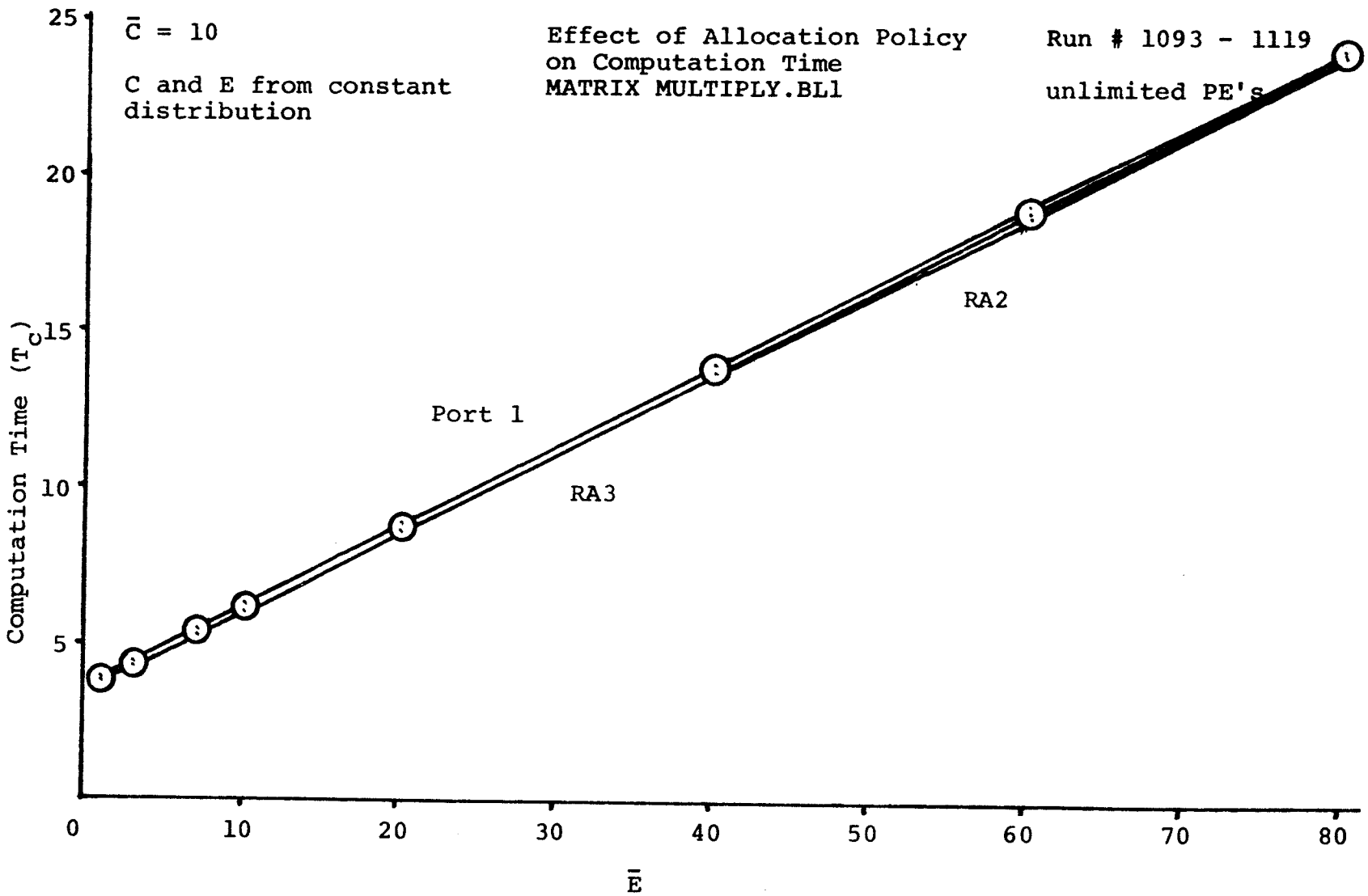
Graph (21)


$\bar{C}$ = 10
C and E from constant distribution
Effects of Allocation Policy
MATRIX MULTIPLY.BL1
Run # 1093 - 1119
unlimited PE's
RA2
RA3
Port 1
Increasing $\bar{E}$
Allocated Processor Resource (APR)
Token Bus Resource (TBR)
0
2
4
6
8
10
12
14
16

