A COMPUTING STRUCTURE FOR DATA ACQUISITION IN HIGH ENERGY PHYSICS

by

GARRY ALEXANDER LESTER

Submitted for the Degree of

Doctor of Philosophy

at the

University of Salford

in the

Department of Electronic and Electrical Engineering

1988

MCMLXXXVIII

Volume I

To Someone, possibly as yet un-met.

My theory that I have follows the lines that

I am about to relate.....

Anne Elk (Miss) (circa 1970).

Contents

MCMLXXXVIII

ACKNOWLEDGEMENTS

I wish to thank my supervisor Dr C A Owen for allowing me the freedom to approach this research in a largely independant way whilst providing constructive advice when this was sought. I would also wish to thank Dr E C G Owen and his colleagues at Daresbury Laboratory for opening up this avenue of research to me. Finally I would acknowledge Dr R Vaughan-Williams for having written such a splendid set of symphonies which have frequently proved a source of inspiration, particularly the Fourth in F minor.

ABSTRACT

A review of the development of parallel computing is presented, followed by a summary of currently recognised types of parallel computer and a brief summary of some applications of parallel computing in the field of high energy physics.

The computing requirement at the data acquisition stage of a particular set of high energy physics experiments is detailed, with reference to the computing system currently in use. The requirement for a parallel processor to process the data from these experiments is established and a possible computing structure put forward.

The topology proposed consists of a set of rings of processors stacked to give a cylindrical arrangement, an analytical approach is used to verify the suitability and extensibility of the suggested scheme. Using simulation results the behaviour of rings and cylinders of processors using different algorithms for the movement of data within the system and different patterns of data input is presented and discussed.

Practical hardware and software details for processing equipment capable of supporting such a structure as presented here is given, various algorithms for use with this equipment, e.g. program distribution, are developed and the software for the implementation of the cylindrical structure is presented.

Appendices of constructional information and all program listings are included.

CHAPTER 1

1 HISTORICAL REVIEW

1.1 History of Computing

1.1.1 Mechanical Calculating Engines

The earliest automatic computing engines appeared in europe in the early seventeenth century, and were purely mechanical in operation. According to Hayes[64] the earliest of these was one designed and built in 1623 by Wilhelm Schikhard, of far more influence however was the machine built by Blaise Pascal in 1642 though even this later machine was only capable of addition and subtraction. The next notable event was the construction in about 1671 by Gotfried Leibniz of a calculator capable of addition and subtraction in the manner of Pascal's machine and also multiplication and division. These machines were largely regarded as academic curiosities until mechanical calculators were exploited commercially in the 19th century.

The next major figure was Charles Babbage, he proposed two mechanical computers, the now famous Difference Engine (begun in 1823) and Analytical Engine (conceived in 1834). Neither machine was completed for a variety of reasons but probably principally because both machines were very adventurous and required rather better mechanical engineering than was available at the time.

The difference engine of Babbage was conceived for the purpose of automatic computation of astronomical tables using only addition and subtraction by the method of finite differences, hence its name. Babbage had been greatly impressed by the number of errors in manually computed tables and his difference engine was intended to not only calculate the entries of a table but to transfer these results immediately to an engravers plate using steel punches.

The Analytical Engine was conceived as performing any mathematical operation automatically and the final design exhibited many of the main elements of the electronic computers built over a century later. The analytical engine was to have two main parts: *the store*, a memory unit consisting of sets of counter wheels, corresponding to the cathode ray tubes, mercury delay lines and bistable electronic circuits used as memory in electronic computers, and *the mill*, corresponding to a modern arithmetic logic unit being capable of performing the four basic arithmetic functions.

Babbage proposed to control the machine using punched cards not unlike those developed for use with the Jacquard loom. There were to be two sets of cards, *operation cards*, used to control the operation of the mill and *variable cards*, used to select the memory locations to be used as the sources of operands and the destination of results for a particular operation. One of the most significant contributions of Babbage's proposals was a mechanism to allow the sequences of operations to be changed automatically, in modern terms unconditional and conditional branch instructions.

The earliest known reference to parallelism in computer design is thought to be in a *Sketch of the Analytical Engine Invented by Charles Babbage* by General L F Menabrea in 1842 according to Hockney and Jesshope[68]. Though parallelism was not incorporated in Babbage's analytical engine he was clearly aware of the possibility of its application to improve performance over a century before the technology was available to make its large scale application possible. Though neither of Babbage's engines were constructed mechanical four function calculators were in widespread use from early in the nineteenth century until electronic calculators became widely available cheaply in the latter half of the 20th century.

1.1.2 Electro-Mechanical Computing Machines

In the 1930s and 1940s Konrad Zuse built several electro-mechanical computers, apparently unaware of the work of Babbage, Z3 built around 1941 is believed to be the first operational general purpose program controlled computer, however this work was interrupted by the second world war and had little influence on later machines. A more influential machine was the Harvard Mark I (originally called the Automatic Sequence Controlled Calculator) proposed in 1937 by Howard Aiken, which was completed by IBM in 1944.

1.1.3 Electronic Computers

The first purely electronic, as opposed to electro-mechanical, computers appeared in the late 1930s and early 1940s, the first general purpose electronic computer

probably being ENIAC (Electronic Numerical Integrator and Calculator) built at the University of Pennsylvania. Though it weighed 30 tons and contained 18,000 electronic valves it was around 1000 times faster than the Harvard Mark I, taking 3 mS for a 10 digit multiplication.

The first machines employing the stored program concept in which program and data reside together in the same memory unit appeared at the end of the 1940s, with EDSAC at the University of Cambridge and EDVAC at the University of Pennsylvania.

It is interesting that at this time the possibility of using serial arithmetic, with the consequent reduction in hardware required, was regarded as an advantage of the speed of electronic components over mechanical machines which had performed their arithmetic in parallel.

1.1.4 Mechanisms For Introducing Parallelism

Despite the speed advantages of electronic computers over mechanical and electro-mechanical machines greater processing rates were soon required. Though some improvement in processing speed was achieved through improvements in the speed of electronic components major improvements were achieved through parallelism; the fundamental techniques for introducing parallelism are briefly described here before their discussion in a historical context.

There are several techniques for introducing parallelism as listed by Sharp[136] these are; Multiprocessors, Multifunction Processors, Array Processors and Pipeline processing. Each of these techniques involves replication of some or all of the computer architecture at varying levels.

Multiprocessors

Multiprocessors involve duplication of the entire computer, computers making up the machine using some communication scheme to coordinate their actions. A variety of communication schemes are the subject of current research; shared memory, shared bus and direct interprocessor connections for examples. The major problems with multiprocessors are those of communication and synchronisation of the processors and of decomposition of the task into suitable units to be performed in parallel. Multiprocessors are dealt with further in the survey of currently researched processor types in section 1.3.

Multifunction Processors

If units within a processor are replicated such as floating point processors, then it is possible to perform several operations simultaneously within a single cpu.

Processor Arrays

The development of processor arrays is dealt with fully in section 1.1.6. In a processor array a large number of processors are made to perform the same operations

simultaneously but on different sets of data. Usually some mechanism for data exchange between the processors is provided. Though such machines can achieve high processing rates for the large number of problems that can be reformulated for such a processing scheme they are not efficiently applied to all problems. The term array processor frequently used for such computers is not used consistently as is illustrated in section 1.1.7.

Pipeline Processing

Pipeline processing is applicable when long sequences of instructions are to be repeated for different data. A production line type of approach can be used to overlap an instructions execution with the execution of the preceding instruction on the following item of data in the sequence. It is common to combine pipeline processing with one or more other techniques for introducing parallelism.

1.1.5 Scalar and Vector Processors

There are several techniques that have been used to achieve speed up through the implementation of parallelism. The development of parallelism as applied to 'traditional' architectures is presented. This account is based on those by Hockney and Jesshope[68] and Sharp[136].

All parallelism requires some additional hardware, usually in the form of replication of some part of the architecture already present, the level of replication and the way it is used varying depending upon the scheme to be

implemented.

The pilot ACE and the commercial machine derived from this, the English Electric DEUCE, first constructed in 1951 had several features to permit parallel operations to be carried out. The input/output devices (a card reader and card punch) could operate in parallel with the rest of the machine and also instructions were available to operate on all of the words in one of the eleven mercury delay lines with a single instruction, what would now be classed as vector instructions.

During the 1950s several important advances were introduced, represented by the IBM commercial machines of the period which incorporated bit-parallel arithmetic, and later I/O *channels* which were in essence dedicated I/O processors, these were the earliest multiprocessors.

At about this time some consideration was given to large scale multiprocessor designs, however the programming of such systems has several problems and most of the commercial development was towards introducing parallelism into scalar computers (computers operating on data items comprising single values only) to achieve higher computing speeds.

1.1.5.1 Scalar Processors

Multiple functional units allowing arithmetic operations to be performed in parallel and pipelining where stages of an operations execution are overlapped with later stages in a previous operations execution (usually employed in the instruction fetch and decode sequence) became common in

computer architectures of the 1960s. A modern example of a processor employing multiple functional units is the CYBERPLUS[72].

Multiple functional units were usually of arithmetic units, registers and memory, to make use of these in parallel some degree of lookahead was required to determine which operations could be performed in parallel. This lookahead approach allowed the overlapping of instruction decoding, address calculation and fetching of operands using a pipelining technique. A good representative example of a machine using such techniques is the IBM 360/91.

1.1.5.2 Vector Processors

The logical development from scalar machines using pipelining techniques to achieve high computing speeds was the construction of vector processors using pipelining techniques. Vector machines operate on vectors (an ordered group of numerical values) as a basic unit of data. The most famous of these are the CRAY machines, the CRAY-1 having regularly achieved 130 MFlops/sec on appropriate problems.

This is one of the main points about departures from scalar machines in that the problems must be suitable for the machine architecture; vector machines exhibit little, if any improvement over purely scalar processors when performing purely scalar computations. This dependence upon problem suitability is clearly demonstrated by the benchmarks of a CYBER 205 vector processor and a CYBERPLUS scalar processor on a Monte-Carlo crystal growth model involving a large number of

testing and branching instructions[72].

1.1.6 Processor Arrays

In 1962 a paper of Slotnick *et al* described the SOLOMON computer, SOLOMON standing for Simultaneous Operation Linked Ordinal MOdular Network. This was one of the earliest references to the concept of an array of processors, each with some memory, and all under the control of a central control stream. Though never built as originally proposed several important machines developed from this concept, such as the ILLIAC IV and ICL DAP[49]. The ILLIAC IV was not a success, costing four times its contract figure and never coming within an order of magnitude of its proposed performance, when finally operational in 1975, it was however a very influential machine. ILLIAC IV was, like the engines of Babbage a century before, too ambitious for the technology available at the time. The ICL DAP was commercially viable however, the first one being installed in 1980, this machine had in its production form an array of 64 × 64 processors, each with 4096 bits of memory and capable of bit-serial arithmetic on the values held in this memory, 4096 such calculations being performed in parallel. In common with the processors of SOLOMON and ILLIAC IV the processors of the ICL DAP had connections with their nearest neighbours, in an array pattern from which this genre of machines get their name. Though these machines used physical hardware connections to their nearest neighbours an alternative is to use conceptual links, data being passed via common memory. In this case there is no definite structure to the communication pattern between processors, the system being

termed unstructured, the term *ensemble* was used for such an arrangement. A notable example of such a machine is the Burroughs Parallel Element Processor Ensemble PEPE developed in the mid 1970s which had 288 processing elements, each containing three processors (one each for input of radar signals, processing of data and output of control signals) controlled by three control units for the three types of processor within each processing element. When necessary communication between the processing elements took place via the memories of the control units.

1.1.7 Array Processors

Many special purpose computers have been produced for processing large amounts of data, usually in the form of arrays, these tend to be referred to by the generic name *array processors* though their architecture does not necessarily consist of an array of processors. A good example of such processors are the special purpose devices for Fast Fourier Transform (FFT) and similar algorithms frequently used in signal processing applications. A list of the attributes required of a subset of array processors, peripheral array processors, has been suggested by Karplus[82] though the generality of the term array processor is acknowledged.

1.1.8 Orthogonal and Associative Processors

An alternative approach to simultaneously processing all of the bits of a word in parallel is to do the converse and process the same bit of several words in parallel. In the

orthogonal computer described by Shooman in *parallel computing with vertical data* a 'horizontal unit' for word serial/bit parallel operations and a 'vertical unit' for bit serial/word parallel operations were both provided allowing the most appropriate mechanism of referencing the data to be used.

The notion of testing all words in parallel leads to the idea of *associative* processing and *content-addressable* memory in which items are referenced by a match between the data and a given bit pattern or mask rather than by the address of its location in memory. It is usual to provide both associative and address reference in a processing scheme though in a purely associative memory there is no facility to address data by its position in store.

A series of commercial machines under the name of OMEN (Orthogonal Mini EmbedmeNt) were produced in the early 1970s, these used a PDP-11 as the horizontal arithmetic unit and an array of 64 processing elements as the associative vertical arithmetic unit.

Several machines based around the orthogonal computer concept have been built and machines along these lines are the subject of considerable present research.

Recently a large amount of interest has been shown in multiprocessor computers, most commercial machines use only a small number of processors; the CRAY X-MP, regarded by many as a state of the art multiprocessor supercomputer has a maximum of only four processors. Some academic multiprocessors use rather more processing elements and these are dealt with in the

survey of multiprocessor types in section 1.3.

1.2 Classification Schemes

Several schemes have been proposed for the classification of multiprocessor computers, however most of these are inadequate to classify the wide variety of machine types presently recognised.

A natural classification is by the level at which parallelism is implemented within the computer, Hockney and Jesshope[68] divide this up into four levels:

1 - Job Level

2 - Program Level

3 - Instruction Level

4 - Arithmetic and Bit Level

at the job level separate jobs or large sections thereof are regarded as the units to be executed in parallel. Since jobs are usually independant these can be executed in parallel without communication or determinacy problems arising.

At the program level sections of a program that do not exhibit data dependencies may be executed in parallel on separate processors, a good example of this are loop constructs which do not require data exchanges between iterations of the loop.

Pipelining of the stages of instruction fetch and decoding has allowed parallelism at the instruction level to become a commonplace feature of computer architecture.

At the lowest level parallelism between the operations on elements of instruction operands is possible, e.g. bit parallel arithmetic or vector arithmetic.

The machine to be described in this thesis would reside at the top most level of this scheme, the duplication of the program being regarded as separate jobs using different data.

A very simple taxonomy was presented by Crenshaw in a NATO conference paper[48] in which computer systems are regarded as either *'federated'* or *'integrated'*. In this scheme a federated system is one consisting of several computers each performing a particular task and communicating with the other processors through I/O channels. The computers making up a federated computer may themselves be integrated computers and need not be identical. An integrated system is one in which unrelated tasks are performed in a multiprogrammed fashion within a single computer. This computer may be a monoprocessor or a multiprocessor sharing common main storage. The key feature of an integrated computer system is the single job queue and operating system.

In this terminology the computing structure presented in chapter 3 would be regarded as a federated computer made up of a collection of integrated computers (since it is likely that the nodes would be capable of multiprogramming).

One of the most widely quoted classifications is that of Flynn[52,68,136,142]. Rather than describing the architecture of the computer Flynn's taxonomy relates the instructions and the data being processed. Flynn identified four cases of the

relationships between the *instruction stream(s)* and *data stream(s)*:

SISD - Single Instruction stream, Single Data stream

SIMD - Single Instruction stream, Multiple Data stream

MISD - Multiple Instruction stream, Single Data stream

MIMD - Multiple Instruction stream, Multiple Data stream

the first of these represents the serial computer architecture. The SIMD architecture is one in which a single instruction operates on multiple data, a good example being a vector instruction. The MISD case is on first inspection meaningless since it implies that multiple operations are being performed on a single data item simultaneously, however in a later paper Flynn[51] suggest that special streaming techniques, such as the pipeline where different instructions are applied to the data stream as it passes through the machine, are included in this group. The final group, the MIMD machines, includes all multiprocessor configurations, the lack of distinction between different types of multiprocessor being the main deficiency of Flynn's taxonomy. The system presented in chapter 3 falls in this MIMD category.

A classification based on the organisation of the computer from its constituent parts was provided by Shore[138,68,136] in 1973. Shore's classification provides six cases of processor:

I - Word serial, Bit parallel

II - Word parallel, Bit serial

III - Orthogonal computer (Bit parallel and/or Bit slice)

IV - Unconnected array

V - Connected array

VI - Logic in memory array

type I is the traditional serial computer architecture and type II is a bit slice processor. The orthogonal computer, type III is effectively a combination of types I an II being able to access data in two perpendicular directions. Type IV is an array of unconnected processors under the control of a single control unit, type V is similar except that the processors are connected to permit communication. Type V, the final type consists of memory with processing units distributed throughout it as in associative processors. This classification does not adequately cover loosely coupled processors such as direct connection network computers[87] of which the structure presented in chapter 3 is an example, these being regarded as multicomputers rather than a multiprocessor.

Hockney and Jesshope[68] propose a structural notation not unlike that used by chemists to indicate chemical formulae as a means of expressing computer architecture. This provides a comprehensive scheme for description of computer architectures which is then used in the presentation of a computer taxonomy as a set of decision trees. Zakharov[161] is critical of this scheme as being too cumbersome to be useful, also, as in the scheme of Shore, network computers are not included in the classification.

Sharp[136] provides four cases as an extension of the scheme of Flynn, these are:

SES - Single Processor, Scalar Data

SEA - Single Processor, Array Data

MES - Multiple Processors, Scalar Data

MEA - Multiple Processors, Array Data

the last two of these being subdivisions of Flynn's MIMD group.

Other schemes have been proposed by both Kuck and Schwartz, the scheme presented by Kuck is an extension of Flynn's taxonomy by the addition of an execution stream providing for 16 system types in all and that of Schwartz provides a taxonomic table based on 55 designs.

None of the schemes are entirely satisfactory, since, as Hockney and Jesshope state, it is quite possible for computers to have characteristics which belong in more than one section of the classification.

1.3 Current Computer Research

1.3.1 Electronic Parallel Processors

A large number of parallel processing architectures are the subject of ongoing research, most of these involve some form of multiprocessor. Several authors have surveyed the machines and architectures investigated[135,127,104,2,142,82,65]. Implementation of entire machines in VLSI has been the subject of considerable research and an influential factor in the selection of many architectures such as repeated processor and switch units which can be configured by appropriate switch settings as in the CHiP

architecture[140], VLSI array processors[146] and various others[133].

1.3.1.1 Direct Networks

In direct networks a number of processors (nodes), each with its own independant memory are connected to some (or in the case of full connection, all) of the other processors which comprise the system by a dedicated communication mechanism (links). These networks are static since the connection pattern is unchanging unlike indirect networks described in section 1.3.1.2. There is an enormous variety in the connection schemes currently being considered, the schemes usually exhibit regularity and some of the more commonly investigated schemes are the ring (an extension of which, the cylinder, is used as the connection scheme in chapter 3), tree, mesh (toroidal mesh), hypercube and shuffle-exchange networks[133,50]. Such networks are easily constructed from identical processing nodes, such as the DIRMU multiprocessor node[62] or the MDP[33] or the TRANSPUTER[11,102,156,148,155,78,101], though in such a case it is desirable for all nodes to have the same degree (number of links) and for this to remain constant regardless of the size to which the machine is expanded. This increase in the degree of nodes with size is one of the main disadvantages of one of the most widely investigated topologies, the hypercube[113,147,115]. A closely related architecture which overcomes this difficulty are the cube connected cycles and extended cube connected cycles topologies[133,50], in which the processors at the vertices of the hypercube are replaced by

cycles (rings) of processors; such a topology can be realised with processors of degree three.

Completely connected machines are rarely encountered owing to the high degree required of the processors making up such a system for large numbers of processors. The HPDM[91] has five TRANSPUTERS completely connected and in addition uses shared memory to communicate with a similar number of more conventional CLIPPER processors which are connected together with a parallel bus. The HPDM may be connected to other HPDMs by an ETHERNET or X.25 network.

1.3.1.2 Indirect Networks

Indirect or dynamic networks fall into one of three major classes; single stage, multistage and crossbar, Feng[50] gives a description of such networks and the uses to which they are put. Single stage networks are also called recirculating networks since data may have to be recirculated several times before it reaches its destination. They are used in cycling machines[132], the GF11 supercomputer[17] uses a 3 stage network in a cycling scheme. Multistage and crossbar networks are frequently used to connect processors to memory units, the interconnection network permitting access to any of the memory units by any of the processor units.

The Remps machine[73] users a global network to allow memory sharing between processors but in addition uses two other networks, one to allow I/O communication mapping and the second, a one sided network, to allow interprocessor communication without memory sharing.

One sided networks, referred to occasionally as full switches, allow any of the processors attached to communicate with any of the other processors attached, unlike two sided networks which allow any of the inputs to connect to any of the outputs, Almasi[2] refers to these as the 'boudoir' and 'dancehall' arrangements respectively. The IBM RP3[73] is a current example of the application of a one sided network to interprocess communication.

1.3.1.3 Bus Systems

Bus systems use single or multiple parallel busses to permit data exchange between several units connected to the bus in a random access fashion. This free interconnection of devices as compared with say direct network machines readily allows many conceptual interconnection schemes to be mapped on to the parallel bus. The critical component is largely the bandwidth of the bus, limited by the number of physical connections possible. The general any to any connection possibility of the bus was one of the main reasons for its adoption to connect the vector processors of the MU6V[75]. This general mapping allows bus connections to be applied in a variety of environments such as Message-Passing[126] and Data Flow[150]. This argument also applies to other globally shared medium communications such as those used in Local Area Communications as in the TUMULT ring[130].

Multiple buses lend themselves to a hierarchical scheme, often referred to as a cluster structure[159] since clusters of processors may be grouped around a bus, these busses being

connected by other busses, or, as in the case of MuTeam[30] by serial asynchronous links. The Synapse N+1 system[112], in common with many systems, uses one or more global buses (the synapse expansion bus in the case of synapse) to which are connected lower level buses. The TAMIPS multiprocessor[151] has up to eight processors connected to its local bus, this multiprocessor can then be attached to the multibus (IEEE-796) to provide for expansibility. The Flex/32[100] multicomputer uses a local bus with several processors connected, the processors may connect to other groups of processors or input/output devices to provide for expansibility. There are several busses in use, VME, Multibus (as used in the Sequoia computer[98]), Futurebus and others. FERMTOR[125] uses local buses in a ring, the buses making up the ring being connected by station latches which deal with data transfers between devices not connected to the same bus. By varying the number of buses and number of devices connected to each bus this architecture may be 'tuned' to the data access pattern.

1.3.1.4 Cellular Array Processors

The array processor (SIMD) architecture in which arrays of very simple processors can communicate with adjacent processors is of considerable interest especially with a view towards incorporating large numbers of processors onto a single VLSI device. An example of such a VLSI device is the ITT CAP-II chip[107] which has a 4 × 4 array of 16 processors each working with 16 bit words. Mishin and Sedukhin[106] discuss the behaviour of such an adjacent communication cellular computer system and its performance for a number of problems.

Control for such an array of processors is generated from one central control mechanism with little independant activity by the individual processors. It is possible to transfer some of the centralised control to the individual processors but still maintain an overall central control. In such cases the SIMD nature of the architecture becomes less clear as the control tends towards completely independant operation as in MIMD architectures.

1.3.1.5 Systolic and Wavefront Arrays

The systolic array is a possible example of a halfway house between SIMD and MIMD operation, there being an array of processors, each performing computations under a global scheme of synchronisation and control but with each processor performing a distinct function not necessarily the same as that of the other processors. In the systolic array[132] an array of processors synchronously read input data from their immediate neighbours, performing some computation on this data and writing the outputs to their neighbours. Such arrays are termed *systolic arrays* because the way the data flows within them is reminiscent of a heartbeat. Normally data such as matrices are fed into one or more parts of the array, the flows of data and intermediate results through the array interacting so as to produce the desired result which appears from the outputs of some or all of the processors.

The array need not be planar and may possibly contain closed loops or be switched dynamically as in the nonplanar array of Aravena and Porter[10]. Unlike most systolic arrays

which do not store values but process values and output the results the OCSAMO systolic array[1] uses internal registers to allow values to be held and used in later stages of the computation. The Warp computer[5] has demonstrated practical application of a systolic array to a variety of problems.

Closely related to the systolic array is the wavefront array, this is an asynchronous version of the systolic array. The wavefront array is made up of an array of processors connected as in the systolic approach but in the case of a wavefront processor the processors operate in an asynchronous data driven fashion[85], this has been expressed by Kung *et al*[89] as :

Wavefront Array = Systolic Array
+ Dataflow Computing

the processors in the wavefront array only performing actions when all of the required data is present at its inputs. The name wavefront derives from the way data propagates through the array in waves relating to each group of data items supplied.

1.3.1.6 Data Driven and Demand Driven Computing

The dataflow[40,111] model of computing and possible machines using this model have received considerable attention. In the dataflow or data driven model of computing an instruction (actor) is executed when all of its required operands are available, the dataflow program consists of a flow graph with actors on the nodes and data items flowing over the

arcs. Data driven operation is asynchronous in that the data is passed on to the next actor as soon as the result is produced. The data driven concept is applicable at any level of grain, the machine presented in this thesis possibly being regarded as a dataflow machine with the grain level being the entire processing required rather like a large grain parallel codeblock data flow scheme[23]. A variety of machines have been proposed and built[153,61,25,150,46] and have now reached the stage of a commercial product[149,81].