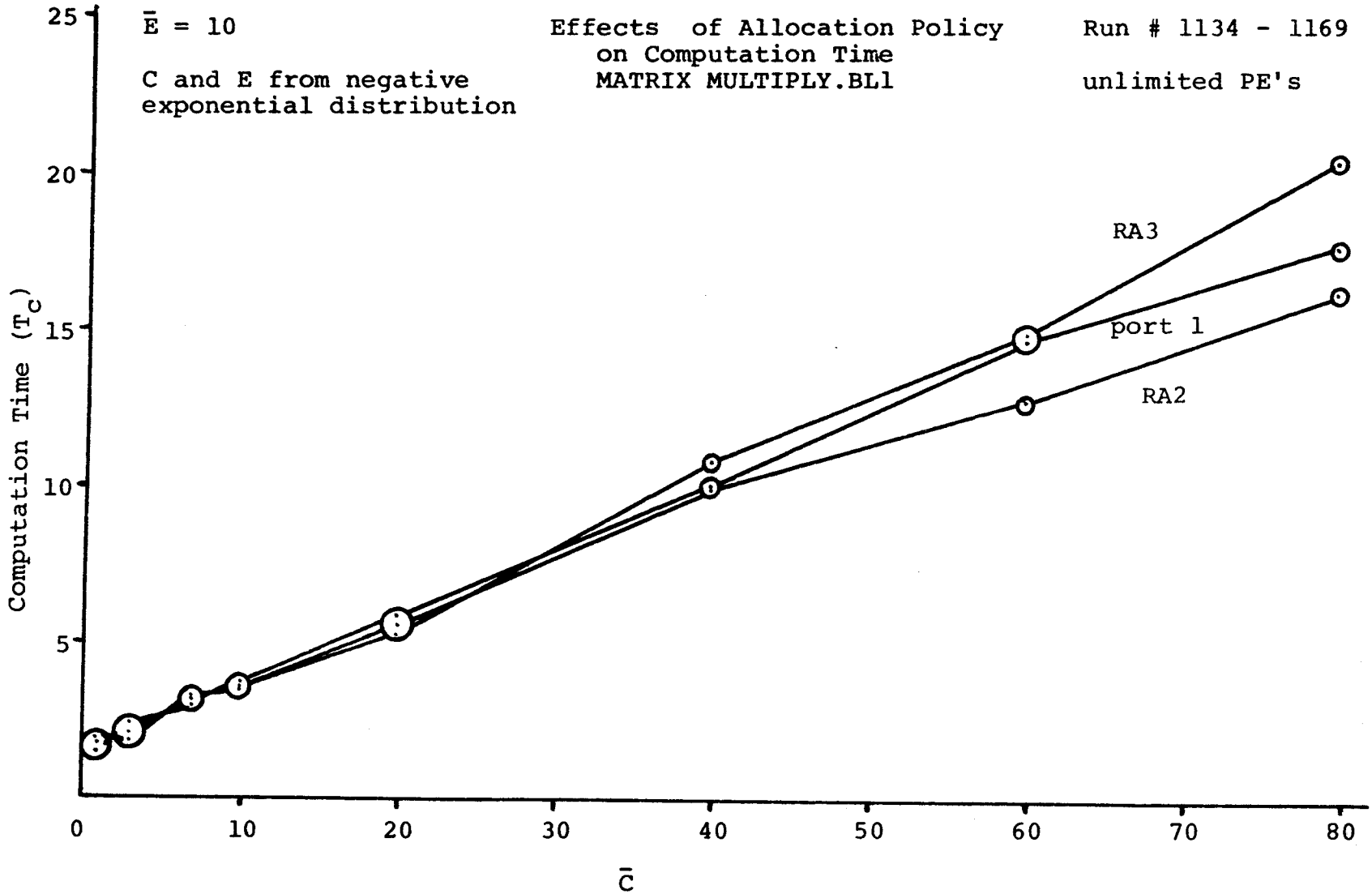
Graph (22)


$\bar{E}$ = 10
C and E from negative exponential distribution
Effects of Allocation Policy
MATRIX MULTIPLY.BL1
Run # 1134 - 1169
unlimited PE's
Allocated Processor Resource (APR)
Token Bus Resource (TBR)
RA2
RA3
Port 1
Increasing $\bar{C}$
0
2
4
6
8
10
12
14
16