The principle divisions within dataflow computers are those of static and dynamic. In a static machine such as the HDFM (Hughes Data Flow Machine)[24] the dataflow graph is mapped onto one or more processors in a fixed (static) pattern, each processor being active only when one of the dataflow actors it has been allocated is active. In a dynamic machine such as the Manchester Data Flow Machine[61] any processor may deal with any actor that is ready for execution.

Dataflow has been modified and combined with other schemes such as the combination of control and dataflow of Maeng and Cho[96] and the suggestion by Sowa[141] that performance of a dataflow multiprocessor may be improved by using a program counter in a more traditional type of approach for the serial parts of dataflow computations.

The demand driven model is the converse of data driven, the computation begin broken down into a similar flow graph but the execution of an instruction being initiated when its result is requested rather than when its operands become available. This request for results triggers requests for arguments on the

instruction input arcs, requests being passed back as necessary until data is available.

1.3.1.7 Other Schemes

There are many other multiprocessor architectures, some are combinations of mechanisms described above such as the OPSILA computer[12,13] which uses a vector processor combined with and driven by a scalar processor, many other architectures can be regarded as extensions of those presented above; in the logic architecture where a certain goal is unified with particular definitions the request of a goal initiates a search for the definitions necessary to obtain the results in a manner similar to that in demand flow[65], the processing being carried out by a Parallel Inference Engine[66].

1.3.2 Digital Optical Computers

In contrast to computing using the passage of electrons through conductors computing machines have been proposed and constructed which operate using the transmission of photons, usually in the form of laser light. Both digital and analogue computers have been designed[128] though of particular current interest are digital optical computers. Lohmann[95] lists four principle motivations for the development of digital optical computers:

1 - Optical Subsystems

2 - Very Fast Optical Gates

3 - Immunity against electromagnetic interference

4 - Highly parallel processing with global interconnections

and of these the last has been expressed as being of greatest importance by Wherret[154].

A variety of schemes have been proposed to perform processing on information expressed as patterns of light intensities, Ichioka and Tanidal[76] describe a system of overlapping shadow patterns that allow the sixteen possible logical functions of the pixels of two binary patterns to be generated and many systems employing some form of spatial light modulator (SLM) have been described, an SLM being essentially a mask programmable by incident light intensity allowing a wide variety of functions to be performed on two-dimensional data (a useful comparison of commercially available SLMs is to be found in [18]). Non-linear optical devices (those in which the transmissive properties of the device vary with the applied light intensity) can be used to perform logical operations if the device has a threshold in the output/input transmission characteristic[71] and the combination of a non-linear element with positive feedback may be used to create bi-stable elements such as the Fabry-Perot cavity[18]. These bi-stable elements can be used as latches and to create state machines operating on a large number of pixels simultaneously.

The ability to perform operations on two-dimensional data provides the highly parallel processing referred to above and holographic techniques permit optical interconnection or coordinate transforms of this two-dimensional data on an any to any basis without the problems of physical siting and crosstalk of comparable electronic interconnections.

The advantages of optical connections in terms of bandwidth and the removal of the CR time constant limitation inherent in any electronic connection along with the global interconnection possibilities and non-interference of overlapping communication paths has stimulated investigation of hybrids of electronic circuits and optical interconnections; Goodman *et al*[59] discuss the possibility of using integrated optics and/or fibres for the distribution of clock signals and the use of holographic elements for global interconnections in a VLSI environment and Bell[18] describes various interprocessor connection schemes using optics, including an optical crossbar switch.

Interfacing to an optical computer should be immediately feasible since optical storage and communication methods are already well established and electronically controlled optical switches are available, using materials which change their refractive index depending upon the strength of an applied magnetic field[123].

1.3.3 Biological Computers

Some proposals and discussion of computers based on biological materials have appeared in the literature[29]. None of these have proceeded further than the stage of discussion and it has been suggested[123] that a biological computer already exists in the form of the human brain and that any biological computers are likely to share the weaknesses of it that have inspired the development of other computing mechanisms.

1.4 Computers in Experimental High Energy Physics

Both theoretical and experimental physics research are making greater and greater demands of computer processing power, particularly in the field of High Energy Physics (HEP)[31,162] for a variety of purposes. Mount[109] illustrates the need for parallel processing techniques in HEP particularly with respect to vector and pipeline processors and discusses software details for such machines and Kunz[90] provides a brief resume of vector and parallel processing applied to HEP. A multimicroprocessor suitable for computationally intensive theoretical physics is presented by Christ and Terrano[26], the processor being made up of Intel 80286/287 microprocessors and floating point vector processors in a planar array. Computers have also found a wide variety of uses in experimental work, the systems briefly surveyed here will be concerned mainly with data acquisition, data acquisition in this case being taken to include some degree of pre-processing.

Computers in experimental High Energy Physics serve functions of both control and data acquisition though the two roles are not distinct since the same communication paths are frequently used for both, and acquired data is often used to make control adjustments (ie feedback).

Most control and data acquisition systems incorporate a selection of microprocessors and minicomputers in a networked scheme[19,9] though some complex control systems, such as those used for fusion experiments, require the use of a multiprocessor[84]. Many experiments require real time processing for control and data acquisition with the ability to respond to asynchronous interrupts quickly, hierarchical structures utilising a central controller interacting with and distributing work to sub-processors have been presented for this type of work[6,124].

Frequently encountered in High Energy Physics are systems utilising high speed front end electronics (for coincidence detection, thresholding etc) followed by successive levels of processing to filter out unwanted or 'noise' events[7,131,28], the data acquisition system described in chapter2 for Daresbury Laboratory falls into this group.

Though many theoretical HEP computing tasks may be efficiently vectorised experimental computing does not often run efficiently on vector or pipelined machines; however much of the processing is of totally independant 'events' which can be readily processed in parallel on separate processors leading to the concept of processor *farms*[108]. These farms take the form of several identical processors, an events being passed to

the next available processor for processing, the structure described in the following chapters is a structure to allow a multimicroprocessor to be used in the fashion of a processor farm.

CHAPTER 2

2 COMPUTING REQUIREMENT

2.1 Introduction

The computing structure that forms the basis of this thesis was principally intended to process experimental data from experiments in High Energy Physics, a field that in both its theoretical and experimental requirements demands ever increasing amounts of computation, as has been noted by Creutz[31]. Though currently available computing hardware can achieve very high rates of data processing the large volumes of data being created from experiments in HEP take long periods of 'number crunching' after the experiment has taken place. This situation is far from ideal since many of the experiments performed require run time adjustments to be made in the light of the data obtained and since the experiments are inherently complex and the detectors used are rather fragile immediate feedback of any failures, indicated by the change in the processed output data, would be highly desirable.

2.2 Present Data Collection System

The niche in which the computing system had to reside is illustrated by a brief description of the present computing facilities used for data acquisition for the NSF at Daresbury Laboratory. The following is a brief summary of the facilities for data acquisition, more detailed information being available in the relevant manuals[38,34,36,37,35]. The main elements of

the data acquisition and processing system at the NSF at Daresbury Laboratory are shown in fig 2.1.

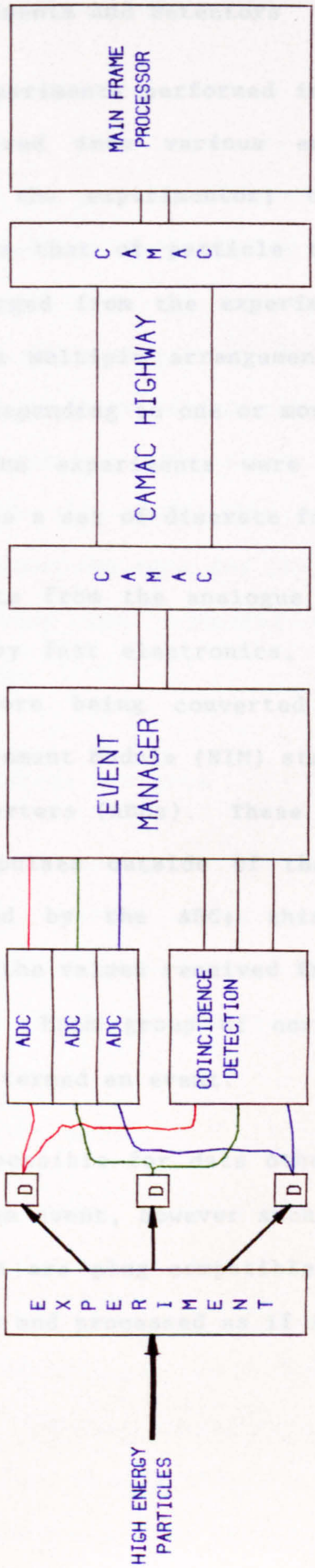


FIG. 2-1 DARESBURY DATA ACQUISITION SYSTEM.

2.2.1 Experiments and Detectors

The experiments performed involved high energy beams of particles fired into various experimental arrangements as required by the experimentor; the principle type of data collected was that of particle type and energy as reaction products emerged from the experiment, using detectors either singly or in multiple arrangements[60,137] to give analogue signals corresponding to one or more parameters of the particle detected. The experiments were run on a continuous basis rather than as a set of discrete firings.

The data from the analogue detectors is correlated for coincidence by fast electronics, non-correlated signals being ignored, before being converted into digital format using Nuclear Instrument Module (NIM) standard compatible analogue to digital converters (ADCs). These ADCs permit analogue windows to be set, pulses outside of the specified range of values being ignored by the ADC; this being the only hardware filtering of the values received from the ADCs available to the experimentor. Each group of correlated data are considered together and termed an event.

It is possible for data other than that from ADCs to be included in an event, however such data can only be taken from devices which are plug compatible with the ADCs and the data will be taken and processed as if it were from an ADC.

2.2.2 The Event Manager

The event manager (EM) is based around a state machine, programmable to some extent by changing values in internal registers, it manages the collection of data from the ADCs, the data being read out individually or as groups, depending upon the experimental set up and operating mode of the event manager. The EM packages up the data and passes this via a CAMAC network to a mainframe computer for storage processing, analysis and display. The event manager does no mathematical processing of the data only acting as an 'intelligent' interface between the ADCs and the CAMAC network, it is capable of some masking and vectoring of data depending upon the operating mode.

2.2.2.1 Operating Features

The event manager offers three principle operating schemes, all schemes require fast coincidence detection electronics to be provided by the experimentor to determine when an event of possible interest has occurred.

2.2.2.2 Singles Mode

In the first operating scheme, 'singles' mode, a histogram of the values detected is produced for each of the ADCs connected to the event manager by a simple increment of the position in the spectrum indexed by the value. This is the simplest and fastest scheme since it requires virtually no processing other than a simple comparison of range of value, which is performed by the ADCs before conversion into digital

representation and therefore little improvement could be anticipated by the addition of computing hardware.

2.2.2.3 Biparameter Interface

The second scheme is an extension of singles mode in that an increment of a histogram is generated for each event. In this case however two of the values are used together to generate the increment, one of the values is used as an index to indicate which histogram to update. The principle use for this mode is to handle Mass-Energy events, and the description of this mode is in terms of such use. Two ADCs are used with each biparameter interface, it being possible to use more than one biparameter interface in an experimental set up, the two ADCs used for the interface must be determined to be coincident by the experimentors fast front end electronics.

The mass ADC value is used to index a lookup table, the entry in the lookup table being a 'not interested' code or a 'mass window' number, each mass window representing a spectrum, within which the energy ADC value is then incremented. Up to 16 such mass windows (spectra) are permitted and it is also possible to set up upper and lower windows on the energy spectra with events having values falling outside these windows being rejected.

As with singles mode, no processing other than that performed by the hardware is required and no real improvement could be achieved by the addition of computing hardware.

2.2.2.4 Multiparameter Event Handling

Multiparameter events are the most general form of data dealt with by the event manager, each of the above schemes are merely a hardware implementation of a frequently used interpretation of the data of a multiparameter event. When a group of ADCs are determined to have fired in a pattern corresponding to a multiparameter event of interest, this being done by the experimentors fast front end electronics, the event manager takes the values from the ADCs and records these as a group, termed an event, which indicates their mutual association. It is possible for an ADC to take part in a multiparameter event and singles mode as part of the same experimental setup.

These events are passed, via the CAMAC network, to a mainframe computer for processing, analysis and display. No processing whatsoever is applied to the data by the event manager, raw data being passed back to the mainframe computer.

At the mainframe computer only a small proportion of the events may be processed as they are obtained from the event manager ($\simeq$10%) the processing of the majority of events being performed offline. The small number of events processed online are used to provide some immediate feedback of the behaviour of the experiment, though due to the small percentage possible this feedback is slow to respond to changes and is somewhat limited in its usefulness.

For this mode of operation on line processing of the data would allow all of the data to be processed and displayed to provide more responsive feedback to the experimentor and also would prevent unnecessary taking and storage of any data not regarded as interesting. This mode operation was the one for which the processing scheme presented in this thesis was intended.

2.2.3 Limitations and Bottlenecks

The first two of these schemes are able to provide the required processing of the data at an adequate data rate, what little processing is required being performed by the event manager along with data acquisition. The use of the event manager to collect multiparameter data leaves the data unprocessed and varying amounts of processing time are required depending upon the experiment, for many experiments this processing is a bottleneck. The aim of the work included in this thesis was to produce a design for a computing device capable of being used in the above environment to process data from the High Energy Physics experiments at a suitably high data rate. This 'computing engine' would be placed between the event manager and the mainframe computer and would be required to pass through data not requiring processing (in the modes handled sufficiently well by the event manager alone) or behave as a 'data filter' on the basis of some user defined function for those operating modes where data is not processed by the event manager.

2.2.3.1 Current Data Rates

The current rate of data taking is in the order of 50 000 events per second, where an event represents all the data from the ADCs that were determined to be part of the event by the coincidence electronics. With the present event manager up to 64 analogue to digital converters can be involved in any one event and the computing and communication structure must be able to cope with this volume of data, with a possibility of extension in the future.

2.3 Design Aims

Although the computing engine to be developed was aimed at a research environment and therefore the main emphasis in the design was one of flexibility and expansibility it was possible to set some initial targets and consider the feasibility of achieving these. It also had to be noted that since the event manager was a potential bottleneck this could be subject to replacement, possibly with some form of multiprocessor.

A throughput of 100 000 events per second was set as initial target figure since this was the region of the limit of the current event manager and ADCs. In normal operation the filtering function applied to the data is set up before data taking commences and left unchanged during the experimental run, in some experiments tailoring of parameters of this function is required in the light of collected data but this is performed in an iterative 'stop and restart' fashion rather than dynamically. Considering this pattern of usage suggested

some form solution using a look up table in the interests of processing speed, the look up table being set up at the beginning of each experimental run.

2.4 Processing by Look-Up Table

A look up table of results for various input parameters could be calculated before the commencement of data taking and thereafter used to process data. This would allow a result to be obtained in a single memory access, a time period in the reqion of 200 nS. In addition to evaluation of a function masking, indexing and various other windowing and logical operations on the data are normally required, these could be included in a complete mapping of the input data values using a look up table.

2.4.1 Estimation of Memory Requirements

A simple calculation of the memory required, however, soon indicates that this approach can make unreasonable demands of memory size. For data of 8 bits in length and a function of two parameters producing an 8 bit result requires a memory size of 64 KBytes, in practice most of the data collected is of greater than 8 bits and the possibility of evaluating functions with more than two parameters was regarded as desirable. For 12 bit data and a 3 parameter function giving an 8 bit result requires a memory size of 68 000 M bytes which is obviously unrealistic even considering the recent advances in memory devices[77,86].

2.5 Analytical Solution

An alternative to a direct memory map of the function to be evaluated would be an analytical solution of the functions involved, this would require a considerable amount of intensive calculation and be many times slower than the look up table approach. If some form of multiprocessor were used then it may be possible to achieve a sufficiently high processing rate while obviating the need for vast amounts of memory to process the functions involved. The programmable nature of a multiprocessor could provide a great deal of flexibility both in the function evaluated and the method of evaluation, where appropriate it would be possible to mix a look up table method of evaluation with other processing.

2.5.1 Serial Nature of Data Taking

The event data occurs and is collected in a time sequential fashion which would seem to contradict the concept of parallel processing of the data events. This 'serial part' (Amdahl[3]) is inherent since there is only one experiment and one set of detectors. Experiments in High Energy Physics deal with events over time scales in the region of pico and nano seconds so the experiment itself is highly unlikely to be a limiting factor in the data rates achievable, producing data several orders of magnitude faster than it can be collected and processed.

2.5.2 Parallel Requirement of Processing

The event manager can, on detection of an event, read out the required data in a matter of a few hardware cycles but the processing of such an event analytically can take several to many hardware cycles. Some overlapping (parallelism) of event processing would be required to bring the processing data rate to that of the collection of data by the event manager. This suggests that some form of multiprocessor could usefully be applied to the processing of data.

2.5.3 Event Nature of Data

The data in each event is entirely independant of the others, the events can be regarded as separate entities since there are no data dependencies from one event to another.

2.5.4 Possible Granularity of Parallelism

From the above it would seem that some form of parallel processing would be required to perform an analytical solution of the data processing, the level at which this parallelism occurs is open to consideration. This, along with other considerations that influenced the approach to this problem is dealt with in chapter 3.

CHAPTER 3

3 THEORETICAL CONSIDERATIONS

3.1 Introduction

From the previous discussion it is clear that high rates of data processing are required in High Energy Physics experiments. Though some experiments are done as discrete 'shots' and require the processing of a large amount of related data to a fixed (usually short) deadline[6] before the next firing in the case under consideration the requirement was for a high average throughput of data over an appreciable period of time. The data from the experiments under consideration consisted of separate 'events', each being independent of all other events and there being no determinate order of arrival of events. The information sought in this data was principally the frequencies of occurrence of particular types of events.

3.2 Processor Criteria

Several criteria were drawn up as guidelines for consideration in the selection of a computing structure;

1) The system should be expandable to a large degree, preferably infinitely, without any of the mechanisms involved becoming saturated.

2) The system should have at least sufficient fault tolerance to allow the processed data to be recovered and possibly to process data at a reduced rate following a

failure.

3) A throughput of events in the order of 100 000 events/second was taken as an initial target figure.

4) The possibility of replacement of the Event Manager in its present form was to be considered, possibly with some form of multiprocessor, perhaps using a multi-write bus to compare detected values[94]. It is possible that such a multi-processor could achieve a large bandwidth through the use of multiple output streams and a structure capable of accepting multiple input streams would be an advantage in such a case.

Consideration of the above points does imply that a multiprocessor system of some form could be usefully applied, the alternative, a large monoprocessor is currently in use and is an expensive and not entirely satisfactory method of processing the data.

Having established that a multiprocessor of some form is to be used then a further criterion may be added to the list :

5) All processors should be identical in form.

Dealing with these points in more detail; the processor was intended for a research environment with a consequent wide variety in the exact nature of the processing to be performed and the possibility that as experiments become more complex a higher degree of computing power would be required. The ability to expand the processor without changing the form of its use would allow the processor to be tailored to the

processing task.

The requirement for fault-tolerance points towards a system with built in redundancy, possibly with multiple busses or a suitable interconnection network.

The figure of 100 000 events/second was taken as a target figure as this is approaching the maximum rate that the current event manager could achieve. This figure sets a lower bound on the number of processors required in terms of the processing time for each event, the number of processors is given by

$$\text{Number of Processors} = \lceil 100\ 000 \times T_p \rceil$$

where T_p is the time required to process an event and $\lceil x \rceil$ is the smallest integer not less than x. This expression assumes the presence of sufficient parallelism in the computation to keep this number of processors busy.

Finally, if all processors are made identical then the design is simplified to some extent since once designed and tested the processors can be readily duplicated. Identical processors also allow for a reduction of the number of replacement boards required as stock.

These criteria are comparable to the design objectives of the TUMULT ring network[129,130], however, the ring structure proposed for use with TUMULT takes the form of a shared communication ring, in the direct connection machine[87] proposed here the term ring reflects the nature of the direct connection graph formed by the interconnections of the processors rather than the form of the communication structure

to which they are connected.

3.3 Assumptions on the Form of the Data

The principle assumption made about the form of the data was that the data occurred as discrete independent events, an assumption also made in the scheme of Glendinning and Hey[57]. It was assumed that these events could be processed in any order so long as the overall distribution of events was maintained and that a sufficiently high rate of processing was achieved. This assumption allows the grain of parallelism to be set at replication of the entire event processing program, the experiment and the event manager providing data sufficiently rapidly to allow a large degree of overlap of the processing of separate events and consequently maintain the required parallelism indicated above. This scheme avoids many decomposition and synchronisation problems associated with smaller grain size[120,55,21,81].

3.4 Possible Structures

3.4.1 Bus Connected Structures

With these considerations in mind some form of multiprocessor structure and processing scheme were required, bus structures were not appropriate since a bus structure will inevitably saturate at some point, however high a bandwidth the bus may have. In addition, if the communication algorithm is not carefully selected then buses can prove prone to lockout and starvation of processes. It is also possible for the buses to fail such that data cannot be retrieved from any of the

nodes without the implementation of multiple buses or some form of bus isolator which are themselves prone to failure and expensive to implement.

3.4.2 Direct Network Computers

For these reasons the mechanisms considered were message passing direct network computers where processors are connected together in a network communicating over short links between pairs of processors only. Such machines are often referred to as direct connection[87], direct network[133] or homogeneous ensemble machines[56], they have several advantages over a shared communication medium as in a bus or lan type of structure. Since each processing node includes communication hardware as the number of processors increases the bandwidth for communication increases proportionately, a very important point[97], possible internal schemes for a node in such a system have been suggested by various authors[8,122]. The TRANSPUTER [78,102,156,148,11,155] has been specifically designed for construction of direct connection machines and these have four serial links; to allow the possibility of using TRANSPUTERS to construct the machine it would be necessary to implement connection graphs with connectivities of less than four. Von Conta[152] has also observed that connectivities ≤ 4 are regarded as advantageous for practical reasons. A comparison of the relative merits of different networks has been given by Wittie[158].

3.4.2.1 Usage Pattern

The anticipated usage pattern of a network computer as a processor for High Energy Physics event processing differs from that of many computationally intensive tasks which use the communications links for interprocessor communication of internally generated messages; the links would be used as a means of distributing the externally generated (by the experiment and event manager) events making up the workload. This usage makes external communication a factor of greater importance than that where a computing problem is loaded and the results collected after a period of intensive computing.

3.4.2.2 Tree Structures

A tree structure made up of such processors would be infinitely expansible and for some applications may present a reasonable solution[20,22] though the communication bandwidth can become a limitation particularly through the root of the tree[87]. Some modifications to tree structures have been proposed to overcome its disadvantages[42,103,69]. The nodes may be made identical in such a structure but there is no inherent fault tolerance since a tree is a separable graph (it has a vertex connectivity of one) and the failure of a single node or link disconnects the related sub-trees which can isolate one or more nodes completely from the rest of the system.

3.4.2.3 Pyramid Structures

Since there is only one source of data, ie the event manager, a pyramid structure with data fed to its apex may be appropriate. If the network is built up from pyramids with the base nodes being connected to the apexes of other pyramids, as shown in fig 3.1, then the connectivity of the nodes is very likely to become excessive. In addition there is no possibility of feeding the structure simply at several different points to allow for improvements in the event manager unless the pyramid is fed from the base. In this case, as in that of the tree structure, the bandwidth through the apex nodes may prove to be a limiting factor.

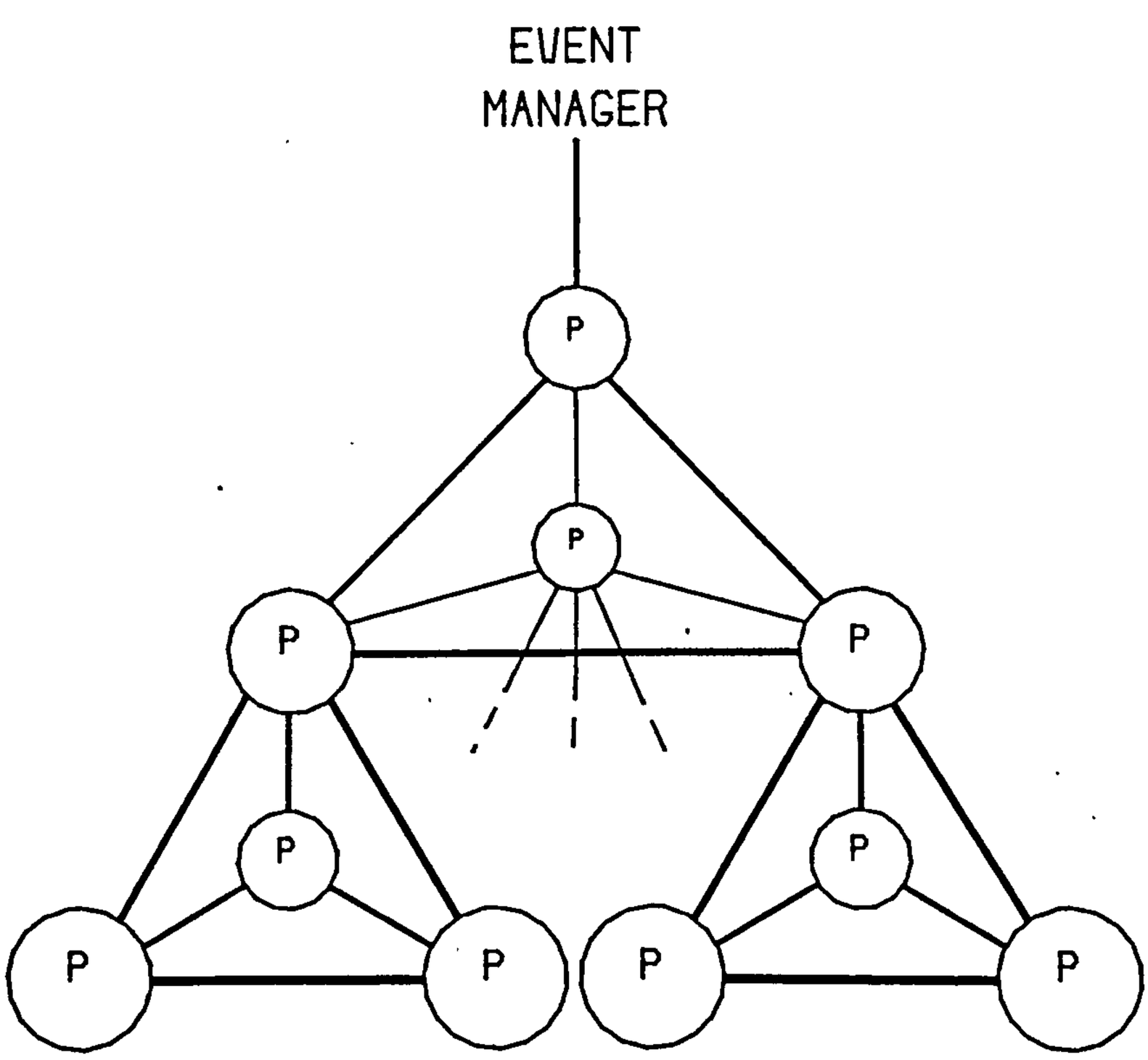


FIG. 3-1 PYRAMID STRUCTURE.

3.4.2.4 Ring Structure

A ring structure may be expanded infinitely though as the number of processors increases the bandwidth required for communication between processors would also increase and may lead to saturation of the communication links. The nodes of such a ring may be made identical in form during normal operation though a single fault may prevent proper functioning of the ring. In the case of one fault the state and processed data could be retrieved by a modified communication algorithm, this would require the processors in the ring 'knowing' that a fault had occurred, it is also possible that two faults could occur in such a way as to isolate part of the ring completely.

Despite its disadvantages the basic ring structure appeared to be worthy of further study.

3.4.2.5 Cylindrical Structures

A cylindrical structure, ie one or more rings of processors connected vertically in addition to the horizontal connections of the ring structure, overcomes many of the problems of the simple ring; this structure does not appear to have previously been investigated in the literature. The shape of the cylinder thus formed, ie its height/width ratio, may be varied to tailor the nature of the structure to the application. The nodes of a cylinder may be identical though an additional two communication links are required compared to those of the simple ring structure, the edge connectivity is four as compared to two for a ring (as far as processor-processor connections are concerned). The vertex

connectivity of a cylinder will vary with the shape of the cylinder but would still be greater than two, the vertex connectivity of a ring.

3.4.2.5.1 Feed Mechanism

Though in this analysis the data items are assumed to require uniform processing it is possible and indeed likely that some types of data may be processed more rapidly than others. If the input data streams are entirely independent this could result in an imbalance of data taken in at these points and consequently disturb the event distribution. If the data is generated from one experiment and one set of front-end electronics then the distribution of events processed will ultimately be the same as that collected by the electronics though some types of events may be slower to be processed and counted. If the data input streams originate from separate sources it is possible to preserve their arrival distribution by simply gating together the status of the input streams of the fed nodes, if any one node cannot accept more data then the others should be prevented from accepting data, this would ensure that those input streams that cleared faster were held back to match the rate at which other data items were dealt with.

With the event manager presently in use, and possibly any future event manager, data occurs as only one output stream. It would seem therefore illogical to consider multiple input arrangements. By a similar argument to that for the requirement for a multiprocessor in section 2.5.2 if an event

manager is available that is capable of collecting event data at extremely high data rates it would be beneficial to be able to divert this along the multiple lower bandwidth paths (restricted by both hardware cost and any routing algorithm software required) to allow full use to be made of the event manager's bandwidth. This splitting of the data stream would require no processing since any routing could be performed by the computing system itself allowing a simple autonomous mechanism to perform the splitting and consequently maintain a high throughput. This would be an ideal point to perform the required gating of the input streams of data but could be a possible single point failure, the likelihood of such failure is reduced by the simple nature of such a device, the event manager itself being more likely to fail.

3.4.2.6 Distinct Node Flow Models

Most study of communication structures has concentrated on the performance of structures for mutual communication between processors engaged in a computation; properties of the interconnection graph such as diameter, total bandwidth and various path related properties have been proposed as indicators of performance though their value has been questioned[93]. Since the pattern of usage of the structure proposed here was to be one of distribution of events rather than mutual communication a rather different criterion, the mean throughput of events has been used as the indicator of performance.

The usual methods for modelling of multiprocessor systems are Markov models, some form of Petri Net technique[92] or Queueing networks[99]. Modelling of ring structures has been carried out using queuing networks, the approximate model developed by Zablotski *et al*[160] serving the same purpose as the model developed here, permitting structural variations of multiprocessors to be evaluated quickly, Protopapas and Denenberg[119] have developed a modelling technique for delays in multicomputer networks. Housheng[70] has demonstrated the superiority of a buffer insertion ring (of which the ring described below may be regarded as an example) over slotted and token rings using a simple queuing model.

In this case however simple static flow models were developed, the flow model having a direct correspondence with the important criterion, mean throughput of events. A rather abstract approach was taken, the model being at the processor level[64] as it was not intended to model the system behaviour with a pre-conceived design and operation assumptions but rather to find some indication of the characteristics required of processors to work efficiently in such a structure.

Simple models of both rings and cylinders were developed, based on flow characteristics. This does not model the detailed internal workings of a computing system but models the overall behaviour at an abstract level. This model is particularly relevant to asynchronous processors which, if used with suitable buffering should correspond well with such a scheme. In common with other mathematical modelling techniques the system being modelled is very complex and the mathematical

model must be a simplified approximation to the real system.

3.4.2.6.1 Single Ring

The first case considered was that of a ring of processors with data being fed into only one of the processing nodes. Assuming that each processor can process input data at a rate of β_n/R (where β_n is the bandwidth of new data into the fed processors and R is the number of processors in the ring) then the problem becomes trivial. Data rates around the ring decrease in steps of β_n/R until the originating node is reached as shown in fig 3.2. This particular model applies to the case where the data must be routed to a particular node for processing (an even distribution is assumed) and also the case where any data can be processed in any node.

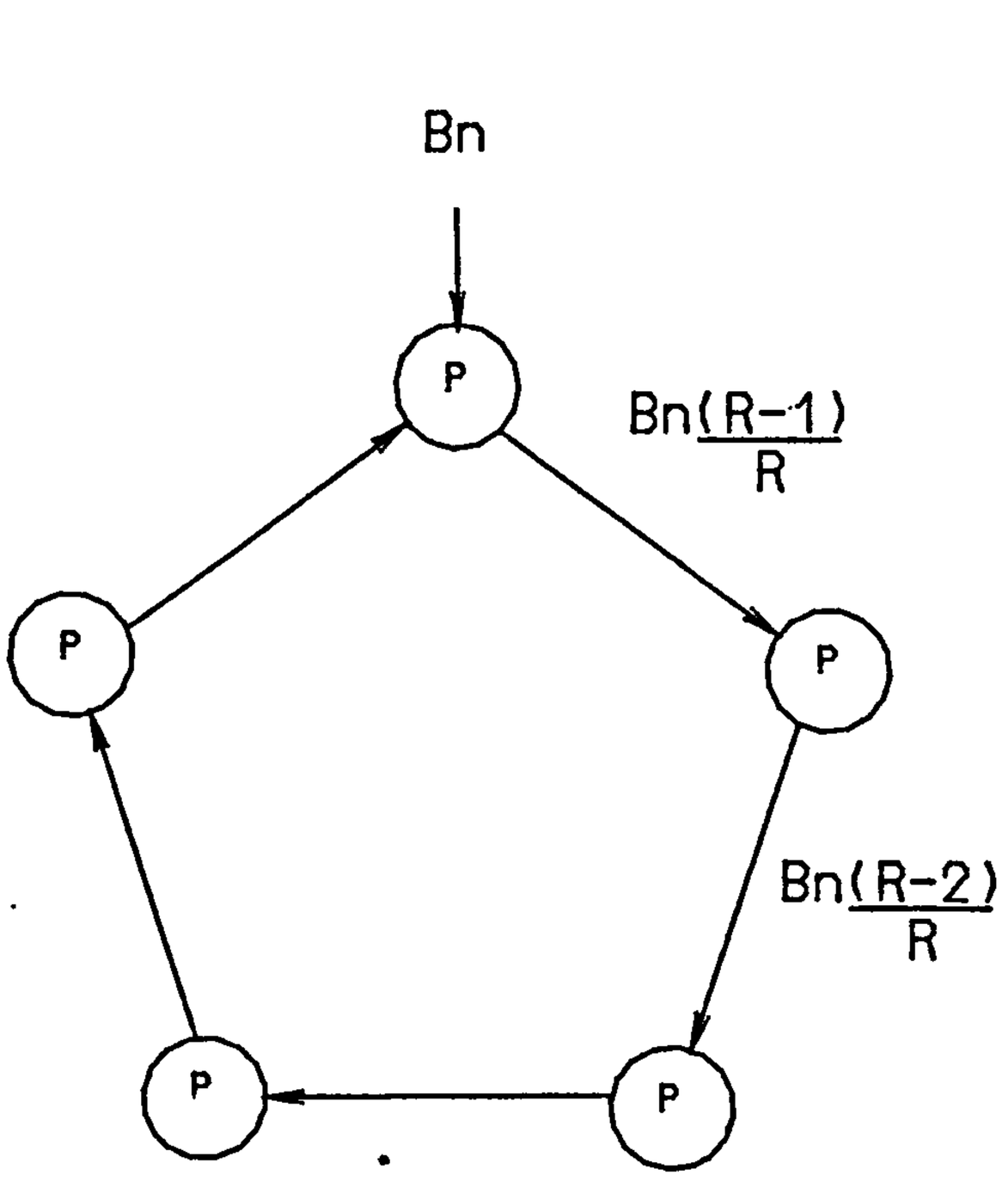


FIG. 3-2 DATA RATES AROUND A RING FED AT A SINGLE NODE.

This assumes of course that the processing of all data items is identical and that the amount of additional bandwidth required for retrieval of the accumulated spectra is small. In a well buffered system the deviation of processing time of events from the mean values should be largely smoothed out to a constant mean value which is assumed to be reasonably similar for each processor. The accumulated data could be read out of the nodes at relatively infrequent intervals or, if bi-directional links are used, be sent in the reverse direction to the incoming data allowing the bandwidth required for the retrieval of the accumulated spectra to be small.

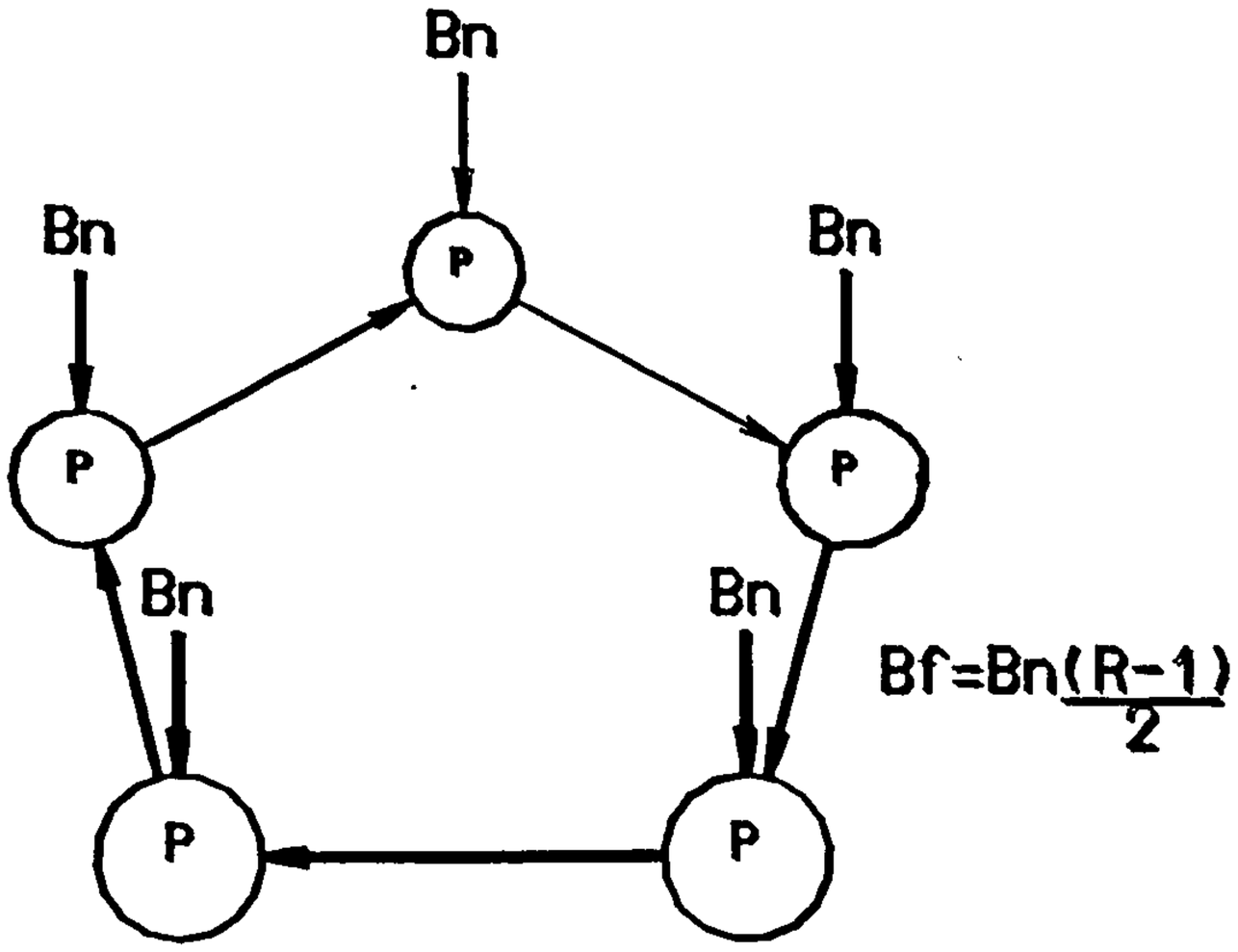


FIG. 3-3 DATA RATES AROUND A RING FED AT ALL NODES.

A ring structure with a single entry point has a total data input bandwidth equal to that of the feed link, which does not meet the requirement of expansibility. The next structure considered was a ring structure, as above, but with data being fed into all of the nodes in the ring simultaneously, this is

shown in fig 3.3.

Again, β_n is the bandwidth into one node from the data source and β_f is the bandwidth forwarded along a link in the ring and R is the number of processors in the ring.

The assumption that the retrieval of processed data was not significant could be made in this case by the same argument as above. The situation modelled was for distinct processing nodes where data items had to be processed in particular nodes depending upon some parameter of the data. It was assumed that the distribution of processing of such data was uniform across the nodes.

In the alternative case where any data may be processed at any node each processor would process the data fed into it, with possibly a small amount of data passed to other nodes to even out variations in data rates and processing rates around the ring.

In the case of processor distinct processing

β_n data arrives at each node from outside the ring.

β_n/R data is processed at the node that it is fed into.

β_f data must be forwarded onto other nodes.

Considering one of the links in the ring, since 1/R th of the data passed on from any one node is removed at each successive node the data from the *r*th node through the link under consideration will be

$$\beta_n \cdot \frac{(R-r)}{R}$$

and the total data through this link will be

$$\beta_f = \sum_{r=1}^{R} \beta_n \cdot \frac{(R-r)}{R}$$

which can be simplified to

$$\beta_f = \beta_n \cdot \frac{(R-1)}{2} \qquad \text{(i)}$$

This relationship is fixed by the distribution of the processing of data throughout the processing system and still applies in the case of saturation of one of the communication links, the bandwidth of data being passed through the other links being restricted in proportion. This result differs from the value of N/2 (R/2) obtained for the mean message density of a one-way ring by Wittie[158] because of the assumption that nodes do not generate messages for themselves, in the model presented here however 1/R th of the data is destined for the node at which it originally arrives which is effectively a self generated message.

All nodes in such a processing ring are identical so the data flows within only one node need be considered. A simple diagram of the data flows within the node is shown in fig 3.4.

In this simple model β_n is the bandwidth of new data taken in at the node, β_f is the bandwidth of data forwarded to the next node in the ring and β_c is the bandwidth of the processing of the data items.

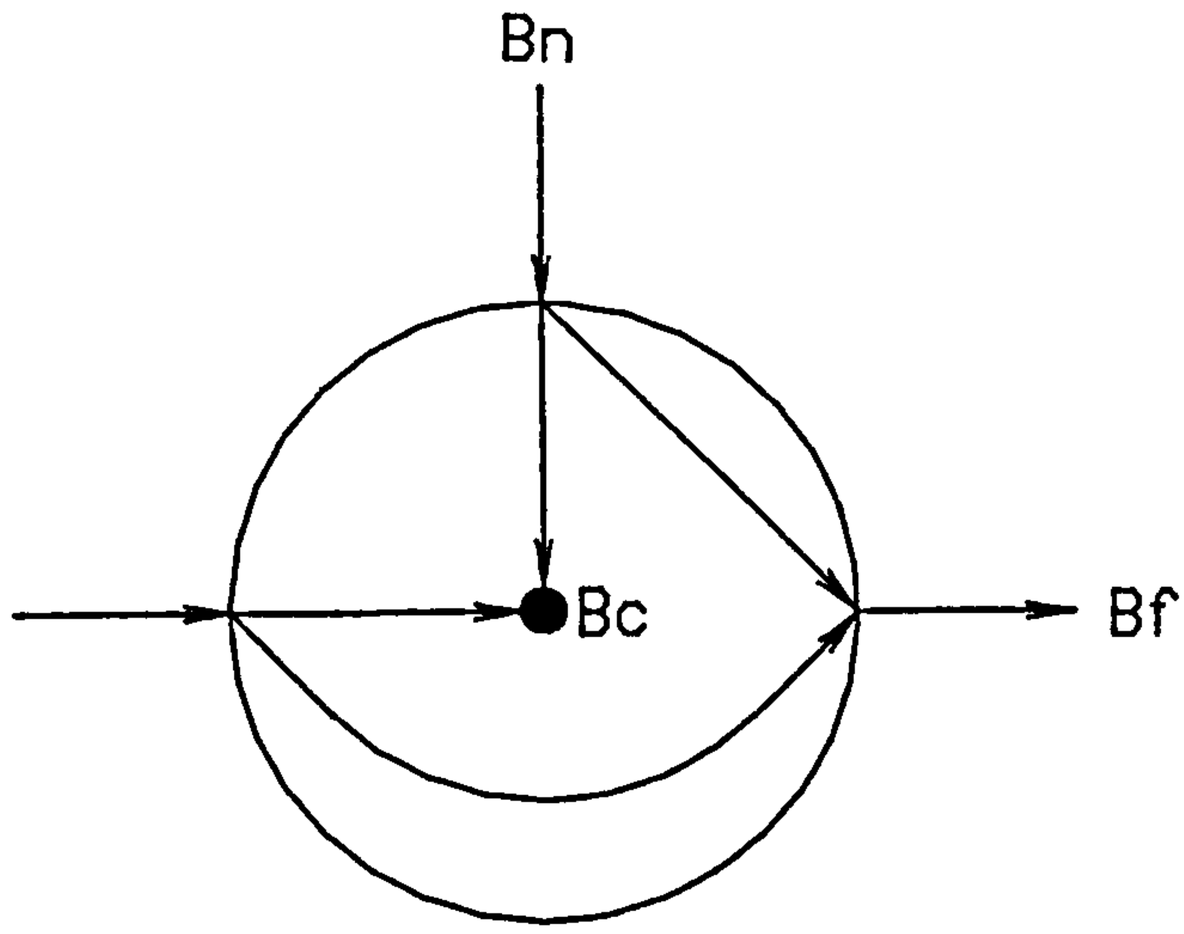


FIG. 3-4 DATA FLOWS WITHIN A PROCESSING NODE OF A RING.

If no communication links saturate then the input data bandwidth is the same as the bandwidth of computation.

$$\beta_n = \beta_c = \beta_{cmax} - K_r \cdot (\beta_n + \beta_f)$$

where β_{cmax} is the processing bandwidth if no data movement has to be performed by the processor and the term $K_r \cdot (\beta_n + \beta_f)$ represents the amount of processing bandwidth used to either pass on data items or to take data from the data streams for processing. The lost computing bandwidth is assumed to be some proportion, K_r, of the data dealt with.

For the case where the communication links are not saturated total processing bandwidth is given by

$$\text{Total Processing} = R \cdot \beta_c$$

$$= R \cdot \beta_{cmax} - R \cdot K_r \cdot (\beta_n + \beta_f)$$

Since $\beta_n = \beta_c$

and $\beta_f = \beta_n \cdot (R-1)/2$

this can be re-written as

$$R \cdot \beta_c = R \cdot \beta_{cmax} - R \cdot K_r \cdot \beta_c \cdot (1 + (R-1)/2)$$

re-arranging this gives

$$\beta_c + \beta_c \cdot K_r (1 + (R-1)/2) = \beta_c max$$

and

$$\text{Total Processing} = \frac{R \cdot \beta_{cmax}}{(1 + K_r \cdot (1 + (R-1)/2)} \quad - \quad \text{(ii)}$$

$$= \beta_{cmax} \cdot \frac{R}{1 + K_r + K_r \cdot R/2 - K_r/2}$$

Graphs of this function are shown in fig 3.5 and fig 3.6 for $\beta_{cmax} = 1$ and values of $0 \leq K_r \leq 1$ and $1 \leq R \leq 20$. When $K_r = 0$, ie no processing is required for routing, a linear speedup with additional processors is achieved.

With this model a processor may be described by the three parameters, β_{cmax}, K_r, and β_{phys}.

If one or more of the processor links saturate then

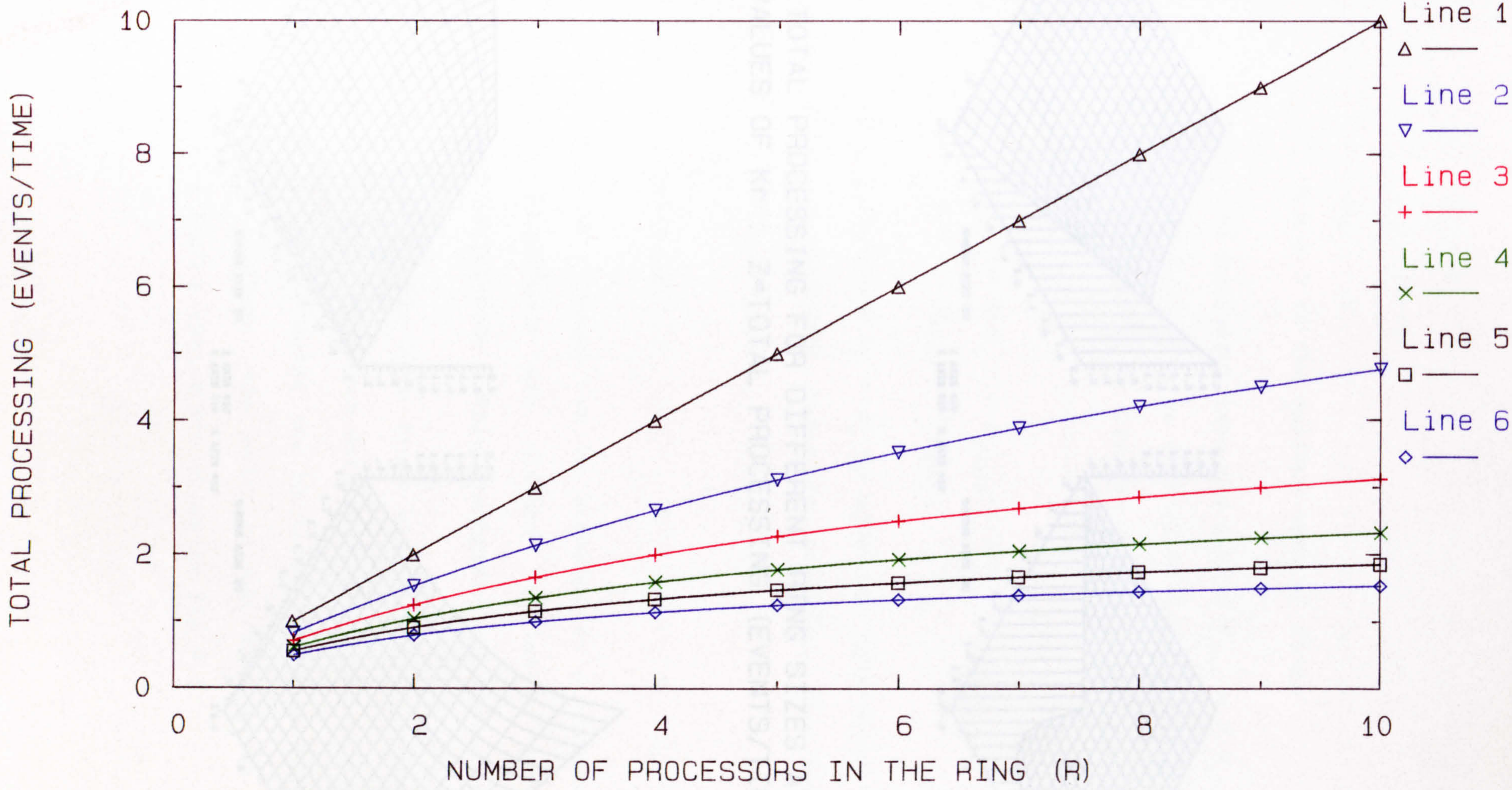
$$\beta_f = \beta_{phys}$$

$$\text{or} \qquad \beta_n = \beta_{phys}$$

β_{phys} being the maximum physical transmission rate of data.