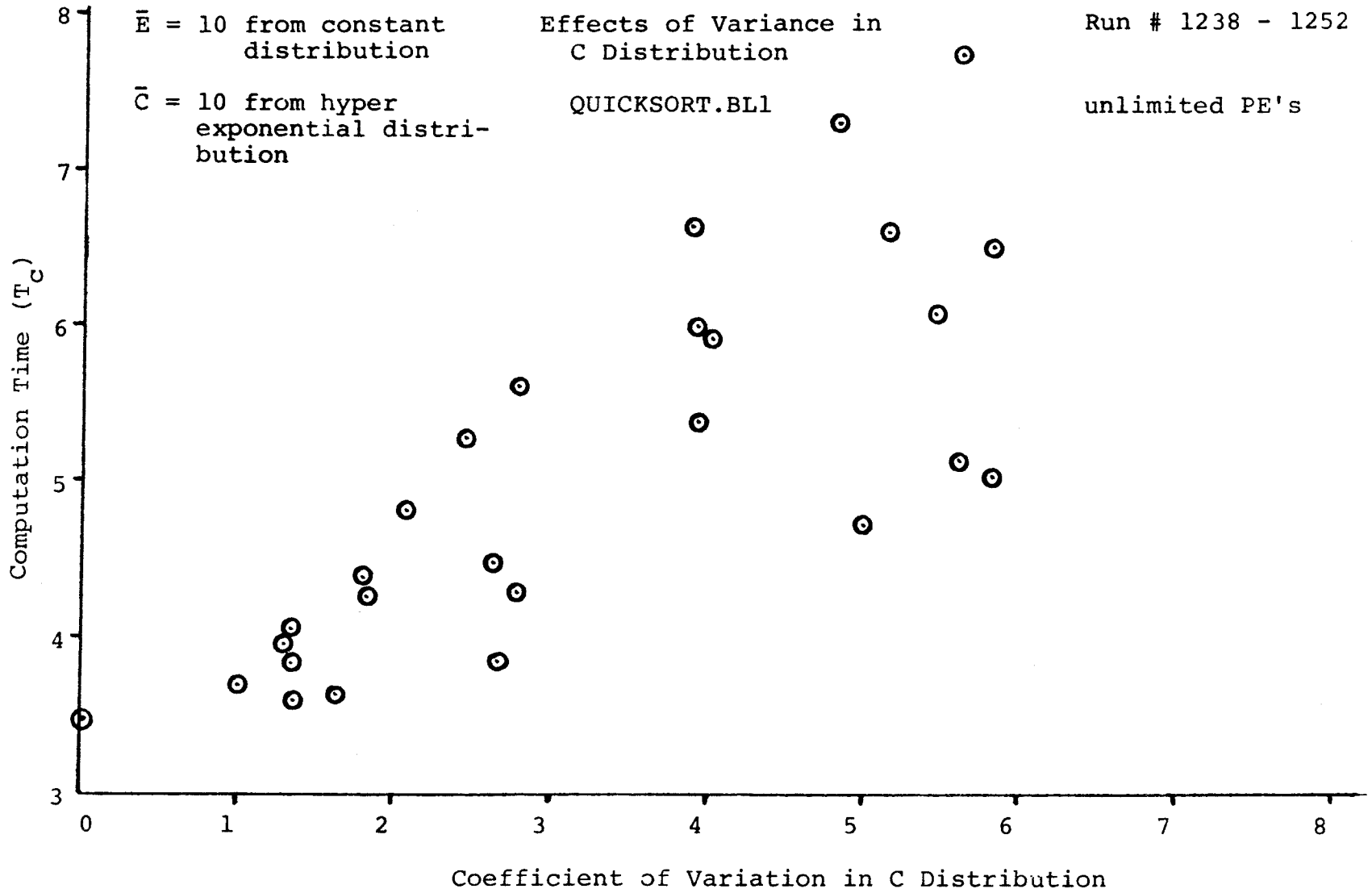
Graph (23)


$\bar{C}$ = 10
C and E from constant distribution
Effect of Allocation Policy on Computation Time
MATRIX MULTIPLY.BL1
Run # 1093 - 1119
unlimited PE's
Computation Time (T_C)
25
20
15
10
5
0
10
20
30
40
50
60
70
80
$\bar{E}$
Port 1
RA3
RA2

Graph (24)


$\bar{E}$ = 10
C and E from negative exponential distribution
Effects of Allocation Policy on Computation Time
MATRIX MULTIPLY.BL1
Run # 1134 - 1169
unlimited PE's
RA3
port 1
RA2
Computation Time (T_c)
25
20
15
10
5
0
10
20
30
40
50
60
70
80
$\bar{C}$

Graph (25)


$\bar{E}$ = 10 from constant distribution
$\bar{C}$ = 10 from hyper exponential distribution
Effects of Variance in C Distribution
QUICKSORT.BL1
Run # 1238 - 1252
unlimited PE's
Computation Time (T_C)
Coefficient of Variation in C Distribution
0
1
2
3
4
5
6
7
8
3
4
5
6
7
8

Graph (26)

REFERENCES

[AG75] Arvind, K. P. Gostelow, "A New Interpreter for Data Flow Schemas and Its Implications for Computer Architecture," Tech Report 72, Department of Information and Computer Science, University of California, Irvine, October 1975.

[AG76] Arvind, K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time," Proceedings IFIP Congress '77 , Toronto, Canada, 849-853. Also, Tech Report 77A, Department of Information and Computer Science, University of California, Irvine, January 1976.

[AG77] Arvind, K. P. Gostelow, "Some Relationships Between Asynchronous Interpreters of a Dataflow Language," Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts, St. Andrews, Canada, August 1-5, 1977. Also, Tech Report 88A, University of California, Irvine, August 1977.

[AGP76] Arvind, K. P. Gostelow, W. Plouffe, "Programming in a Viable Data Flow Language," Tech Report 89, Department of Information and Computer Science, University of California, Irvine, August 1976.

[AGP77] Arvind, K. P. Gostelow, W. Plouffe, "Compilation of ID Loops Involving Steams in the New Base Language," Dataflow Note 15, Department of Information and Computer Science, University of California, Irvine, July 1977.

[DA77] Davis, A. L., "The Architecture of DDM1: A Recursively Structured Data Driven Machine," UUCS-77-113, Department of Computer Science, University of Utah, Salt Lake City, October 1977.

[D73] Dennis, J. B., "First Version of a Data Flow Procedure Language," Computation Structures Group Memo 93, Project MAC, MIT, Cambridge, November 1973, Revised May 1975.

[DM75] Dennis, J. B., D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," Proceedings of the Second Annual Symposium on Computer Architecture, January 1975, 126-132. Also, Computation Structures Group Memo 102, Laboratory for Computer Science, MIT, Cambridge, August 1974.

[DN66] Dahl, O.-J., K. Nygaard, "SIMULA - an ALGOL-Based Simulation Language," Communications of the ACM, Volume 9, Number 9, September 1966, 671-678.

[K75] Kleinrock, L., Queueing Systems, Volume I: Theory, John Wiley & Sons, 1975.

[M74] Manna, Z., Mathematical Theory of Computation, (Chapter 5, The Fixpoint Theory of Programming), McGraw-Hill, 1974.

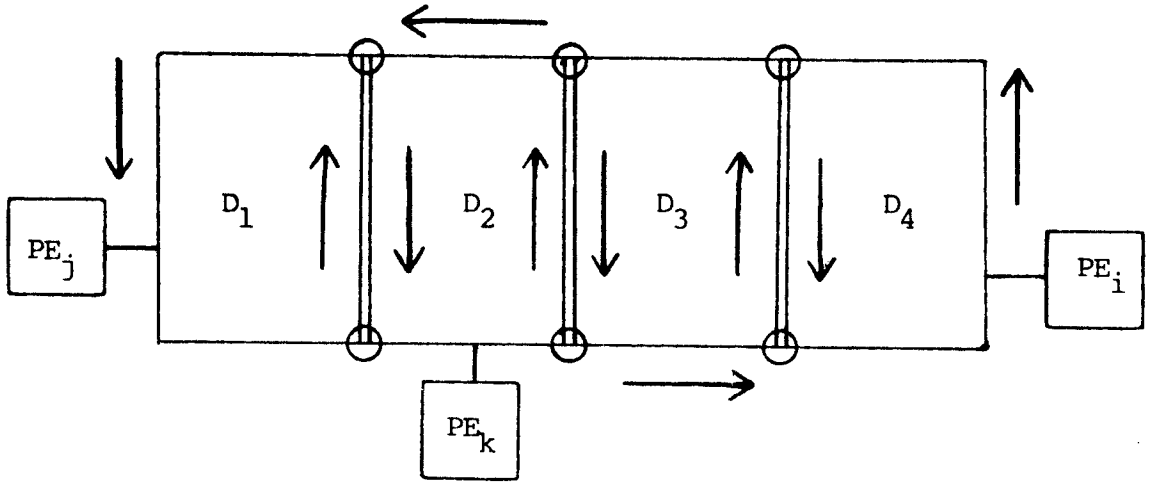
[W75] Weng, K.-S., "Stream-Oriented Computation in Recursive Data Flow Schemas," Technical Memorandum 68, Laboratory for Computer Science, MIT, Cambridge, October 1975.