The other term may be determined from (i) above. For the optimum use of the bandwidth of the communication links β_n and β_f should both reach β_{phys} simultaneously.

FIG. 3-5 TOTAL PROCESSING FOR DISTINCT NODE RINGS AT VALUES OF Kr OF 0.0, 0.2, 0.4, 0.6, 0.8, 1.0.



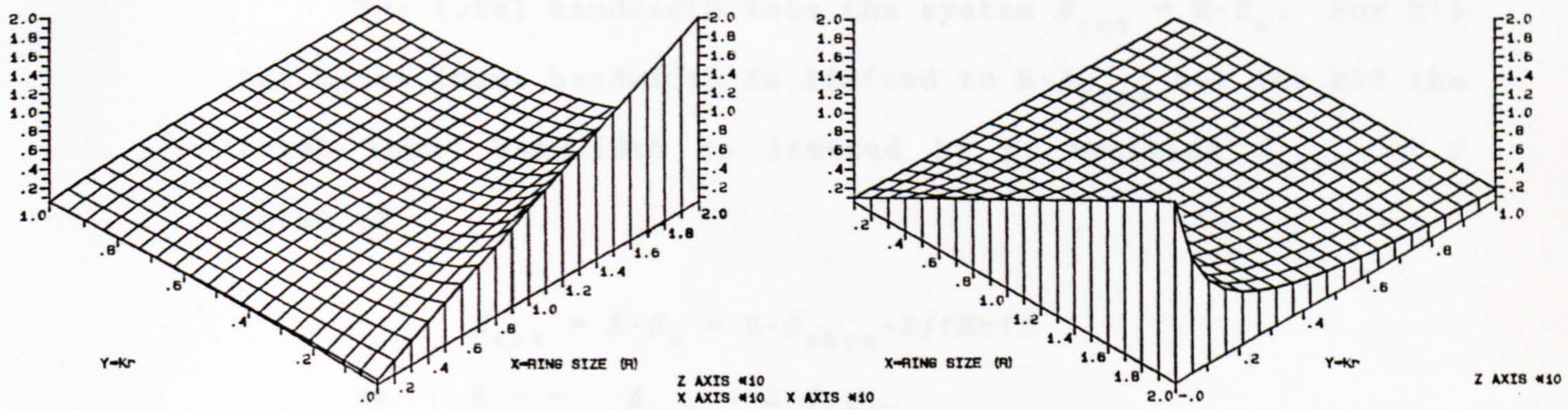
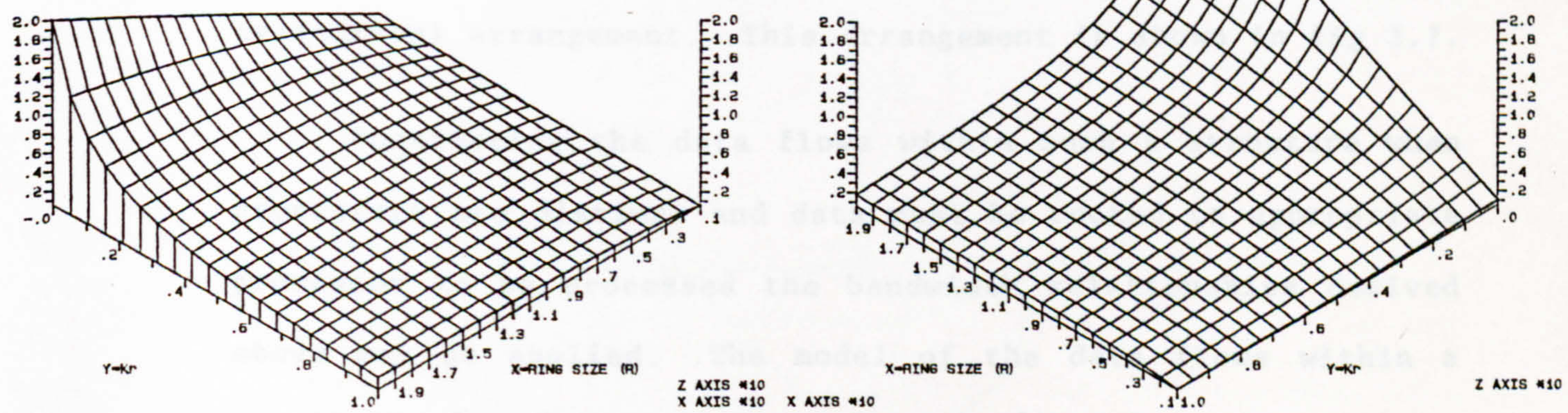


FIG. 3-6 TOTAL PROCESSING FOR DIFFERENT RING SIZES AT VARIOUS VALUES OF Kr : Z=TOTAL PROCESSING (EVENTS/TIME).



$$\beta_f = \beta_n$$

$$\frac{\beta_f}{\beta_n} = \frac{(R-1)}{2}$$

$\Rightarrow$ R = 3 is the optimum size of ring.

The total bandwidth into the system $\beta_{tot} = R \cdot \beta_n$. For R<3 the total input bandwidth is limited to $R \cdot \beta_{phys}$ but for R>3 the total input bandwidth is limited by β_f reaching β_{phys} to a value of

$$\beta_{tot} = R \cdot \beta_n = R \cdot \beta_{phys} \cdot 2/(R-1)$$

as $R \rightarrow \infty$ $\beta_{tot} \rightarrow 2 \cdot \beta_{phys}$

The case where $\beta_n = \beta_f$ is also the case where the total communication bandwidth input is greatest, the total input bandwidth being $3 \cdot \beta_{phys}$.

3.4.2.6.2 Cylinder

The same approach can be extended to a structure where rings of processors are stacked vertically producing a cylindrical arrangement. This arrangement is shown in fig 3.7.

Considering the data flows within such a structure when processors are distinct and data must be routed to appropriate processor to be processed the bandwidth relationships derived above may be applied. The model of the data flows within a processor of such a system are shown in fig 3.8.

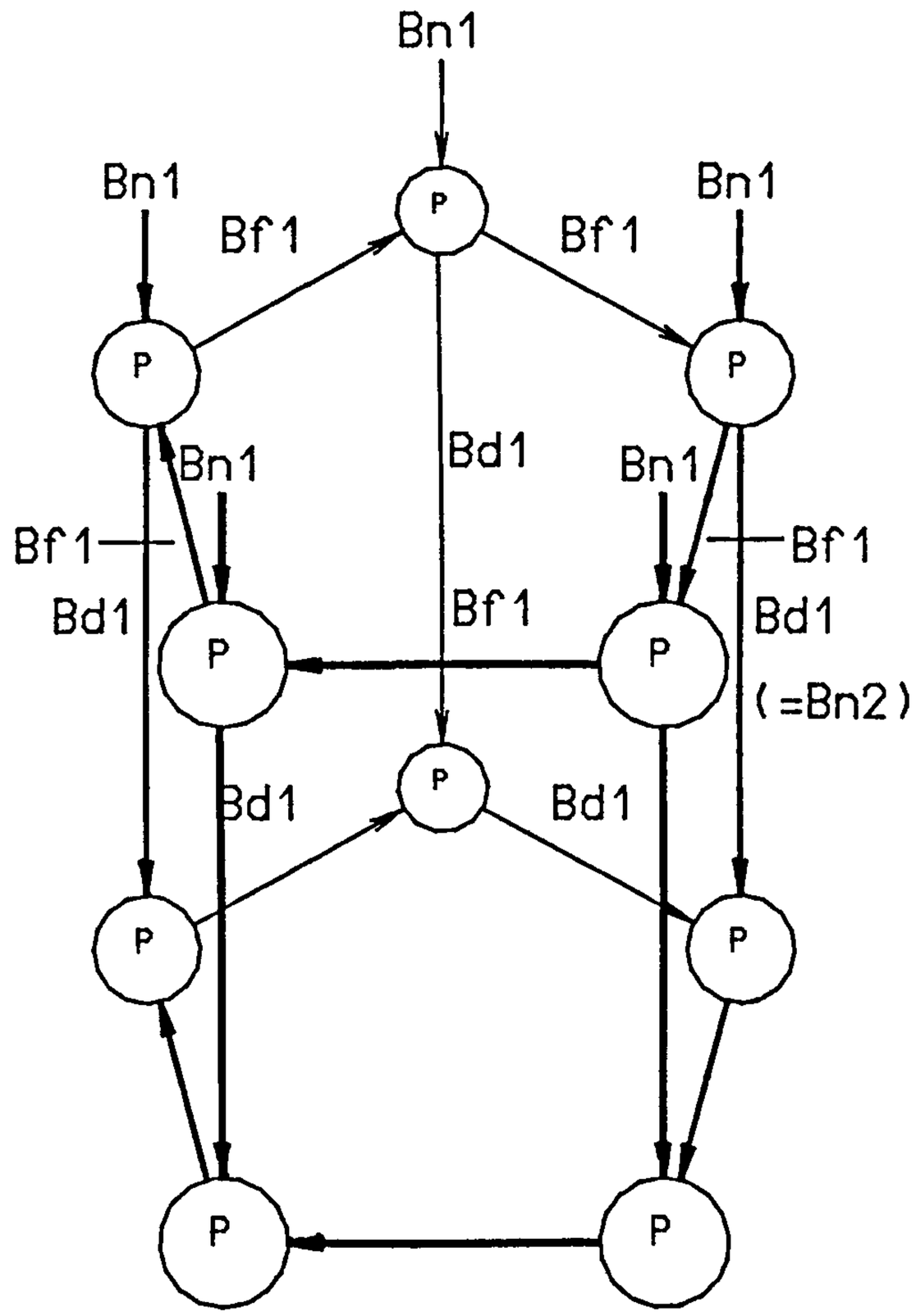


FIG. 3-7 DATA FLOWS THROUGH A CYLINDER FED AT ALL NODES.

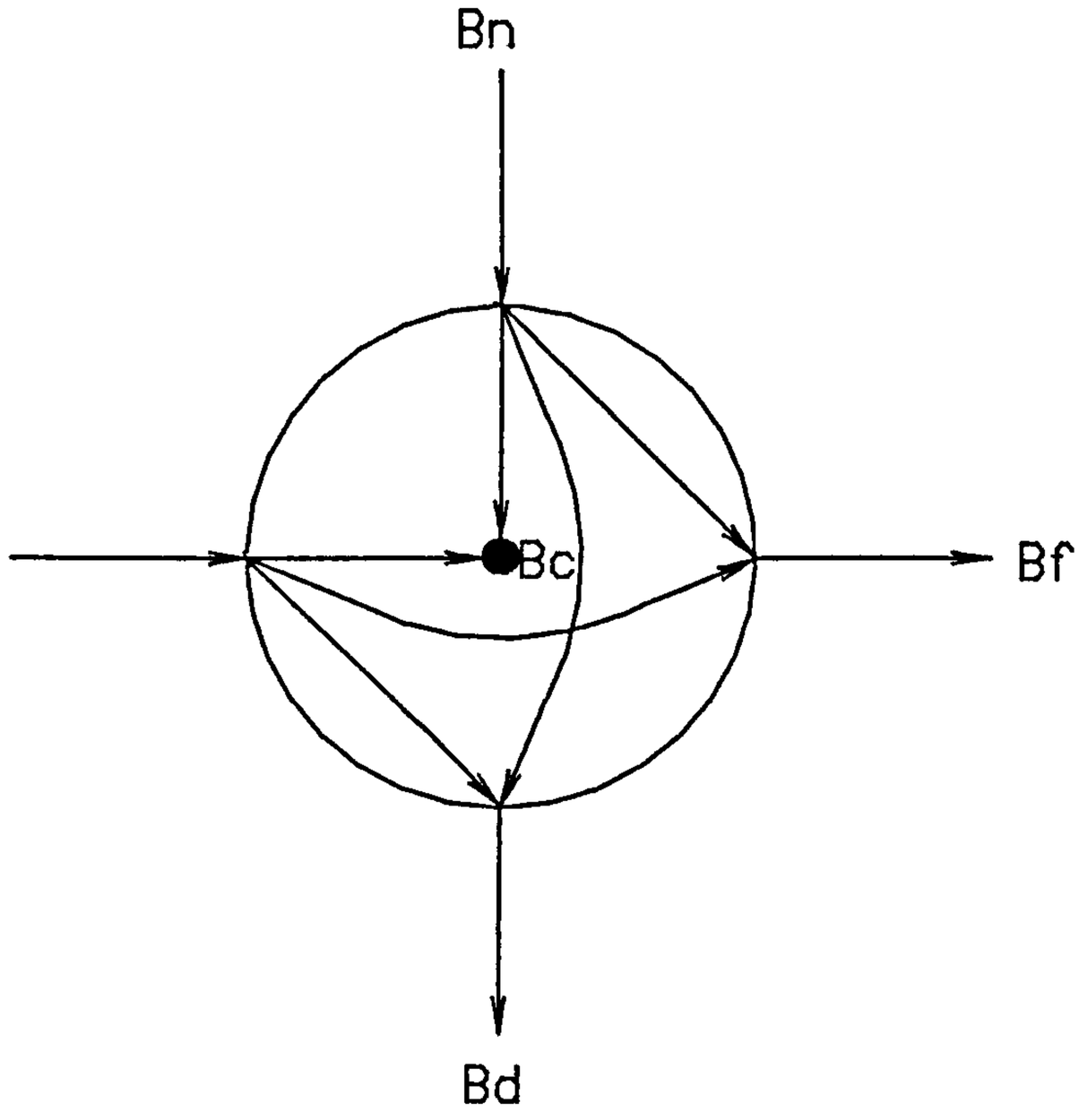


FIG. 3-8 DATA FLOWS WITHIN A PROCESSING NODE OF A CYLINDER.

In addition to the three links of the node model used in the ring a fourth link is added to allow the vertical connection. The bandwidth transmitted through this link is indicated by β_d. The model of this cylindrical arrangement makes the same assumption about the fact that retrieval of spectra from the processors requires little overhead, it also assumes that the processing of all data items takes a similar amount of time. As with the model of the ring the spectra may be read out at infrequent intervals or, if bidirectional links are used, be sent in the reverse direction to incoming data allowing the overhead for retrieval of spectra to be ignored.

The situation where processing nodes were distinct was modelled, the mapping of processing of the data over the structure being assumed to be uniform and data being forwarded to the appropriate node to be processed. Routing of the data was assumed to take place vertically first and then horizontally as is the case in the derivation of the message switching capacity for a square grid by Horowitz and Zarat[69]. The structure was assumed to consist of L layers of R processing nodes, with the R nodes of the uppermost layer (l=1) receiving incoming data.

β_{nl} is the bandwidth fed into the lth layer, β_{n1} being the data bandwidth fed into the top layer of nodes in the cylinder. Each layer of the cylinder can be regarded as a separate ring. Each layer of the structure will receive data given by

$$\beta_{nl} = \beta_{n1} - \beta_{n1} \cdot (l-1)/L$$

and the bandwidth of data to be processed by a layer is given by

$$\beta_{il} - \beta_{i(l+1)} = \beta_{i1}/L$$

It is worth noting at this point that the vertical links of the lower levels should never saturate since $\beta_{i(l+1)}$ is always less than β_{il} and β_{i1} has a maximum value of β_{phys}. Since the data input bandwidth of a layer of the cylinder cannot be greater than β_{i1}/L the optimum value for R (ie the value of R such that the horizontal links saturate as the data input bandwidth reaches its maximum value) is not 3 as in the case of the single ring.

From the bandwidth relationships determined for the ring structure it can be shown that

$$\beta_{fl} = \frac{\beta_{n1}}{L} \cdot \frac{(R-1)}{2} \qquad \text{(iii)}$$

as in the case of the ring model this relationship applies regardless of one of the data flows reaching its maximum since the other data flow will be correspondingly restricted.

If no communication links saturate then the input data bandwidth is the same as the bandwidth of computation

$$\beta_{n1} = L \cdot \beta_{cmean} \text{ and } \beta_{cl} = (\beta_{cmax} - K_r \cdot (\beta_{nl} + \beta_{fl}))$$

where the term $K_r \cdot (\beta_{nl} + \beta_{fl})$ represents the amount of processing bandwidth used to either pass on data items or to take data from the data streams for processing. As in the case

of the ring model the lost computing bandwidth is assumed to be some proportion, K_r, of the data dealt with.

Similarly to the case of the ring

$$\text{Total Processing} = R \cdot \beta_{n1}$$

$$= \sum_{l=1}^{L} R \cdot (\beta_{cmax} - k_r \cdot (\beta_{nl} + \beta_{fl}))$$

since $\beta_{n1}/L = \beta_c$

and $\beta_{fl} = \beta_{n1}/L \cdot (R-1)/2$

$\beta_{nl} = \beta_{n1} - \beta_{n1} \cdot (l-1)/L$

this can be re-written as

$$R \cdot \beta_{n1} = R \cdot \sum_{l=1}^{L} \left\{ \beta_{cmax} - K_r \cdot \left[\beta_{nl} - \beta_{nl} \cdot \frac{(l-1)}{L} + \frac{\beta_{nl}}{L} \cdot \frac{(R-1)}{2} \right] \right\}$$

$$= R \cdot L \cdot \beta_{cmax} - R \cdot L \cdot \beta_{nl} \cdot K_r \cdot \left[1 + \frac{(R-1)}{2L} - \frac{(L-1)}{2L} \right]$$

rearranging gives

$$\frac{\beta_{nl}}{L} + \beta_{nl} \cdot K_r \left[1 + \frac{(R-1)}{2L} - \frac{(L-1)}{2L} \right] = \beta_{cmax}$$

$$\text{Total processing} = \frac{R \cdot \beta_{cmax}}{\frac{1}{L} + K_r \cdot \left[1 + \frac{(R-1)}{2L} - \frac{(L-1)}{2L} \right]}$$

$$= \frac{R \cdot L \cdot \beta_{cmax}}{1 + K_r \cdot \left[\frac{(R-1)}{2} + \frac{(L+1)}{2} \right]}$$

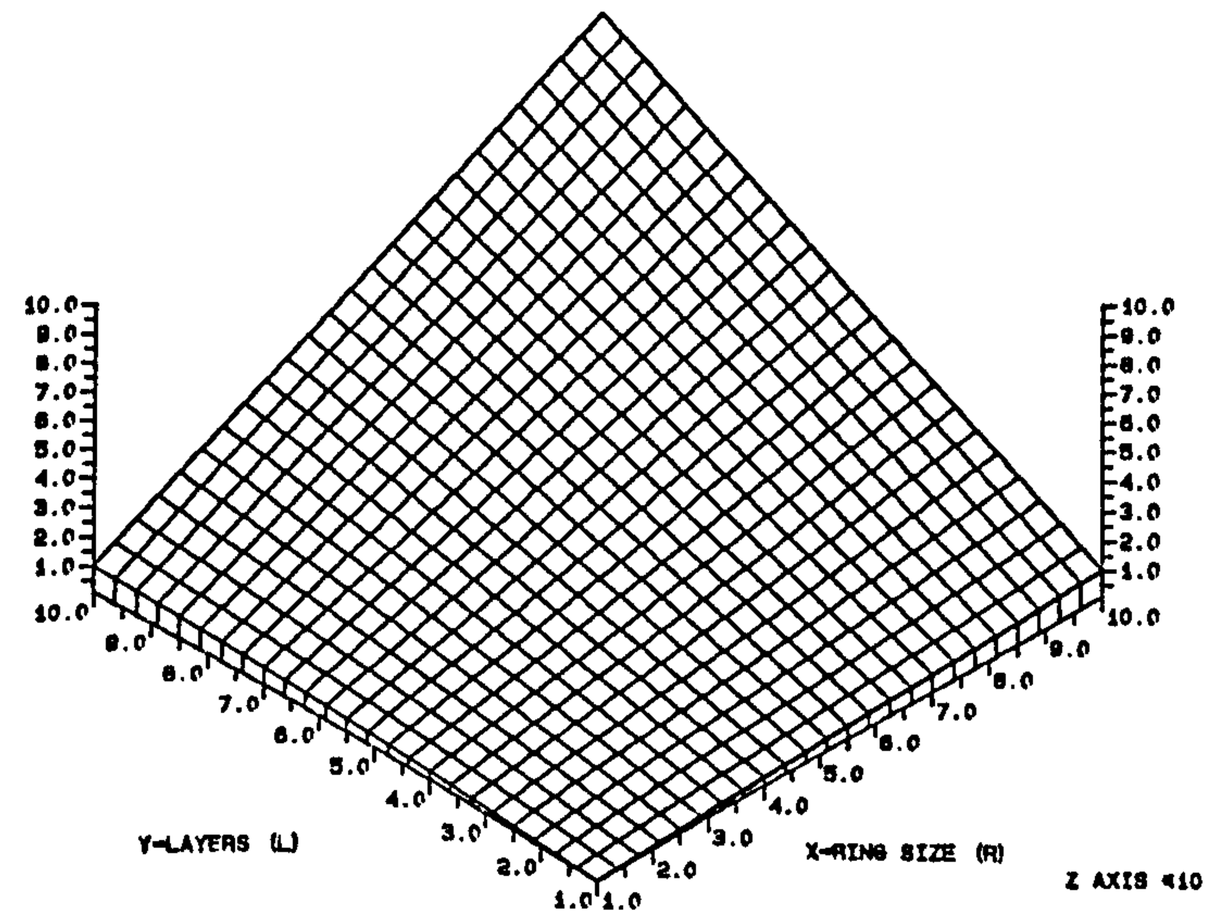
As a check of the consistency of this equation with the ring model it may be observed that for L=1 this equation becomes identical to that for the ring model. Graphs of this function are shown in fig 3.9 - 3.11 for $\beta_{cmax} = 1$ and values of $K_r = 0.0$, 0.5 and 1.0 with values of $1 \leq R \leq 10$ and $1 \leq L \leq 10$. As in the case of the ring when $K_r = 0$, ie no processing is required for routing, a linear speedup with additional processors is achieved. These graphs demonstrate clearly that even a relatively small value of K_r can result in a serious degradation of the processing throughput. In this model a processor may be described by the parameters, β_{cmax}, K_r and β_{phys}.

If β_{f1} or β_{n1} reaches the maximum value of β_{phys} then the other term will be correspondingly restricted, that is if

$$\beta_{f1} = \beta_{phys}$$

$$\text{or} \quad \beta_{n1} = \beta_{phys}$$

then saturation of one of the communication paths has occurred. The other bandwidths involved may be determined from (iii) above. For the optimum use of the bandwidth of the communication links β_{n1} and β_{f1} should both reach β_{phys} simultaneously.



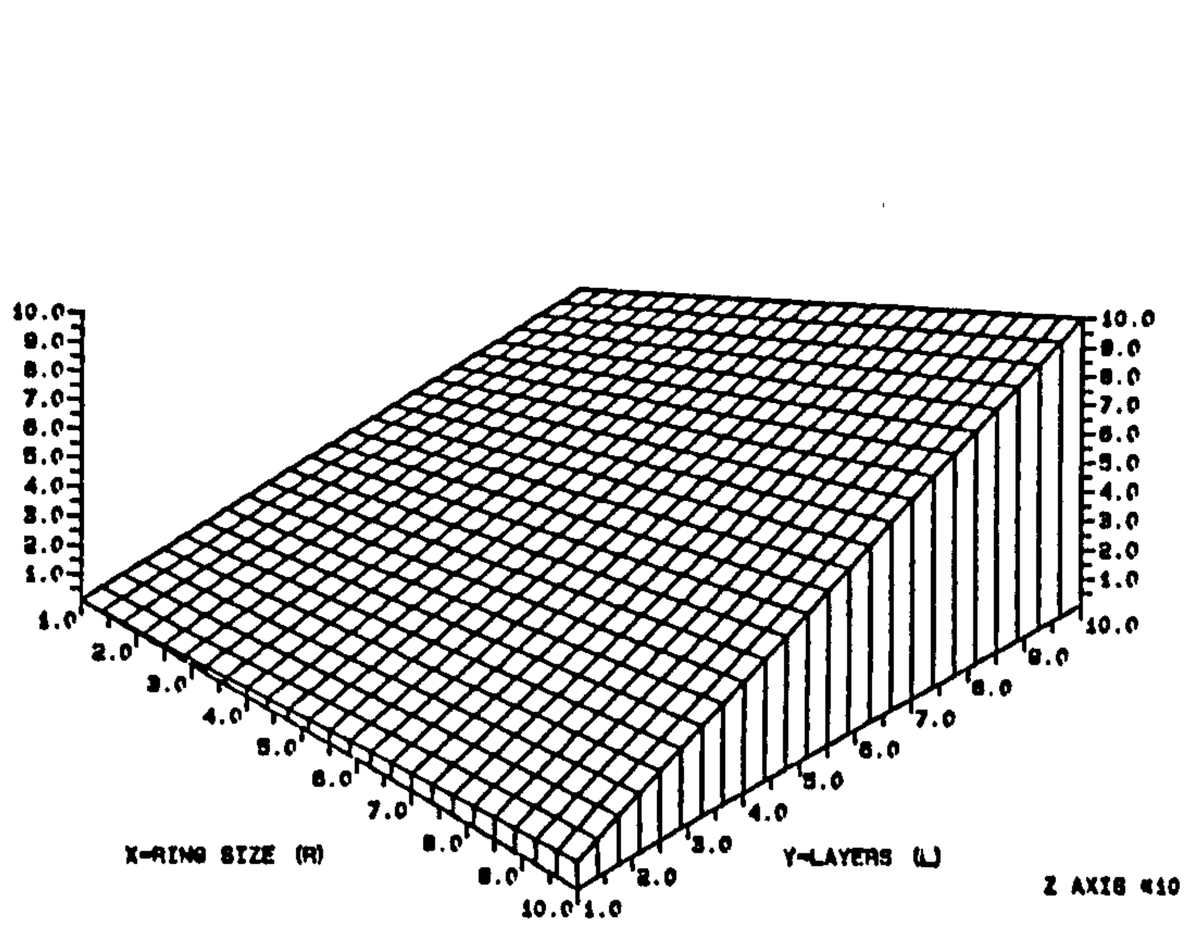
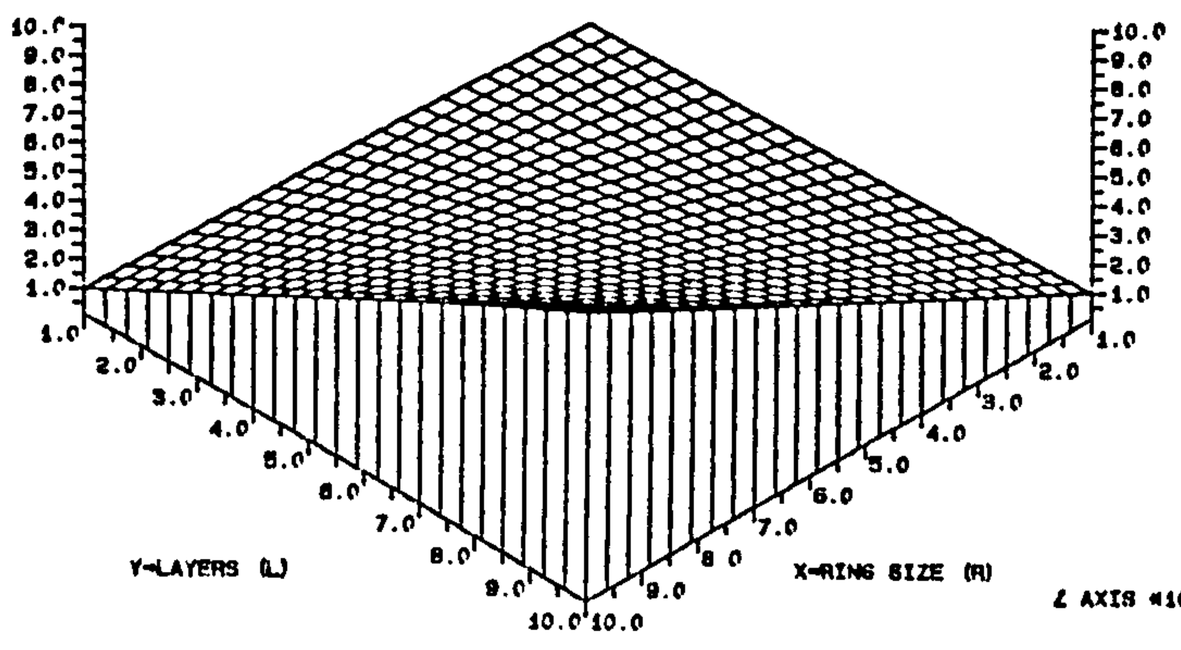
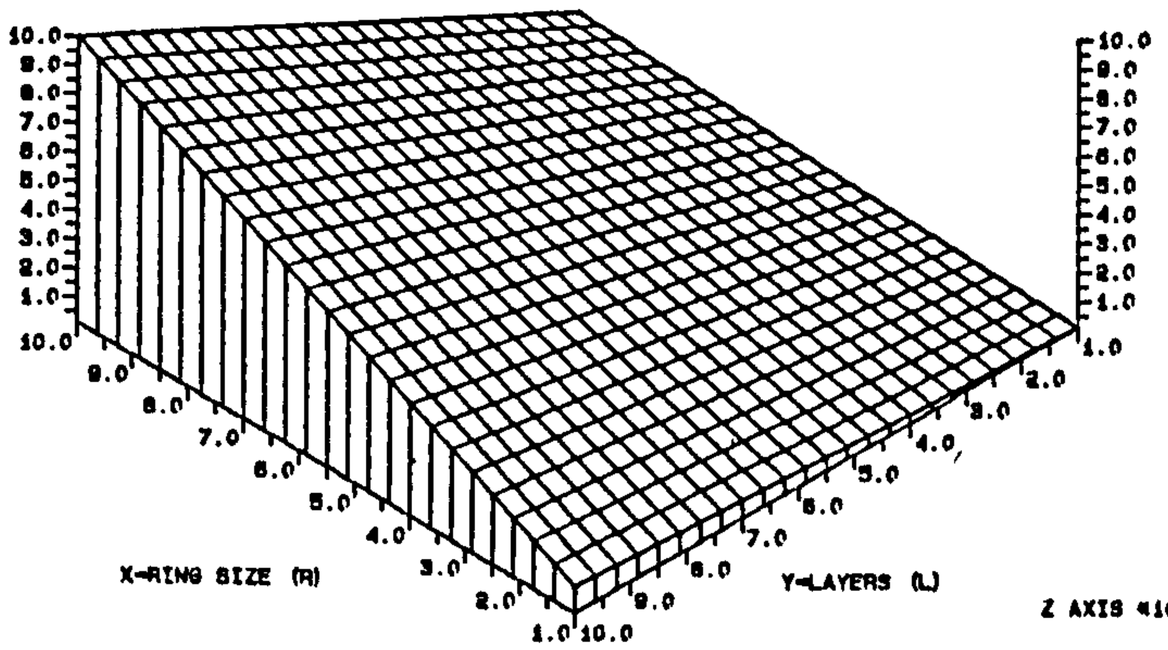
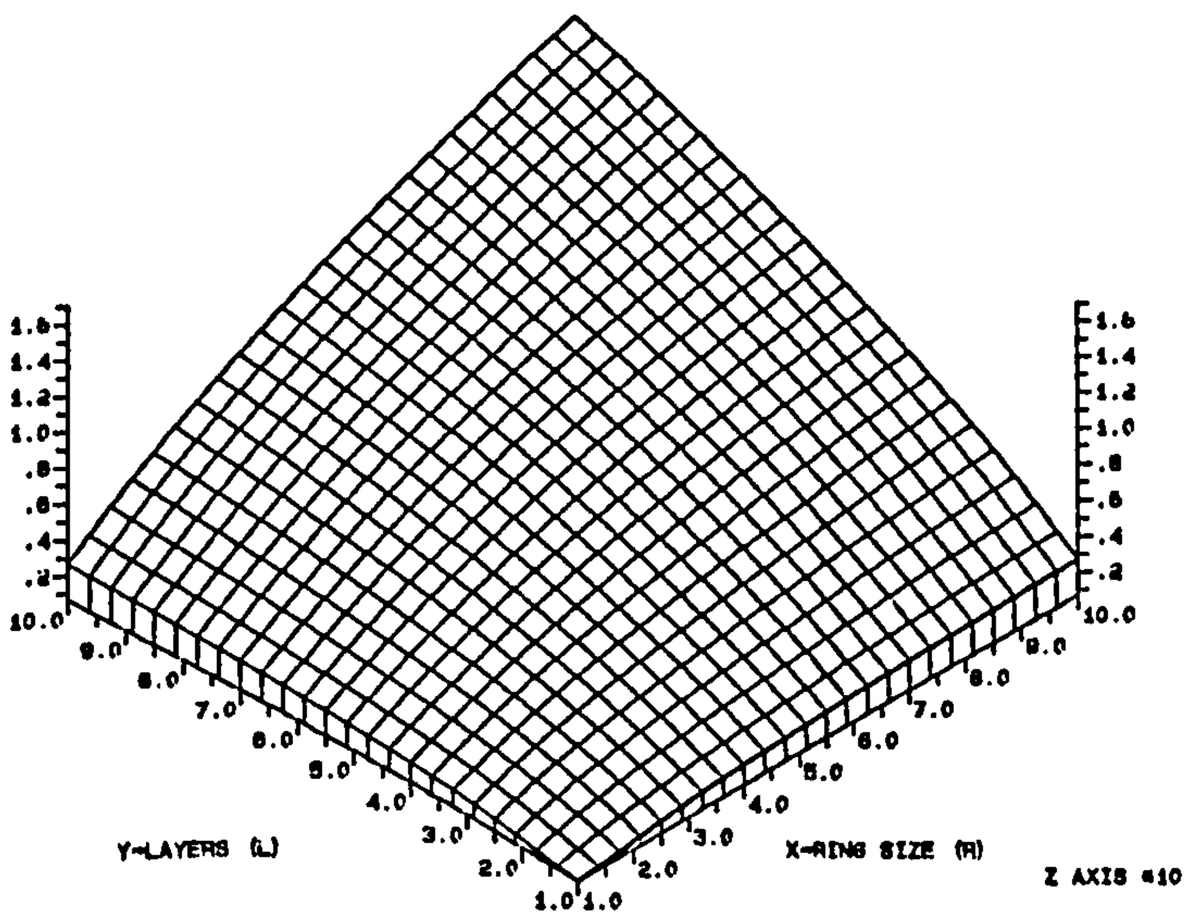


FIG. 3-9 TOTAL PROCESSING FOR DISTINCT NODE CYLINDERS WITH Kr=0 : Z=TOTAL PROCESSING (EVENTS/TIME).







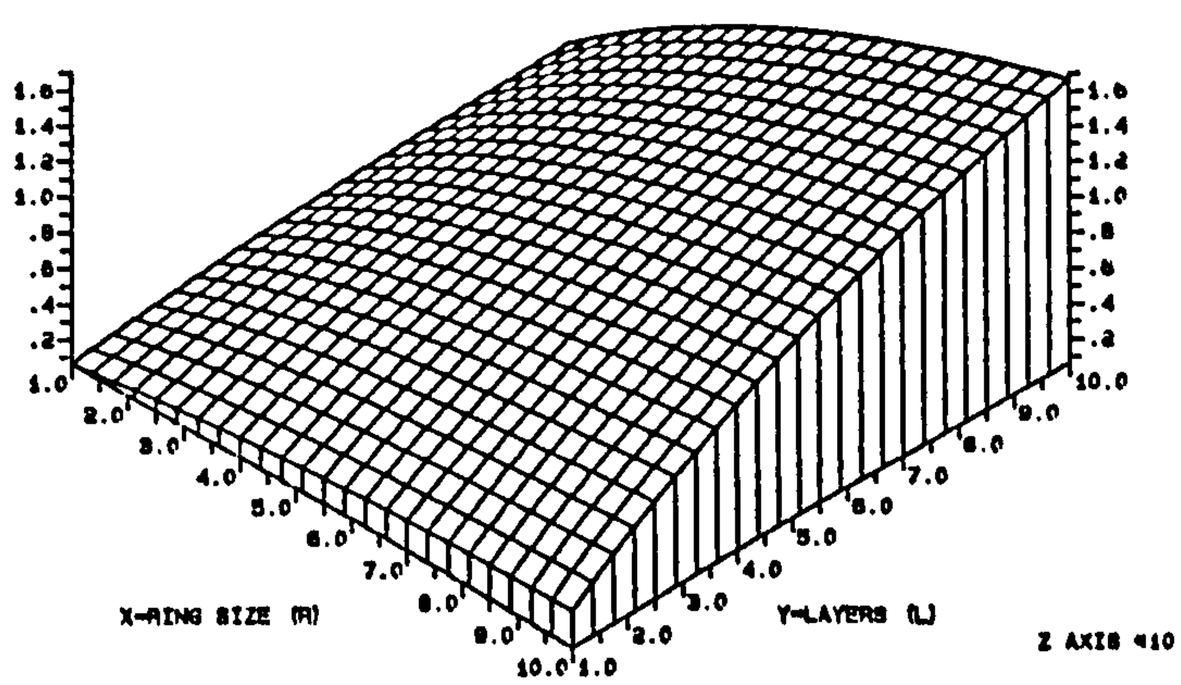
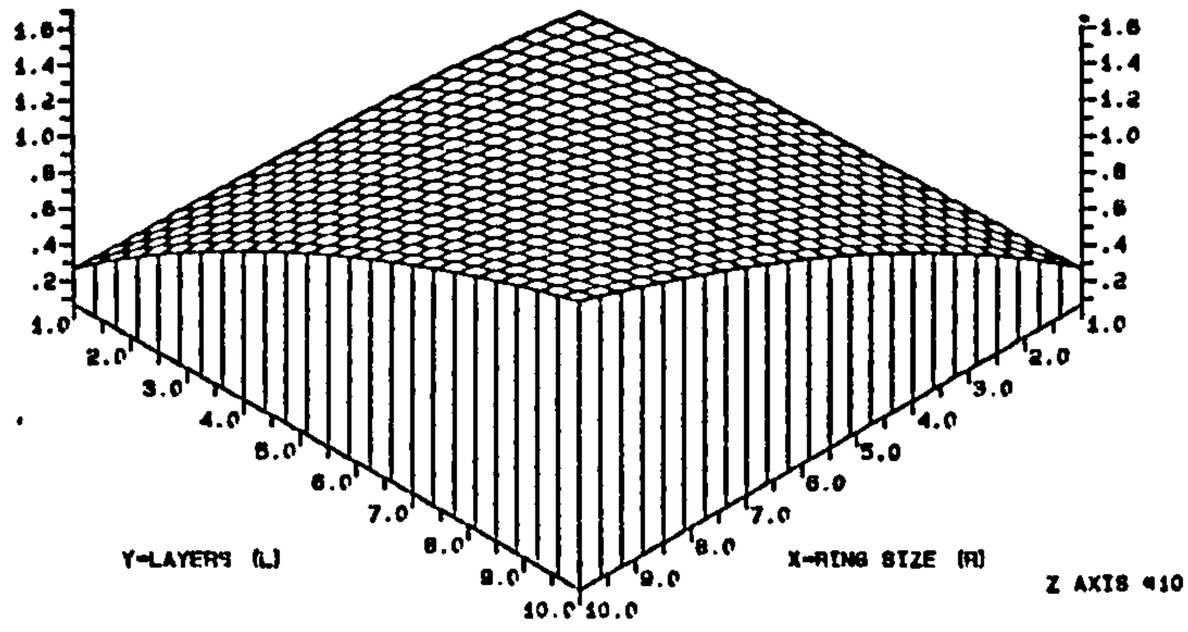
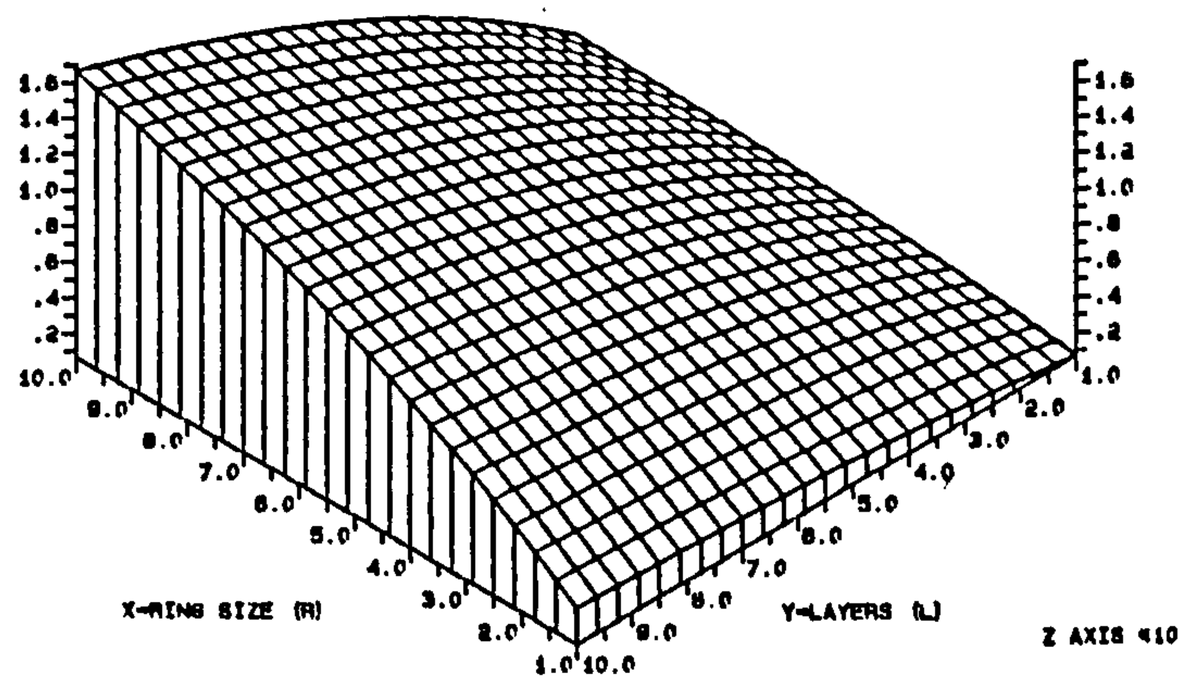
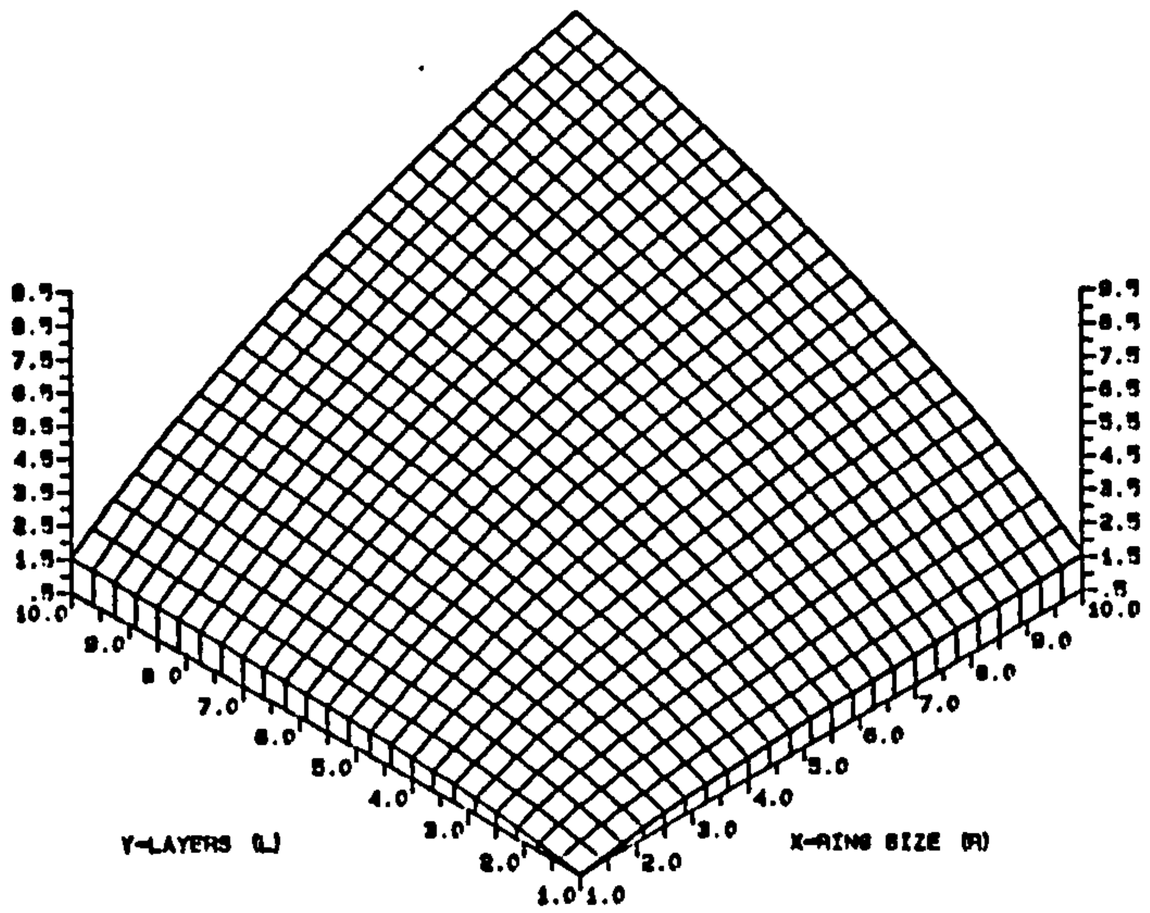


FIG. 3-10 TOTAL PROCESSING FOR DISTINCT NODE CYLINDERS WITH Kr=0.5 : Z=TOTAL PROCESSING (EVENTS/TIME).