Appendix I

A Brief Analysis of Logical Communication Delays for an Implementation of a DASADM

The purpose of this analysis is to define more precisely (by example) the meaning of "preallocated" and "overhead" alluded to in the discussion concerning equation (4).

Imagine a bussing structure composed of a number of interconnected rings such as



where the main buses are uni-directional as shown. A number of processors (not all shown) is connected to the bus each by a bi-directional local bus. In the figure, activities j and k are sending tokens to activity i. Let the token from j be an allocation token (activity i does not yet exist) ; this allocation token travels along the bus until it finds an idle PE and then allocates one as activity i. As before this time will be denoted c_{ji}.

The situation for non-allocation tokens, however, is slightly different. The logical address of these tokens is examined (as they pass) by each PE which is waiting for additional operands. Furthermore imagine that the machine is broken into several domains (i.e. D1 - D4 above) where each of these domains may correspond to the instantiation of a dataflow procedure (or loop). The switches at the intersection of busses (circled in the figure) have the power to restrict tokens to a particular domain once they have entered. The structure of the machine is now complex enough to warrant further discussion of the non-allocation communication delay. As before this time will be denoted c_{ki}, but it is useful to describe it as

$$c_{ki} = d_{ki} + w_{ki} + s_{ki}$$

The definition of d_{ki} is split into two parts. In one case, d_{ki} is the time spent from the output of a PE (in this case k) to the end of the destination domain (in the example, once around D4) <u>when the destination is not allocated</u> by the time the token reaches the end of the domain. (This rather strange definition is of little consequence since this time will not appear on the critical path.) Otherwise, d_{ki} is simply the time from PE k to (previously allocated) PE i. w_{ki} is the time spent (in the local domain of the destination) from the first definition of d_{ki} to the moment activity i is allocated. And s_{ki} is

the time spent from the end of w_{ki} to the time the token reaches activity i. In a more general situation, suppose activities k through n are sending non-allocation tokens to activity i. Let all of the above times be determined and recorded by an outside observer for each activity. Then the definition given in (4b) becomes (recall that m is the label of the previous activity such that)

$$t_m + c_{mi} = MAX(t_k + c_{ki}, \ldots, t_n + c_{ni})$$

then

$$t_i = e_i + [\ \underline{IF}\ (w_{mi} = 0)$$
$$\underline{THEN}\ t_m + c_{mi} \quad (\text{here } s_{mi} = 0)$$
$$\underline{ELSE}\ t_j + c_{ji} + s_{mi}\ ]$$

The above discussion indicates that when the waiting time is defined appropriately, a zero waiting time means that the PE is preallocated. In addition s_{mi} is the appropriate overhead incurred when the PE is not preallocated. In this architecture it is clear that the distributions for c_{ji}, c_{mi}, and s_{mi} would all be different and that equation (6) is a suitable equation for describing the performance of the machine.

Appendix II

Critical Path Length and Number of Late Allocations
Matrix Multiply 2 by 3 x 3 by 2
Deterministic C and E

allocation policy	L_{LIP}	From Boundary Cases		From Least Squares $\bar{E}$ as the independent variable		From Least Squares $\bar{C}$ as the independent variable	
		L_{cp} ($\bar{C}$=0)	a_L ($\bar{E}$=0)	L_{cp}	a_L	L_{cp}	a_L
Port 1	53	53	20	53.00	20.00	53.00	20.00
RA2	53	53	18	52.88	19.04	52.88	16.96
RA3	53	53	17	52.88	15.51	51.84	18.04