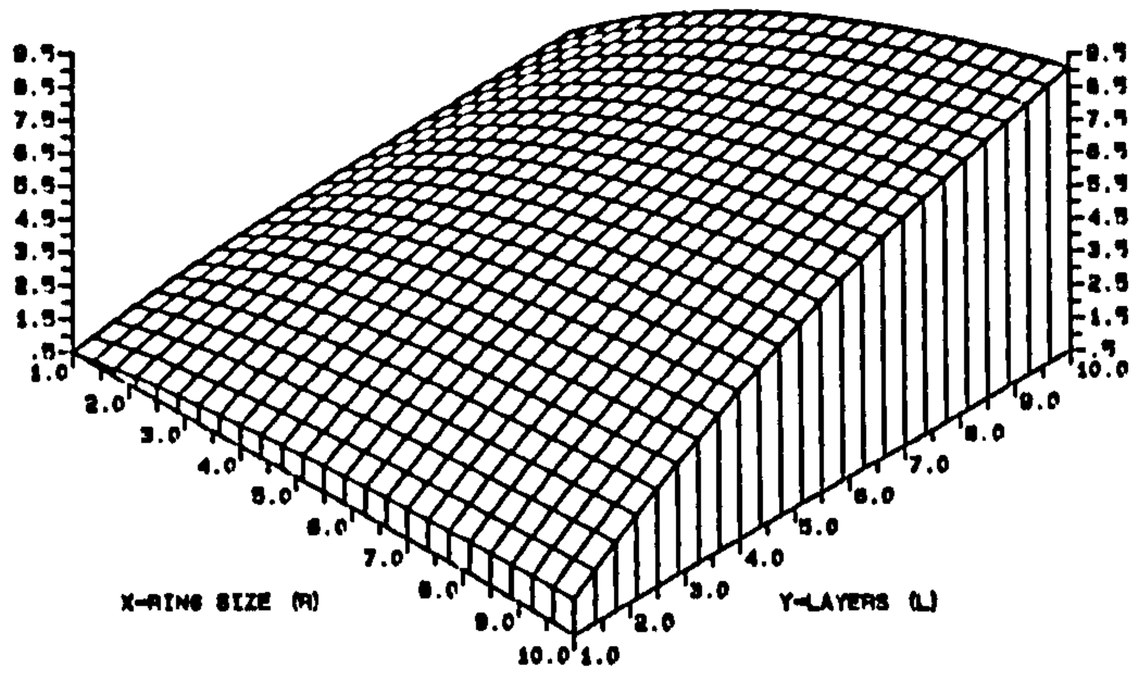
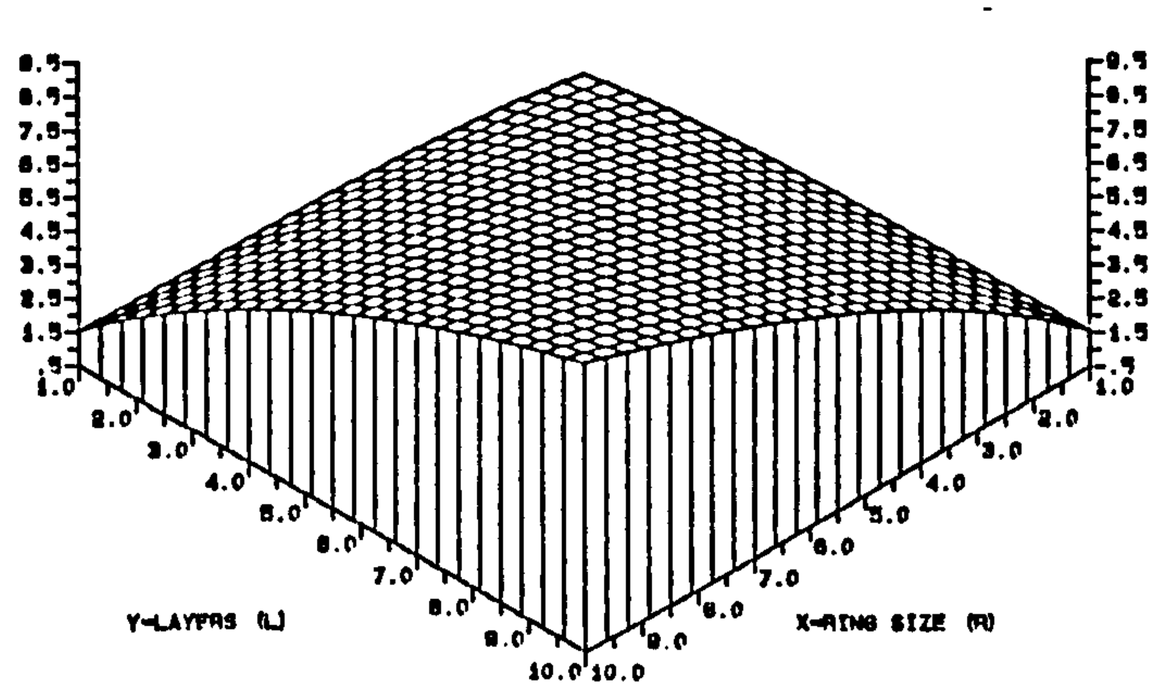
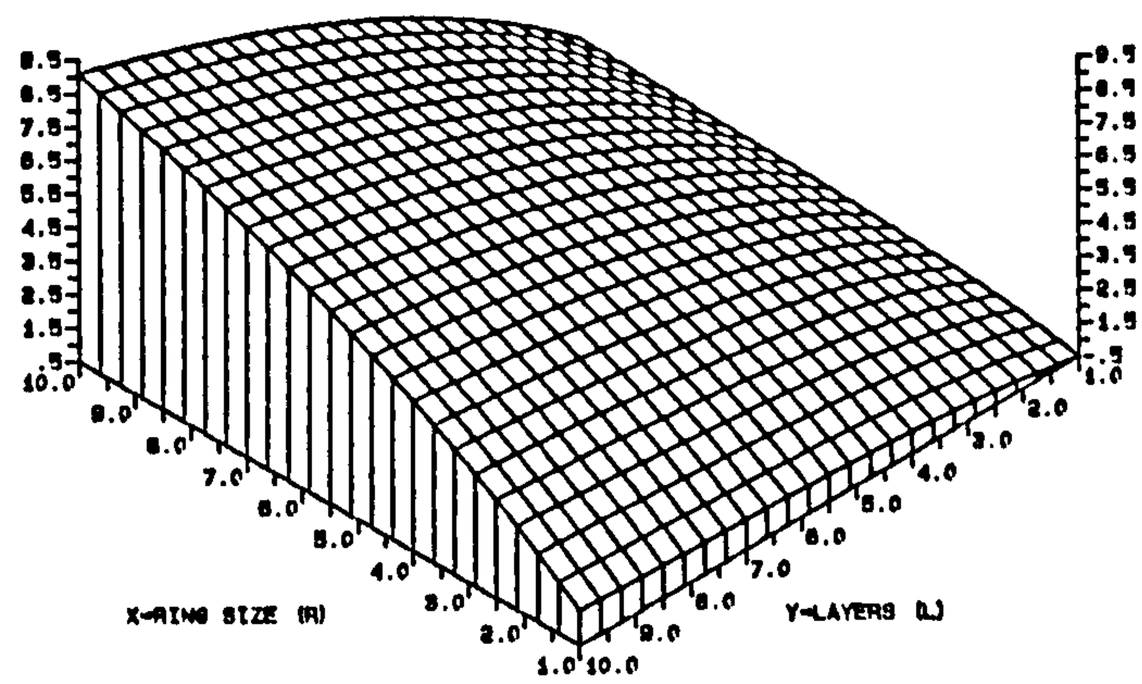


FIG. 3-11 TOTAL PROCESSING FOR DISTINCT NODE CYLINDERS WITH Kr=1 : Z=TOTAL PROCESSING (EVENTS/TIME).





$$\beta_{f1} = \beta_{n1}$$

$$\frac{\beta_{f1}}{\beta_{n1}} = \frac{(R-1)}{2L}$$

$\Rightarrow$ R = 2L + 1 is the optimum shape of the cylinder

The total bandwidth into the system $\beta_{tot} = R \cdot \beta_{n1}$. For R<2L + 1 the total input bandwidth is limited to $R \cdot \beta_{phys}$ but for R>2L + 1 the total input bandwidth is limited by β_{f1} reaching β_{phys} to a value of

$$\beta_{tot} = R \cdot \beta_{n1} = R \cdot \beta_{phys} \cdot \frac{2L}{(R-1)}$$

$$\text{as } r \to \infty \quad \beta_{tot} \to 2L \cdot \beta_{phys}$$

The case where $\beta_{f1} = \beta_{n1}$ is also the case where the total communication bandwidth is greatest, the total input bandwidth being $(2L + 1) \cdot \beta_{phys}$.

From the analysis above several observations can be made. If the processing time of data items is long compared with the time to communicate data, ie $\beta_c << \beta_{phys}$, then the communication links will not saturate and processing bandwidth will be entirely dependant upon the number of processors and not dependant upon the size of the ring structure or the shape of the cylinder, with some possibly small effect due to the changing value of the term $K_r \cdot (\beta_{n1} + \beta_{f1})$. If β_c approaches β_{n1}/L the processing bandwidth will tend to be limited by β_{phys} at some point in the communication structure and the optimum shape allows the total input bandwidth to be maximised. In this case the number of input links may be matched to the number of input sources by suitable selection of the shape of

the cylinder.

The model of distinct processors may be appropriate for some types of processing, perhaps for cases where the storage requirement for spectra is large and it is desirable to store different regions of the spectra in separate nodes or different types of data may require routing to specific nodes for particular types of processing; or possibly in a case where an analytical solution was too slow to be performed, even by a large number of processors and a look up type of approach had to be adopted (see sections 2.4 and 2.5) with a large look up table distributed throughout the processing nodes. If this latter scheme were implemented then the routing would reduce to a simple comparison of the data with the limits of the bounds of the array held at that particular node to test for the data having found the node in which its solution resides.

There are some disadvantages with this computing scheme, one of the most significant being the need to identify which node data is to be processed at, which is likely to involve some pre-processing of the data each time it is routed (with a corresponding high value of K_r). Considering this, it would seem more efficient to process each event to completion in whichever node it found itself and then possibly route the processed data to a particular node for storage. The need to route data to particular nodes also carries with it the penalty that a large amount of data shuffling is performed occupying a correspondingly large amount of communication bandwidth.

The logical conclusion of this discussion is to consider instead a system in which an event is processed to completion in the first node that is not already busy and no un-necessary data movement takes place. This system results in simplification of several aspects of the computing structure, notably the routing algorithm and the implementation of fault tolerance.

3.4.2.7 Homogeneous Flow Model

3.4.2.7.1 Cylinder

An alternative to the above processor distinct scheme is a truly homogeneous machine where any of the data items may be processed at any one of the processing nodes. In this case not only are the hardware units identical but the software may also be made identical. This allows the programming of the processing system to be simplified considerably, as is discussed more fully in section 6.2, only one program being required rather than several programs. It is likely that this single program would be longer than the individual programs of the previous case but not much more so since the processing for routing may be considerably simplified and may even be fixed and retained in firmware rather than down-line loaded since this part of the program is not dependant upon the type of data. This homogeneous scheme does not require events to be routed to specific destinations, unlike most schemes[14] a very simple "if there's room send it" type of approach may be applied since data may be processed in any order. Similarly the problem of mapping a system of interacting processes onto

the machine[143] is avoided.

In the case where the system is homogeneous and all of the top ring are fed with data the system will tend towards a state where only vertical communication takes place as data is passed downwards. However, the horizontal links are useful in that if processing times of events are not equal or if not all of the nodes of the top ring are fed with data they can be used to share the data amongst the processors, they also provide path redundancy making some degree of fault tolerance possible.

Considering a homogeneous cylinder, since $\beta_{n1} \leq \beta_{phys}$ the lower layers should never saturate since for every source of data to a ring there is a corresponding processor and at least one communication link along which data may be sent. However, the further that data is sent before being processed the more processing is lost to forwarding data rather than completion of the processing of events. This should not be taken to imply that processing of events should be performed to the exclusion of forwarding data as this leads to a situation where only those processors directly fed with data perform any useful work, as has been shown by the simulation described in chapter 4 section 4.10.

Since there is no prescribed node to which data should be sent in the case of the homogeneous processor the flow of the data throughout the structure and the consequent behaviour of the structure cannot be modelled in the same way as the case of distinct processors. However, some general statements can be made without reference to the particular algorithm adopted.

Ignoring the computational bandwidth lost due to data requiring forwarding

$$\text{if} \quad \beta_c > \beta_{phys}/L$$

$$\text{then} \quad \beta_{tot} = R \cdot \beta_{phys}$$

ie the communication mechanisms would be the main limitation on throughput, as noted earlier, saturation of the lower levels will not take place and this limitation will be due to the bandwidth of data into the uppermost nodes of the system.

$$\text{If} \quad \beta_c < \beta_{phys}/L$$

$$\text{then} \quad \beta_{tot} = R \cdot L \cdot \beta_c$$

ie the processing mechanisms would be the main limitation on throughput.

Data is only forwarded a limited distance within the system, the distance being dependant upon the amount of processing required for each event. As processors are situated further from the fed nodes they receive a proportionately lower bandwidth of data to deal with. This carries the important implication that there is a limit to the size of the cylinder beyond which processors will never be supplied with data to process.

The processors were described by the same parameters, β_{cmax}, β_{phys} and K_r as in the distinct node model and the processing bandwidth was assumed to take the same form:

$$\beta_c = \beta_{cmax} - K_r \cdot (\beta_n + \beta_f)$$

as in the model for the distinct node processor.

The data flows in the homogeneous model are largely algorithm dependant and not predictable in the way that they are in the non-homogeneous scheme making prediction of the processing bandwidth spent towards movement of data and the remaining processing bandwidth considerably more difficult.

If all processors are fed with data the simplifying assumption that processing of events is sufficiently similar for β_f to be insignificant or to have a cancelling effect with β_f of adjacent nodes and that each column behaves largely independently is made. Only one of such columns of processors need be considered.

The computing bandwidth at the processor *1* in the column is given by the equation

$$\beta_{c1} = \beta_{cmax} - K_r \cdot \beta_{n1}$$

and the bandwidth of data sent down to the next layer (β_d) is given by

$$\beta_{(1+1)} = \beta_d = \beta_{n1} - \beta_{c1}$$

The bandwidth into any layer of the column is given by

$$\beta_{n1} = \beta_{n1} - \sum_{1=1}^{L-1} (\beta cmax - Kr.\beta_{n1})$$

the recursive nature of this relationship does not allow a general equation for a maximum value of β_{n1} in terms of L, β_{cmax} and K_r to be readily found, the values of β_{n1} may be calculated recursively from β_{n1}, K_r and β_{cmax}. It is also

possible to calculate β_{n1} recursively in terms of L, K_r and β_{cmax}, a plot of total input bandwidth for values of R=1, $1 \leq L \leq 20$, $0 \leq K_r \leq 1$ and $\beta_{cmax} = 1$ is shown in fig 3.12; this shows clearly that, like the distinct node processor case, relatively small values of K_r give rise to serious degradation of the processing throughput.

The values of β_{n1} at maximum loading may be obtained by considering first a single processor. The computing bandwidth is given by

$$\beta_{c1} = \beta_{cmax} - K_r \cdot \beta_{n1}$$

and since no data is sent on to another processor $\beta_{n1} = \beta_{c1}$ so the maximum processing bandwidth is given by

$$\beta_{c1} = \frac{\beta_{cmax}}{1+K_r}$$

Considering next a processor above the one just considered, in this case the processing bandwidth is given by

$$\beta_{c(1-1)} = \beta_{cmax} - K_r \cdot \beta_{n(1-1)}$$

and $$\beta_{n(1-1)} = \beta_{c(1-1)} + \beta_{d(1-1)}$$

$$= \beta_{c(1-1)} + \beta_{n1}$$

combining these gives

$$\beta_{c(1-1)} = \beta_{cmax} - K_r \cdot \beta_{c(1-1)} - K_r \cdot \beta_{n1}$$

$$\beta_{c(1-1)} \cdot (1+K_r) = \beta_{cmax} - K_r \cdot \beta_{n1}$$

$$\beta_{c(1-1)} = \frac{\beta_{cmax} - K_r \cdot \beta_{n1}}{1+K_r}$$

$$\beta_{n(1-1)} = \frac{\beta_{cmax} - K_r \cdot \beta_{n1}}{1+K_r} + \beta_{n1}$$

Applying this equation recursively allows the maximum value of β_{n1} to be obtained. Achieving this maximum flow through the processors is very much dependant upon the algorithm used for communication.

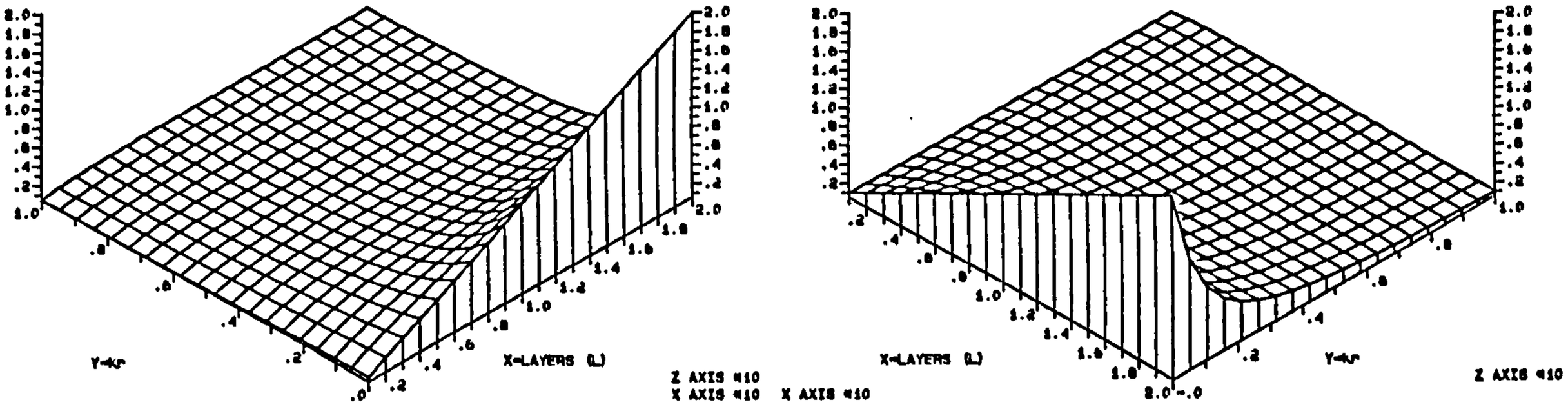
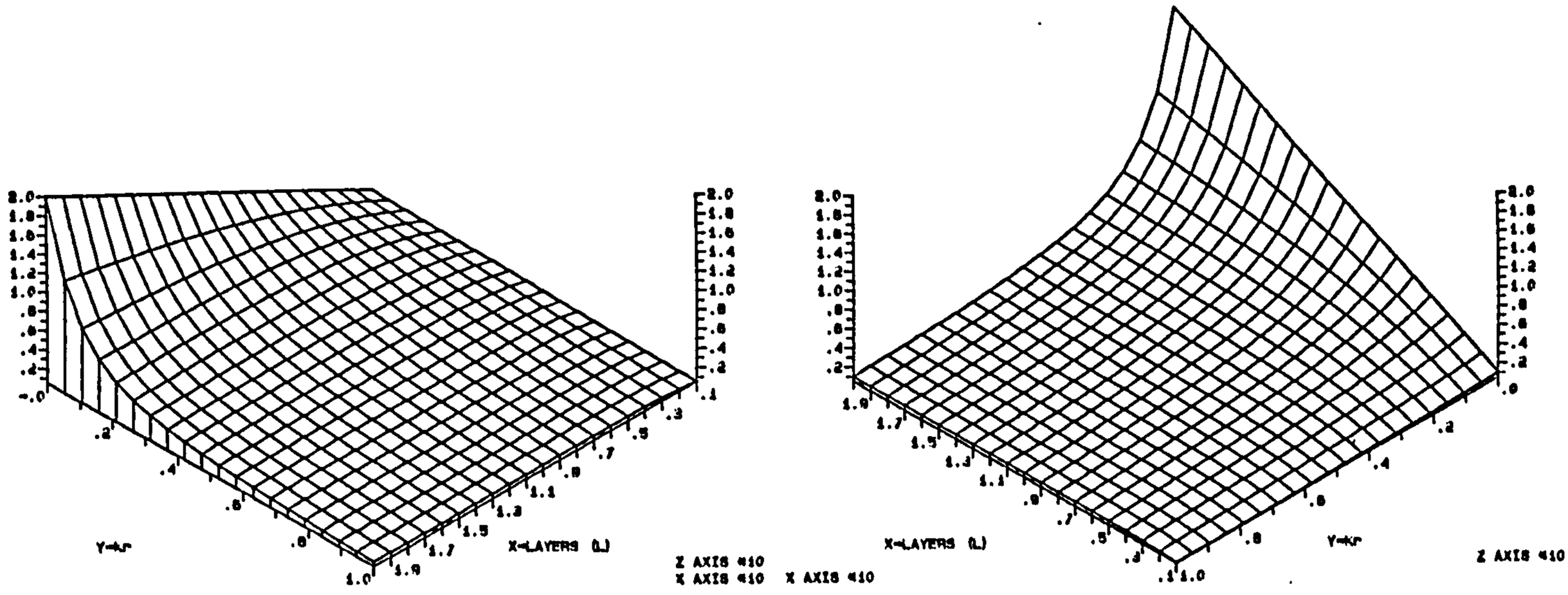


FIG. 3-12 INPUT BANDWIDTH FOR A HOMOGENEOUS COLUMN OF PROCESSORS Z=MAXIMUM BANDWIDTH (EVENTS/TIME).



This model describes the behaviour of one column of processors within a homogeneous cylinder of processors fed with data at all of the processors in the top most layer of the cylinder. One of the advantages of the homogeneous cylinder is the ability to feed data into the system at any or all points as desired though this flexibility makes useful performance predictions rather more difficult.

A cylinder fed with data is likely to feed the larger part of its data downwards, with minimal horizontal communication, giving a total input bandwidth close to R times the input bandwidth for a single column calculated above.

If the cylinder is fed at less than R of the processors the balance of communications assumed above would be broken. The bandwidth calculations above are related to the path length of the data, the total bandwidth of the cylinder not fed at R nodes will be dependant upon the mean path length of the data through the system. The mean path length could possibly be used in place of L to provide an estimate of the maximum computational bandwidth.

3.5 Relationship To Other Topologies

The topology proposed is that of a cylinder of processors connected as a direct connection network computer. The cylinder can be regarded as a rectangular array with additional connections to produce rings in only one direction. These cyclic connections give an advantage over a rectangular array (or a tree) in that data will not reach a 'dead end' if not accepted by a processor but will cycle round the ring until

either accepted by a processor or sent to another ring, similar design considerations have been outlined by Hyvärinen[74].

This development could be taken a stage further, with connections from the top of the cylinder to the bottom to give a toroidal surface[152]. However this arrangement, though having cyclic paths both vertically and horizontally, would require nodes to be fed with data to have four communication links plus a link to fed data into the node. Though an additional link would not be technically unfeasible, for all nodes to be identical in physical construction and not feed data in at all nodes could leave something approaching 20% of the communication hardware redundant, resulting in reduced cost effectiveness.

Inspection of the way that the cylindrical arrangement is used, with the decreasing bandwidth fed to the lower nodes, would perhaps suggest that an inverted pyramid structure would provide a better mapping to the problem. Though this is probably true to some extent this would leave some nodes with links unused if the intended identical nodes were used to construct the system, with a consequent loss of path redundancy and no additional processing could be anticipated.

3.6 Relationship to Other Systems

The distinct node scheme presented uses similar processing nodes and operating system mechanisms to that described by Ansade *et al*[8], the asynchronous operation of separate processors being common to both. Many of the synchronisation and similar problems studied by Ansade and his

colleagues are avoided by the homogeneous processing scheme.

The homogeneous processing scheme is similar in many respects to the scheme used by Glendinning and Hey[57], in this scheme a 'reserve' of TRANSPUTERS are used to process events independently, either for theoretical simulation or for data processing, using the same program in each processor operating on different and independent data. The communication structure used in this work was a simple chain of processors in a linear arrangement and also involved a distinct master TRANSPUTER and, unlike the homogeneous scheme presented here, does not offer any fault tolerance.

Both the distinct node and homogeneous schemes are driven by the arrival of data to process and are in essence a large grain dataflow computer[40,111,81], the grain being set at the level of the entire program and the parallelism being introduced through replication of the dataflow graph. Many of the problems even of macrodataflow[58] are circumvented by this choice of grain size.

3.7 Fault Tolerant Nature of a Cylinder

3.7.1 Path Redundancy

The cylindrical structure proposed has a considerable amount of path redundancy, for any data that would have been sent through a failed link may be sent down to a lower ring and for any failed vertical link data may be passed around the ring to a node with a functioning vertical link before being passed down. Though some links must carry additional data traffic as

a result of this re-direction the lower layers will not be heavily loaded unless this has specifically been arranged to be the case and alternatively it is possible to ensure that links are slightly underloaded to permit re-direction of data without causing saturation.

3.7.2 Link Failure

3.7.2.1 Distinct Node Case

In the case of distinct processing nodes the failure of a link must cause re-direction and the communication algorithm must permit data (as opposed to results) to be sent both up and down in the vertical direction, involving the mixing of data and results (this may allow deadlock situations to be created).

3.7.2.2 Homogeneous Node Case

In the case of a homogeneous processor the processor to which data is sent is of no consequence, however if a horizontal link fails this could produce a processing 'shadow' where the following processors in the same ring do not receive any data, this effect should not be evident if all of the uppermost processors are fed with data and may be simply overcome by allowing data to be sent upwards as well as downwards, again this would involve mixing data and results though there should be no attendant problems of deadlock as events are completely independent and may be routed round queues of data and results.

3.7.3 Processor Failure

Failure of a processing node such that is does nothing would be identical to failure of the four links connected to it. If a fed node fails then no data will be fed to the system at that point and if only one node is fed with data then this is a possible single point failure of the system.

3.7.3.1 Data Sink Failure

There are three other principal cases of failure of a processing node. A processor may fail such that it will accept data and not take any corresponding action, acting as a sink for data. This situation would not produce any symptoms readily detectable by the other processors, however this could be dealt with by using suitable error detection software mechanisms[4]. This could ensure that if a processor were not providing finished results then no further data items would be accepted.

3.7.3.2 Data Source Failure

A processor may also fail such that it generates output data that is not the result of processing input data items, this could take two forms; a random stream of 'noise' data or a stream of correctly formed processed events but carrying meaningless values. The former may be detected and dealt with (simply ignoring the 'noise' would be an effective remedy) by requiring valid data to be framed correctly ie with an appropriate header and correct checksum etc, the probability of a valid frame being generated being very small. The latter

case is somewhat more difficult to deal with since no readily detectable symptoms are produced as far as the other processors are concerned. The erroneous behaviour of a processor may be detected and dealt with using suitable software error detection mechanisms within the failed processor[4]. This could ensure that if a processor were producing unwarranted output data then the processor would shut itself down or take some other action. In this case, as in that where data is accepted without producing a corresponding output, a simple count of data items accepted as compared with results produced with appropriate action being taken if an unacceptably large discrepancy arises could be used to detect erroneous behaviour.

3.7.3.3 Faulty Processing Failure

The final and most difficult form of processor misbehaviour to detect is that where data items are accepted for processing and the results sent on correctly but the actual processing performed is erroneous. This could be dealt with using N Modular Redundancy (NMR)[4], this would require either N processors at each node or a rigidly defined communication and interaction scheme between N processors of the system, both methods involving a reduction in processing 'power' by a factor of approximately N as compared with a system not implementing NMR. Having the N processors at one node would be a more efficient solution as it would avoid the use of the network to compare results with the attendant problems of network loading and routing. Having all N processors in one node allows a simple comparison of the results, results over which there is a disagreement may simply be destroyed avoiding the need for

rollback[4,80], this loss being small in comparison to the total number of events processed. However, this situation cannot be left to continue unabated as if the processor continually destroys results it will become a 'sink' for data as in section 3.7.3.1 above. In the case of the homogeneous processor the action taken on detecting an error by the NMR implementation may simply be to shut the processor down, because of the inherent fault tolerance of the network no action would be necessary to maintain the function of the system. This would result in the destruction of an event, however at a processing rate of 10 000 events per second (an order of magnitude less than that targetted which may or may not be regarded as significant depending upon the experiment) the resulting error after 1 second would be 0.01% of the total. In such a situation where a result is not essential following a failure, only the detection of failure, 2 Modular Redundancy could be applied with a consequent minimising of cost.

3.7.4 No-Action Resilience to Faults

In the case of distinct processors the NMR error detection mechanism could be used to provide corrective action, because of the mapping of event processing onto the system a processor cannot simply be shut down without preventing the processing of a particular group of events.

For a homogeneous processor however, provided that the processing nodes can be relied upon not to misbehave in one of the ways mentioned above then the network would ensure graceful degradation of the processing performed without the system or

the user having to perform any corrective or reconfiguration action, unlike most fault tolerant schemes which require the system to repair itself after a fault[4,15,53,14]. In fact only the node that has sustained failure need be 'aware' that a fault has occurred; an inactive processor would appear to any processor that attempted to send it an event as already busy, if this were the case then the sending processor would simply send the data along the alternative output link and the data would pass around the fault through neighbouring processors, this mechanism could lead to a processing 'shadow' as mentioned above.

This graceful degradation would not continue indefinitely as the failed processors and links could soon form a cut-set within the topology and isolate a possibly large section of the network. For this reason it would be desirable to allow for the provision of replacement of faulty processing nodes within the system with the system running.

3.7.5 On-Line Replacement of Processors

3.7.5.1 Reprogramming

For a system consisting of distinct processing nodes a replaced processor would require reprogramming with its location within the network and the routing and processing programs for the data it would be intended to deal with. Unless each processor were to store the programs for its immediate neighbours, considerably increasing the memory requirements, the programmming information would have to be sent through the network interfering with its normal operation.

In this instance, with the implementation of NMR, replacement of processors should rarely be required and re-initialisation of the system following processor replacement would probably be the most cost effective strategy.

For a homogeneous processor many of these problems would not exist or would be significantly reduced in complexity. Since all processors execute exactly the same program no additional memory is required for storage of neighbour nodes programs as this is inherent in the nature of the system. All that would be required would be the facility for processors to request a copy of the program from one of their nearest neighbours.

3.7.5.2 Hardware Requirement

Both of these schemes require specific capabilities of the hardware not provided for by all communication schemes. When a processor is replaced the adjacent processors must be informed of the availability of the replaced processor, this is easily achieved at the software level. Problems may arise with some types of hardware communication protocol however, simple handshake lines present no problem but protocols that implement an acknowledge mechanism at the hardware level using transmitted acknowledge packets [78,1061109567] require both communicating nodes to be restarted if one the the nodes suffers a failure as otherwise one of the nodes could possibly have lost an acknowledgement and be left unable to send data. If the facility is not available to restart individual communication circuits within a processing node then all

communication must be restarted and this will have a knock-on effect ultimately requiring restarting the entire system. This problem does not arise at the software level since the state of any part of the software mechanism can be independently set as appropriate.

3.8 Distributed Depth First Search

Several situations arise where it is desirable to interrogate all of the processors of the network in turn, for testing of the processing nodes or to verify that the connection topology is correct. An algorithm was required to enable this to be carried out in a systematic fashion. Two principle established algorithms for searching of graphs exist, these are the depth first search (DFS) and the breadth first search (BFS)[132,41], though others do exist. Both of these algorithms are well known for searching graphs using a monoprocessor and both create a spanning arborescence of the graph, algorithms also exist for searching of such graphs in parallel[67]. Neither of these were what was required, this being for the algorithm itself to be reproduced at all vertices of a graph and for the active node to progress through the graph as the node under consideration would in the case of the monoprocessor algorithm. The data exchanges within the monoprocessor algorithm would have to be replaced with transfer of data across the network making up the graph. The depth first search was developed into a distributed algorithm though the same approach could be applied to the breadth first search algorithm. Additional data exchanges were added to the algorithm to provide a more complete description of the

interconnection graph than a spanning arborescence.

3.8.1 Development of the DDFS Algorithm

The starting point for the development of the algorithm was the algorithm as quoted by Sedgewick[132].

```
procedure dfs;
  var now,k:integer;
      val:array[1..maxV]of integer;
  procedure visit(k:integer);
    var t:link;
  begin
  now:=now+1;  val[k]:=now;
  t:=adj[k];
  while t<>z do
    begin
    if val[t^.v]=0 then visit(t^.v);
    t:=t^.next;
    end
  end;

begin
now:=0;
for k:=1 to V do val[k]:=0;
for k:=1 to V do
  if val[K]=0 then visit(k);
end;
```

This algorithm is intended to scan an adjacency list held in a monoprocessor and a great deal of the detail is specific to the representation. The important details are the initiating call to visit(k) (in the algorithm above all of the nodes stored are tested to allow non-connected graphs to be scanned), the tests of val[t^.v] for all of the nodes to which the node being scanned is connected and the resulting visit to the connected node.

The distributed algorithm is not required to visit its connected nodes until it is visited by another node, the initial visit being generated by an external driver. The initialisation of the assigned values was performed by each

node declaring itself as unvisited on commencement of the program. The procedure visit is recursive the calls of the procedure being replaced by interactions between processors, the procedure visit was split into two parts, visit call and visit answer, a call to visit being represented by the interaction between the procedure visit call of one node and visit answer of the next. Since vertices of connectivity ≤4 were under consideration the links t were represented as the integers 1 to 4. These changes give a first outline for the modified algorithm.

```
procedure ddfs;
  var now, val:integer;

  procedure visit_call(t:link);
  begin
  send now to visited node;
  while waiting
  if val_requested  then
    send val to requesting link;
receive new now from visited node;
end;

procedure visit answer(t:link);
begin
receive value of now;
for t:=1 to 4 do
  begin
  if valreq(t) = 0 then
    visit call(t);
  end;
return new value of now;
end;

begin
repeat
  begin
  wait for something;
  if visited then
    visit answer;
  if val_requested then
    send val to requesting link;
  end
until forever;
end;
```

The function valreq is one to return the value of the node (if any) connected to link t. This would involve some interaction between the two processors in addition to visit call and visit answer interactions.

3.8.2 Circuits and Self-Loops

Inspection of the behaviour of the algorithm reveals an inability to cope with some cyclic graphs where the node that is next to be visited is already part of the visit call/answer chain. Unless the algorithm can respond to val requests while engaged waiting for the response from its visit call cyclic graphs cannot be scanned correctly, this is the purpose of the code to respond to val requests in the visit_call section of the algorithm though this still does not permit self-loops to be scanned.

3.8.3 Use of Multi-Programming

With the increasing availability of multiprocessing systems it was possible to consider an algorithm using such a facility, a separate process could be set up to respond to val requests at any time allowing all graphs to be scanned correctly. With this modification the algorithm written in a pseudo parallel pascal using parbegin/parend to indicate parallel execution[44] becomes:

```
procedure depth first search;
var now, val:integer;

  procedure valresponse
  begin
  wait for a request for val;
  send val to the appropriate link;
  end;

  procedure ddfs;

    procedure visit call(t:link);
    begin
    send now to visited node;
    receive new now from visited node;
    end;

    procedure visit answer(t:link);
    begin
    receive value of now;
    for t:=1 to 4 do
      begin
      if valreq(t)=0 then
        visit call(t);
      end;
    return new value of now;
    end;

      begin
      wait for something;
      if visited then
        visit answer(t);
      end;

  parbegin
  valresponse;
  ddfs;
  parend;
```

All that the system needs to function is a visit call to one of the nodes.

3.8.4 Use of DDFS for testing

The algorithm above provides little useful information, to be useful some additions need to be made to return some information to the interrogating node. To allow the connection pattern to be tested the link connections must be added to the information passed back with the new value of now, with the

recursive nature of the visit call/answer this information will ultimately be passed back to the interrogating node. If testing of the processors is required this can be performed as part of the DDFS, the test results may be passed back to the interrogating node with the assigned value and the link connection information. This assumes of course that the processing nodes are actually capable of running the test programs, if not then the node will not appear in the graph returned, a result that itself indicates a fault. A large amount of work on fault detection schemes in the presence of faulty nodes has been done[118,16,32,63,27,105] and such a scheme could be used to test the system. The purpose for which the search algorithm was principally developed was to allow the initial correct connection pattern of the network to be verified since a simple 'patch panel' type of construction was envisaged in the interests of flexibility and ease of replacement of processors. The implementation of the algorithm is explored more fully in chapter 6.

CHAPTER 4

4 SIMULATION OF PROCESSING STRUCTURES

4.1 Introduction

The ring and cylindrical structures considered in the previous chapter were based on regarding the system purely as a set of data flows. This chapter describes simulations of such structures which were undertaken as a precursor to hardware development to explore various aspects of the behaviour and performance of the computing structures proposed.

The first stage involved the simulation of a single ring, experience gained with the ring was then applied to the selection and design of simulations of cylinders of processors. The programs used for the simulations are to be found in appendices 1 - 4.

4.2 Processing Element Model

The processing elements of the computer structure under simulation were modelled as a sequential machine[41], with the state diagram show in fig 4.1.

Initially the state of a processor would be zero, this state being used to indicate that the processor is idle and waiting to accept further data. The sequential machine is then set to a state when it commences processing of an event, the state assigned representing the complexity of the computational task, as the event is processed the processor takes a

progressively lower state until the state zero is reached, i.e. the processor is idle once again and can accept a further event to be processed.

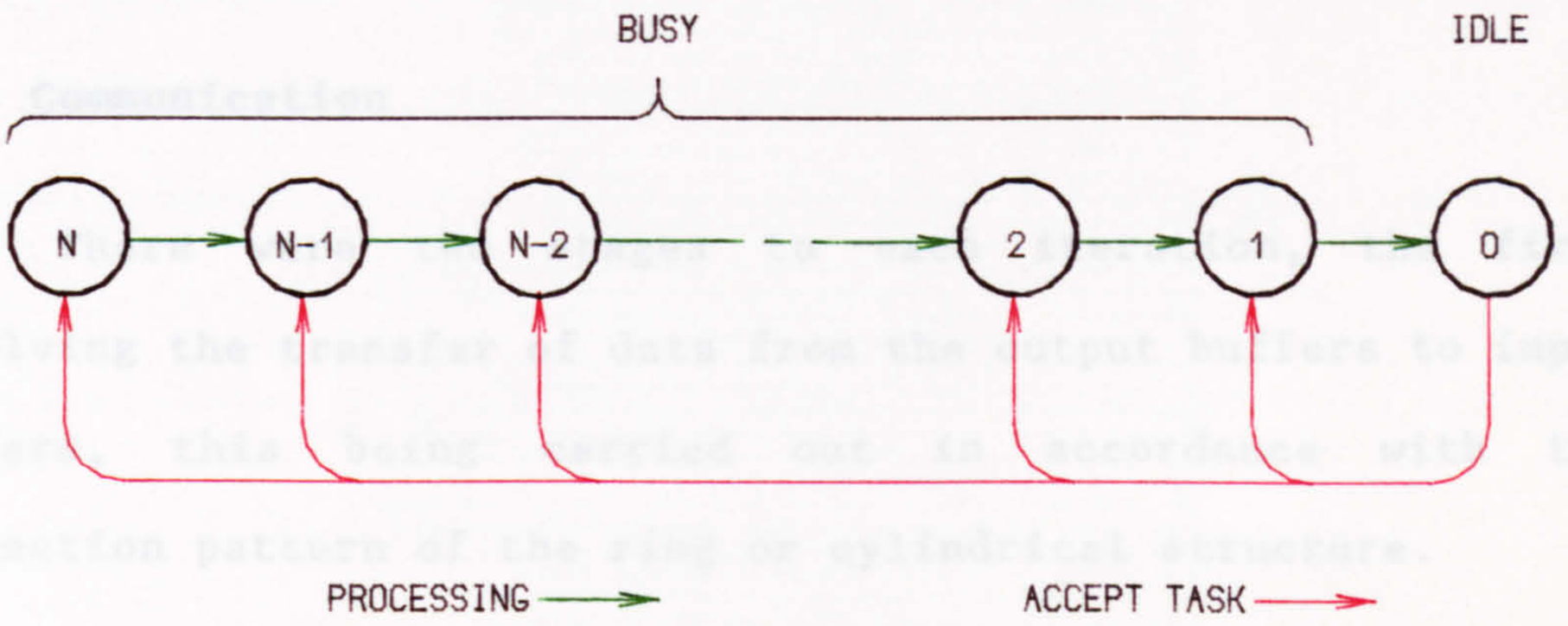


FIG. 4-1 STATE DIAGRAM OF THE PROCESSOR MODEL USED FOR SIMULATION.

4.3 Communication Hardware Model

The communication structure supporting these processors was modelled at the register transfer level[121] as a set of input or output buffers within each processor, these being connected to the appropriate buffer in a separate node by a communication link. The action of the communication hardware was represented by a transfer of data from the output buffer of one processor to the input buffer of another, with necessary changes to internal flags to indicate the state of the buffers.

4.4 Iterative Nature of the Simulation

The action of the communication mechanism and the processing elements were simulated in an iterative fashion, each iteration representing a time unit (Equitemporal

Iteration[121]). This approach produced a system model that would operate with globally synchronised communication and computation, this could allow some undesirable aspect of the real system's behaviour to be overlooked.

4.5 Communication

There were two stages to each iteration, the first involving the transfer of data from the output buffers to input buffers, this being carried out in accordance with the connection pattern of the ring or cylindrical structure.

4.6 Data Input Mechanism

At this stage fresh data was placed in the upper input buffers of the nodes of the ring which were empty, this achieved the function of the data input mechanism. Data was placed into any input buffers that were empty so that buffers were filled as fast as they were emptied, this allowed the computing structure behaviour to be simulated without interference from the effects of the input device.

Several different schemes of inputting data were implemented depending upon the simulation being carried out. In the distinct node set of simulations the data represented the node in which the event would be processed and the processing task incurred by the processor on receipt of this event would either be a user defined constant or one of a range of randomly selected values, the distribution of the random numbers used for these simulations is shown in fig 4.2.

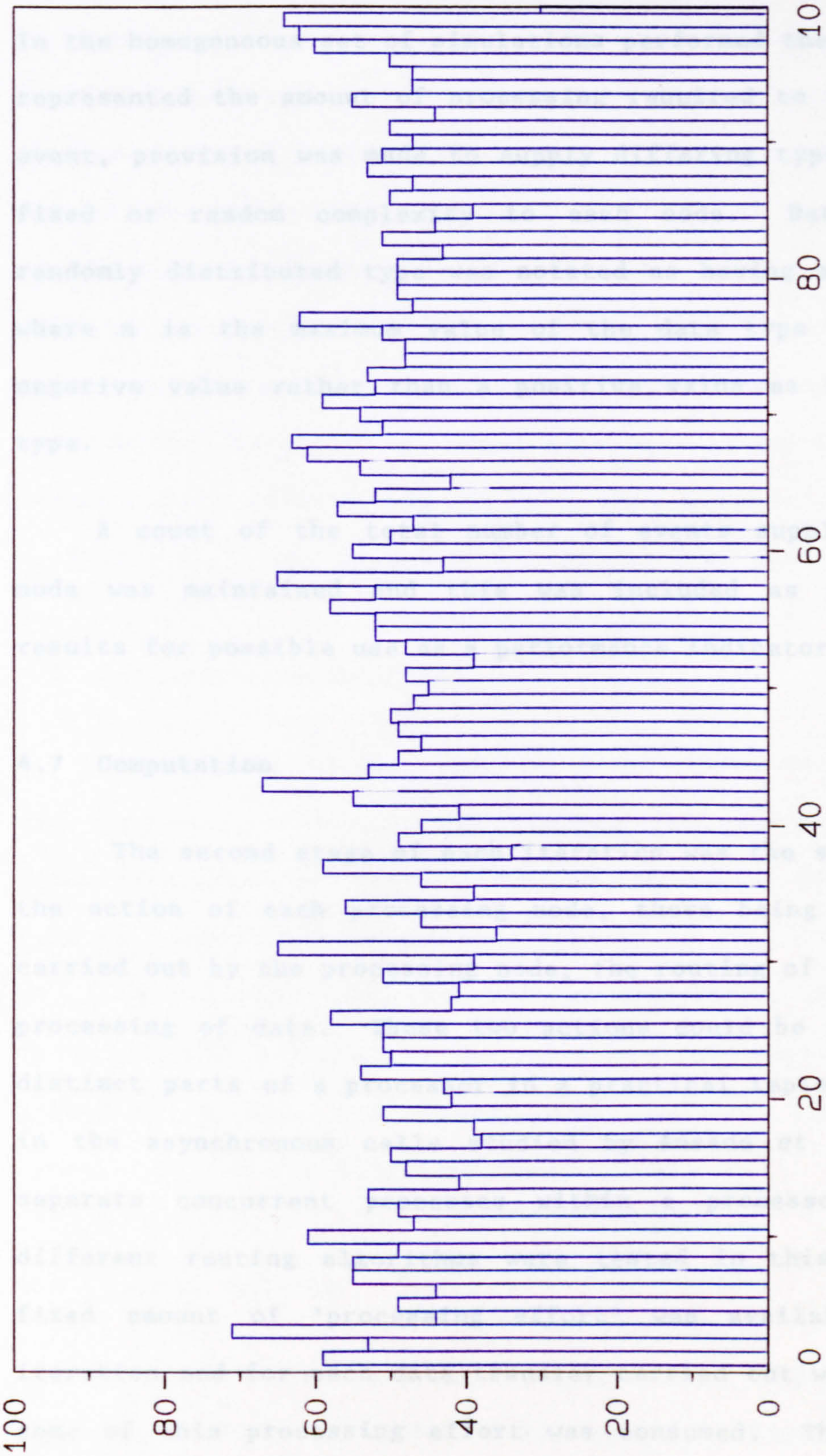


FIG. 4-2 DISTRIBUTION OF

The required processing effort to completely deal with the data is referred to as the data type in the discussion that follows. In the homogeneous set of simulations performed the data simply represented the amount of processing required to complete the event, provision was made to supply differing types of either fixed or random complexity to each node. Data having a randomly distributed type was notated as having a type of R*n* where *n* is the maximum value of the data type and given a negative value rather than a positive value as for constant type.

A count of the total number of events supplied to each node was maintained and this was included as part of the results for possible use as a performance indicator.

4.7 Computation

The second stage of each iteration was the simulation of the action of each processing node, there being two actions carried out by the processing node, the routing of data and the processing of data. These two actions could be performed by distinct parts of a processor in a practical implementation as in the asynchronous cells studied by Ansade *et al*[8] or by separate concurrent processes within a processor. Several different routing algorithms were tested in this scheme. A fixed amount of 'processing effort' was available in each iteration and for each data transfer carried out within a node some of this processing effort was consumed. The action of processing was simulated by decrementing the state of the processor by the amount of 'processing effort' available

remaining after the communication had been performed. On completion of the processing of an event, indicated by the processor entering state zero, a count of the total number of events of each type was incremented as appropriate, to provide an indication of performance in terms of completed events. In all of the simulation discussions that follow the value taken as the maximum 'processing effort' was 4 units per iteration with one unit deducted for each data transfer performed. The amount of processing consumed was perhaps rather high, this was made so deliberately to exacerbate any potentially serious communication effects and to make such effects show up more readily and clearly in any of the simulations performed.

4.8 Performance Indicators

In addition to the counts of events taken into the system and completed events as performance indicators a 'Weighted Total Processed' was kept. This was the sum of events of each type processed with a weighting applied to each type according to the processing time required for completion of that event type. This is effectively the total usefully applied number of processor cycles and takes into account variations in processing time for different types of data.

4.9 Startup Effects

On commencement of the simulation all of the nodes were in an empty state as a real machine would be immediately after having been programmed. It would have been possible to initialise all of the nodes to some form of 'steady state'

value to allow simulation of the steady state flow within the system without interference from any other effects, it was however considered undesirable to do this; this would avoid any tendency to influence the outcome of the simulation by initialisation of the system into an artificial state that perhaps would not occur in practice and would also determine that no untoward conditions arose during initialisation as data percolated through the various systems under study. For these reasons the simulated systems were started up from an empty state, for simulation intended to observe the steady state throughput of the system, as most of them were, this would produce an apparent lowering of the throughput of the system as some processing cycles would be required simply to fill the system with data. This would be particularly pronounced for extremely large systems and small numbers of iterations. The larger part of the simulations were carried out over 1000 iterations, this was considered to be large enough to make the initialisation insignificant for most of the systems investigated. For a 100 node system this would require something in the region of 20-200 iterations (dependant on the diameter of the interconnection graph and the number of nodes fed with data.) to fill the buffers and would produce ≃20% error. It is possible that this is a pessimistic estimate as some processing would be performed during these initialisation cycles and most of the systems simulated were of a size considerably less than 100 nodes. The figure of 1000 iterations proved to be something of a practical limit since the time taken to run these simulations on a Prime 9955 minicomputer took several minutes for each simulation with a

particular system shape and data type.

4.10 Preferential Communication Algorithms

Communication was given preference in the computing scheme implemented, this decision is readily justified by considering the converse case. If 'processing effort' were directed towards completion of the current event then communication would only take place, if at all, when the processor was idle. In any situation other than that where all processors were fed with data individually this would inevitably lead to processors remaining idle while data was waiting to be processed. One simulation of this type of algorithm was carried out and performance was equal to that of a single processor regardless of the number of processors available. Following this result this line of investigation was not pursued further.

4.11 Distinct Node Simulations

The first set of simulations carried out were a simulation of the scheme developed where data types were to be routed to particular nodes for processing. Initially rings were simulated, since the cylindrical structure was a development from the ring and it was believed that experience gained with the ring simulation could provide indications of areas of interest with the cylindrical structure.

4.11.1 Data Routing Information

In all of these schemes the data supplied to the system was the address at which the data should be processed and was passed around until the data reached the correct processor. In the case of the ring structure this address was simply the number of the processor around the ring, in the case of the cylinder the address was the value: number round the ring + (100 × layer number) which allowed the layer and ring values to be sent as one integer value. The data was passed around the ring until the address of the data matched the 'identifier' of the processor node, once this situation occurred the data had been routed to the correct node and the processor was allocated a fixed amount of processing work to perform. All of the data packets in this scheme effectively had a fixed type. This fixed type could be changed by the user and if a negative value were used for the data type this would produce a random type, evenly distributed up to the value indicated.

4.11.2 Ring Simulation

4.11.2.1 Algorithms Investigated

The possible algorithms for movement of data within such a processing system were restricted only by the designers imagination and creativity. Algorithms where data were processed in preference to communication being performed were excluded from study for the reasons stated above in section 4.10. Two out of the many possibilities were selected as cases for study.

The algorithm was split into two sections of code, one taking data from one of the data streams for processing and the other taking data from one of the data streams and forwarding this on to the next node. These sections of code had a similar structure in both algorithms, if the space to which the data was to be sent was ready to accept more data then first one possible source and then the other was tested for the presence of data, any data found being passed on. This produced two algorithms, in one preference was given to new data coming in from above and in the second preference was given to data from the ring.

This essentially simple scheme would not suffice in the distinct processor system, some tests for a match between the data and the identity of the processor being required. This complicates the algorithm but not excessively so, data destined for processing at that particular node must not be sent past and data not destined for processing at that node must not be accepted for processing.

The algorithms as they appear in the programs (which are to be found in appendix 1) are shown below, short procedures and functions for common facilities have been written allowing this algorithmic style of program.

```
begin (* COMMS1 *)
if processor_idle then
  if new_data then
    if new_data_right then
      take_new_data
    else
      if ring_data then
        if ring_data_right then
          take_ring_data
        else
          (* NULL *)
      else
        (* NULL *)
  else
    if ring_data then
      if ring_data_right then
        take_ring_data
      else
        (* NULL *)
    else
      (* NULL *)
else
  (* NULL *);
if ring_ready then
  if new_data then
    if not new_data_right then
      new_data_on
    else
      if ring_data then
        if not ring_data_right then
          ring_data_on
        else
          (* NULL *)
      else
       (* NULL *)
  else
    if ring_data then
      if not ring_data_right then
        ring_data_on
      else
        (* NULL *)
    else
      (* NULL *)
else
  (* NULL *);
end;  (* COMMS1 *)
```

```
begin (* COMMS2 *)
if processor_idle then
  if ring_data then
    if ring_data_right then
      take_ring_data
    else
      if new_data then
        if new_data_right then
          take_new_data
        else
          ( * NULL *)
      else
        if new_data then
          if new_data_right then
            take_new_data
          else
            (* NULL *)
        else
          (* NULL *)
else
  (* NULL *);
if ring_ready then
  if ring_data then
    if not ring_data_right then
      ring_data_on
    else
      if new_data then
        if not new_data_right then
          new_data_on
        else
          (* NULL *)
      else
        (* NULL *)
  else
    if new_data then
      if not new_data_right then
        new_data_on
      else
        (* NULL *)
    else
      (* NULL *)
else
  (* NULL *);
end; (* COMMS2 *)
```

4.11.2.2 Performance of the Algorithms

The two ring communication algorithms were tested with a variety of input data types (see section 4.6) ranging from 1 to 50 inclusive, ie from purely communication bound through to a situation where the processing was computation bound, in

addition to constant data types similar simulations were carried out using data of even random distribution (see section 4.6). It was thought that any tendency for the processors to become 'synchronised' due to the regularity of the data would be avoided, obviously this could not be done for data of type 1 since only one value of the data type was possible.

4.11.2.2.1 Communication Bound

Taking the data types dealt with in order, fig 4.3 shows the variation in weighted total processing (WTP) achieved with different sizes of ring fed with data of type 1. For a ring fed with data requiring so little processing β_{cmax} can be taken to be 1 (since only one event may be accepted to be processed at each iteration in the simulation model) and K_r may be taken to be zero as regardless of the number of data transfers made within a node sufficient processing remains to process the event to completion. The value of β_{phys} would be 1 as only one event may be passed along each link in one direction. The bandwidths are all in events/iteration and K_r is a dimensionless constant. In the graph line 1 represents the behaviour of algorithm 1, line 2 represents the behaviour of algorithm 2 and line 3 shows the values predicted by the flow model of section 3.4.2.6.1. The most obvious feature of the graph is the behaviour of line 2, up to a 2 processor network the algorithm processes as much data as algorithm 1, however, beyond a ring size of 2 processors the processing drops to a very low level. Running the simulation for large numbers of iterations has shown that what is occurring is not merely process starvation, but total lockout or deadlock.

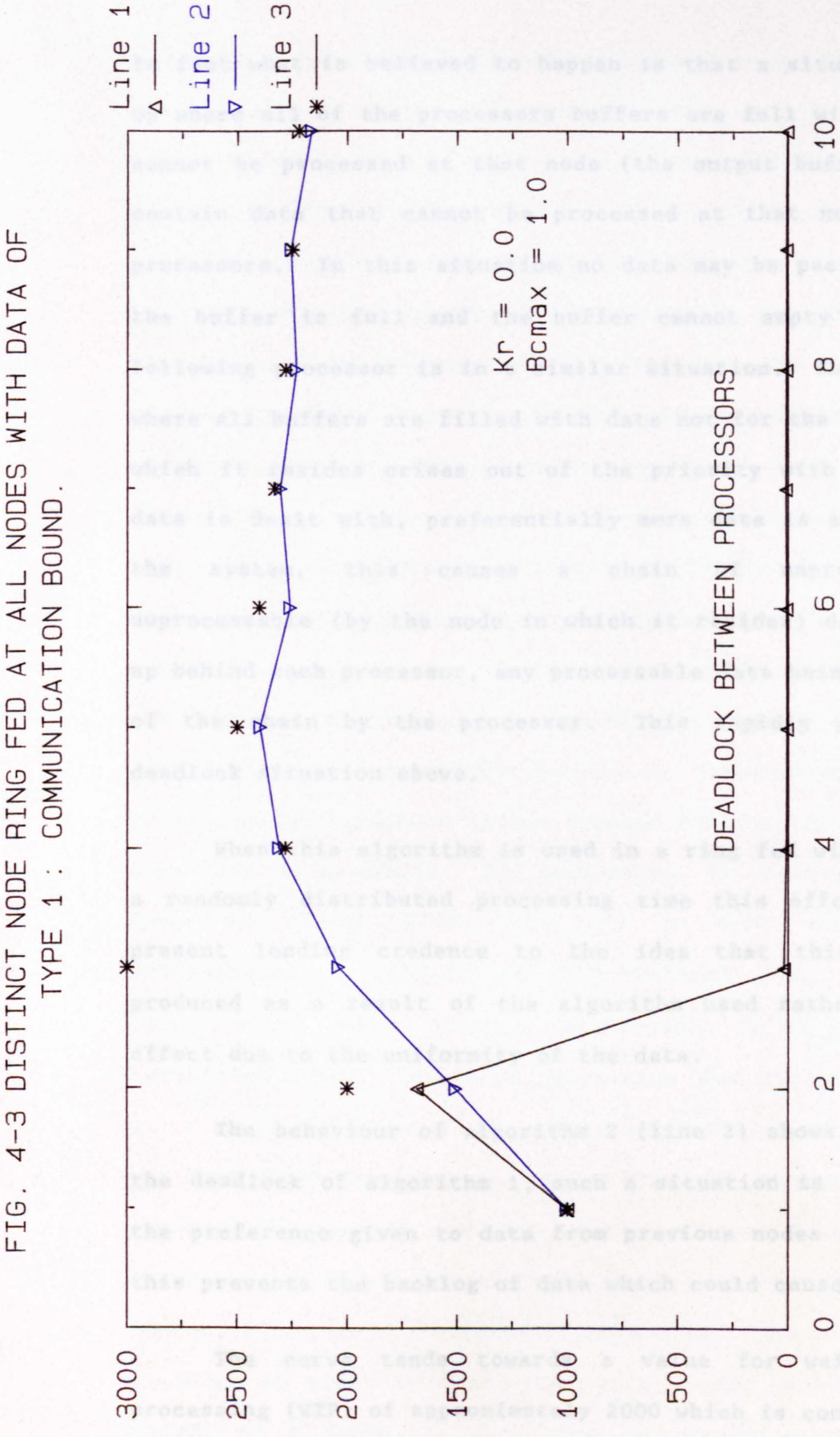


FIG. 4-3 DISTINCT NODE RING FED AT ALL NODES WITH DATA OF TYPE 1 : COMMUNICATION BOUND.

In fact what is believed to happen is that a situation builds up where all of the processors buffers are full with data that cannot be processed at that node (the output buffer can only contain data that cannot be processed at that node) for all processors. In this situation no data may be passed on since the buffer is full and the buffer cannot empty because the following processor is in a similar situation. This situation where all buffers are filled with data not for the processor in which it resides arises out of the priority with which input data is dealt with, preferentially more data is accepted into the system, this causes a chain of unprocessed and unprocessable (by the node in which it resides) data to build up behind each processor, any processable data being sifted out of the chain by the processor. This rapidly produces the deadlock situation above.

When this algorithm is used in a ring fed with data with a randomly distributed processing time this effect is still present lending credence to the idea that this effect is produced as a result of the algorithm used rather than some effect due to the uniformity of the data.

The behaviour of algorithm 2 (line 2) shows no signs of the deadlock of algorithm 1, such a situation is prevented by the preference given to data from previous nodes in the ring, this prevents the backlog of data which could cause deadlock.

The curve tends towards a value for weighted total processing (WTP) of approximately 2000 which is consistent with the predictions made in section 3.4.2.6.1 which suggests that the asymptotic value of the total bandwidth as $R \rightarrow \infty$ is $2 \cdot \beta_{phys}$

events/iteration for events of type 1 and over 1000 iterations gives a total processed value of 2000 'processing units'. Another main feature of this curve is the shifting of the peak from the predicted position at R=3 to a position nearer to R=5. Consideration of the factors that should induce such a peak, ie that of a balance between the bandwidth of data passed around the ring and the bandwidth of data fed into the ring such that the maximum input flow is obtained without causing saturation of one before the other, implies that there is a tendency for there to be a greater bandwidth of data transmitted around the ring rather than into it with a consequently larger ring size being achievable before both bandwidths reach a simultaneous limit. A ring size of 5 would imply that the bandwidth around the ring is twice that of the bandwidth into the ring, ie $\beta_f=2\cdot\beta_n$ from (i) of section 3.4.2.6.1. Considering the values of processing bandwidth allows further insight into the systems behaviour to be gained. The peak value, ie total processing at R=5, is almost 2500, that is a mean value of 0.5 events/iteration supplied to each vertical link, which is the value which will produce saturation ($\beta_f=\beta_{phys}$) of the horizontal links. The total processing when R=3 is ≃2000 which is a mean value of 0.66 events/iteration supplied to each vertical link, producing a mean flow of approximately 0.66 events/iteration in the horizontal links. This behaviour occurs as a result of the algorithms inherent priority for communication in the horizontal direction. The algorithm will transfer an event from the horizontal input to the horizontal output of a node in preference to accepting new data and transferring this to the horizontal output. Of course if data

arrives at the node at which it is to be processed it will be taken for processing as soon as the processor become free for reasons outlined in section 4.10, though even in this case preference would be given to data arriving from other nodes in the ring. This behaviour of the algorithm forces data at the vertical input links to wait until there is no data at the horizontal input link or that the data at this link is not to be processed at that node. Despite the rather crude buffering scheme of the simulation the flow model is a reasonably good predictor of the form of behaviour exhibited by the system though the particular values are influenced by the details of the algorithm involved. In the experimental environment in which processing is to be performed the actual values need not be precise since the processing time for events will vary from one experimental set up to another as will the importance of processing time.

4.11.2.2.2 Intermediate Data Types

Two simulations of the ring were carried out with intermediate data types, data of types of 5 and 10 and randomly distributed data with mean values of 5 and 10 were used.

The simulation using data of type 5 had values of $\beta_{cmax} = 1/2$, $K_r = 1/4$ and $\beta_{phys} = 1$, the bandwidths being in events/iteration and K_r being a dimensionless constant. Examining the behaviour of the algorithms shown in fig 4.4, lines 1 and 3 show the behaviour of algorithm 1 and lines 2 and 4 show the behaviour of algorithm 2.

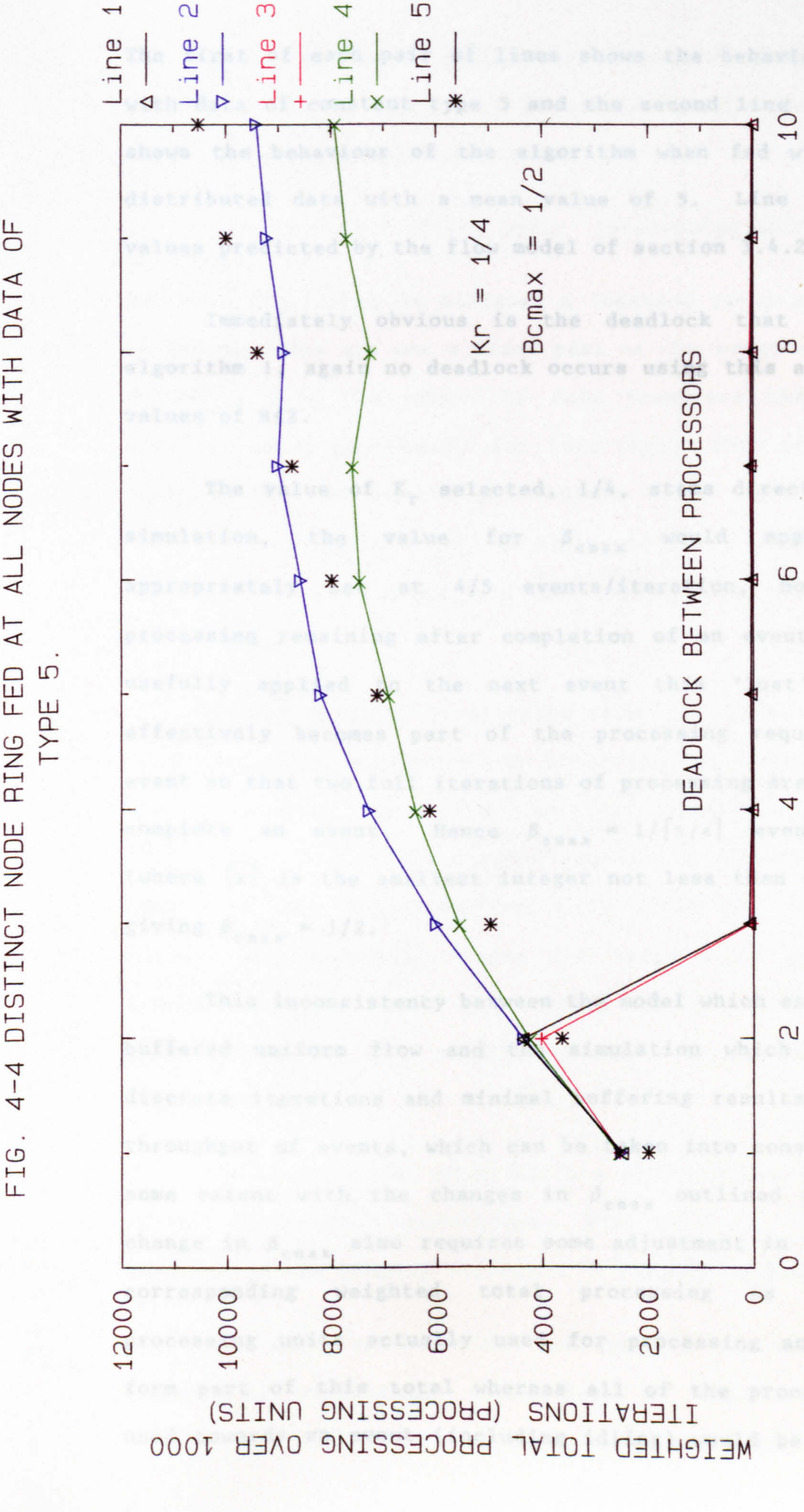


FIG. 4-4 DISTINCT NODE RING FED AT ALL NODES WITH DATA OF TYPE 5.

The first of each pair of lines shows the behaviour when fed with data of constant type 5 and the second line of each pair shows the behaviour of the algorithm when fed with randomly distributed data with a mean value of 5. Line 5 shows the values predicted by the flow model of section 3.4.2.6.1.

Immediately obvious is the deadlock that occurs with algorithm 1, again no deadlock occurs using this algorithm for values of $R \leq 2$.

The value of K_r selected, 1/4, stems directly from the simulation, the value for β_{cmax} would appear to be appropriately set at 4/5 events/iteration, however since processing remaining after completion of an event may not be usefully applied to the next event this 'lost' processing effectively becomes part of the processing required for an event so that two full iterations of processing are required to complete an event. Hence $\beta_{cmax} = 1/\lceil 5/4 \rceil$ events/iteration (where $\lceil x \rceil$ is the smallest integer not less than x) was used, giving $\beta_{cmax} = 1/2$.

This inconsistency between the model which assumes a well buffered uniform flow and the simulation which consists of discrete iterations and minimal buffering results in a lower throughput of events, which can be taken into consideration to some extent with the changes in β_{cmax} outlined above. This change in β_{cmax} also requires some adjustment in the expected corresponding weighted total processing as only those processing units actually used for processing an event will form part of this total whereas all of the processing units used towards an event (including idling) would be included in

the calculated total processing. What is required is an adjustment of the value used to convert events into processing units, an event's type must become

$$type' = type + idling\ units \quad (processing\ units)$$

however this cannot be assigned a constant value as the number of idling units are not a fixed part of the event and will vary depending upon the number of data transfers made while the event is being processed. For this reason this effect was not taken account of in the predictions made. Its effect would amount to a maximum of three processing units per event processed and with large data types this effect rapidly becomes insignificant.

In the light of predictions made with these values there is a good correlation between predicted and simulated values. The processing predicted is 0.4 events/iteration which corresponds to a weighted total processing of 2000. It can be seen that the simulation value is close to this predicted value. The asymptotic value for large R is again $2 \cdot \beta_{phys}$ events/iteration which corresponds to a value of 10 000 processing units, only slightly greater than the value obtained from simulation.

The shape of the curve bears a strong resemblance to that of the theoretical predictions. With $\beta_c < \beta_{phys}$ the effect of any bias in the routing algorithm is largely masked.

The same approach can be applied to inspecting the results of simulations using data of type 10, again the behaviour of both algorithms is shown on the graph fig 4.5.

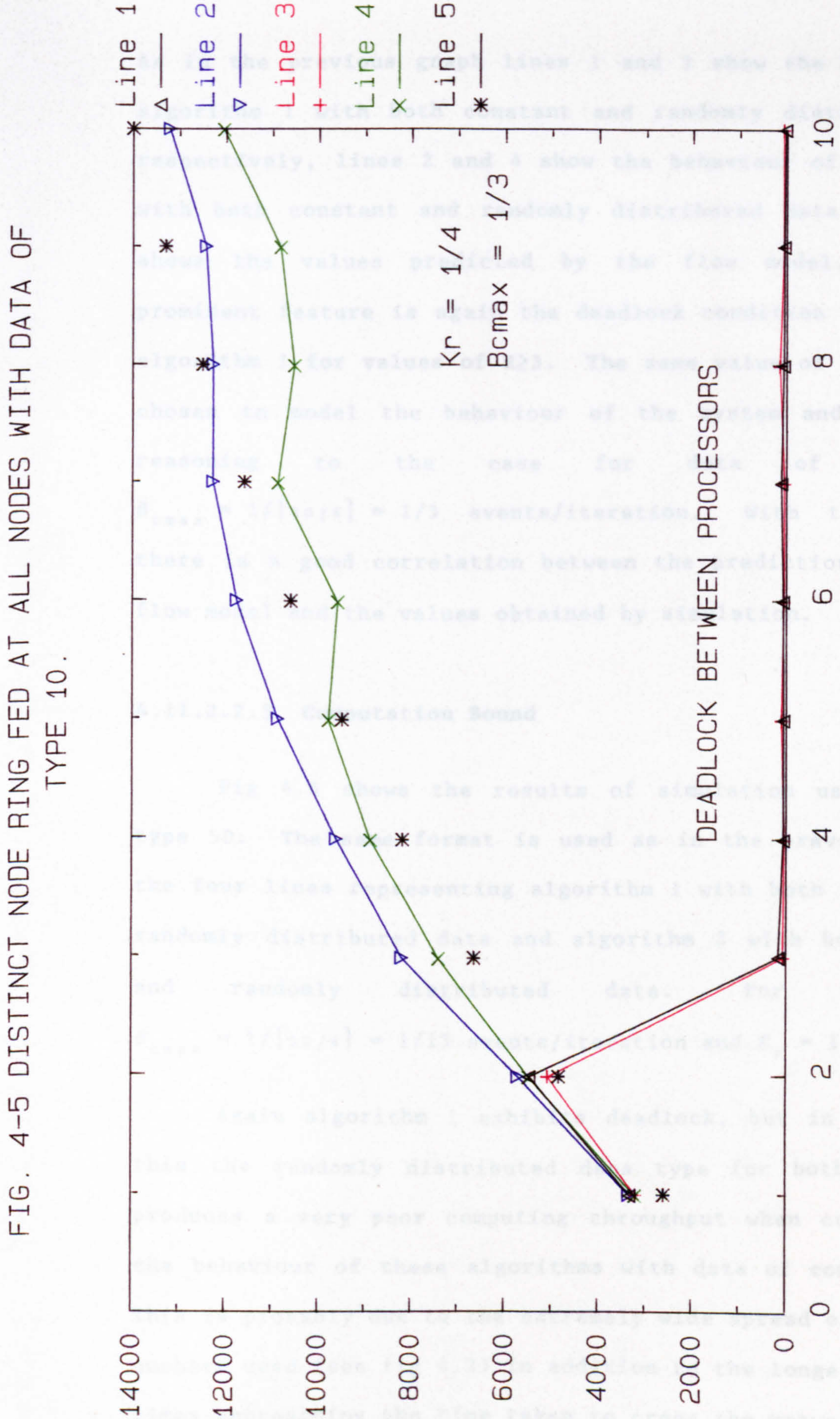


FIG. 4-5 DISTINCT NODE RING FED AT ALL NODES WITH DATA TYPE 10.

As in the previous graph lines 1 and 3 show the behaviour of algorithm 1 with both constant and randomly distributed data respectively, lines 2 and 4 show the behaviour of algorithm 2 with both constant and randomly distributed data and line 5 shows the values predicted by the flow model. The most prominent feature is again the deadlock condition arising with algorithm 1 for values of R≥3. The same value of K_r, 1/4, was chosen to model the behaviour of the system and by similar reasoning to the case for data of type 5, $\beta_{cmax} = 1/\lceil 10/4 \rceil = 1/3$ events/iteration. With these values there is a good correlation between the predictions using the flow model and the values obtained by simulation.

4.11.2.2.3 Computation Bound

Fig 4.6 shows the results of simulation using data of type 50. The same format is used as in the previous graphs, the four lines representing algorithm 1 with both constant and randomly distributed data and algorithm 2 with both constant and randomly distributed data. For this data $\beta_{cmax} = 1/\lceil 50/4 \rceil = 1/13$ events/iteration and $K_r = 1/4$.

Again algorithm 1 exhibits deadlock, but in addition to this the randomly distributed data type for both algorithms produces a very poor computing throughput when compared with the behaviour of these algorithms with data of constant type. This is probably due to the extremely wide spread of the random numbers used (see fig 4.2) in addition to the longer processing times approaching the time taken to cross the network.

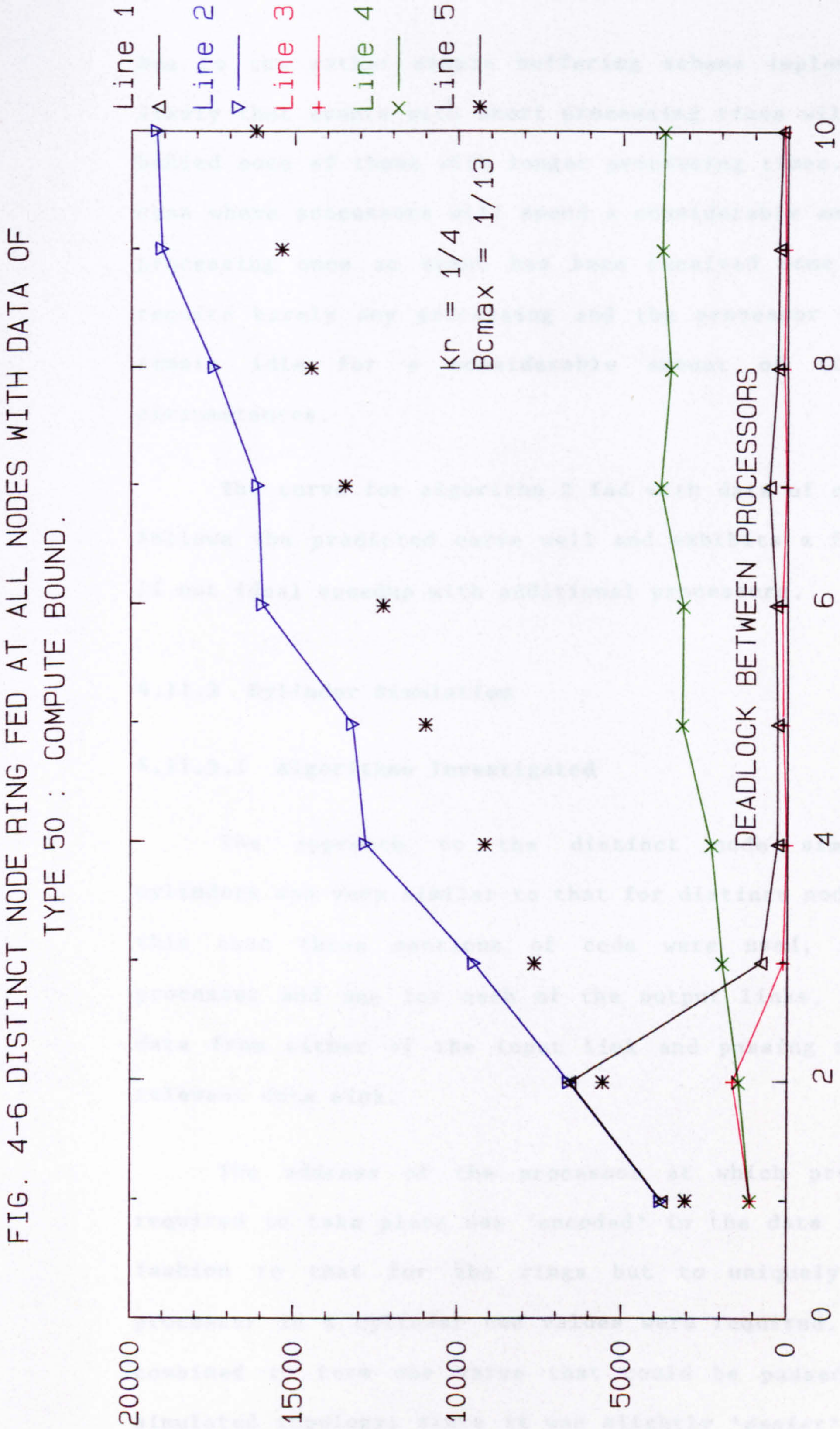


FIG. 4-6 DISTINCT NODE RING FED AT ALL NODES WITH DATA TYPE 50 : COMPUTE BOUND.

Due to the rather simple buffering scheme implemented it is likely that events with short processing times will be held up behind some of those with longer processing times. Unlike the case where processors will spend a considerable amount of time processing once an event has been received some events will require barely any processing and the processor is likely to remain idle for a considerable amount of time in such circumstances.

The curve for algorithm 2 fed with data of constant type follows the predicted curve well and exhibits a fairly linear if not ideal speedup with additional processors.

4.11.3 Cylinder Simulation

4.11.3.1 Algorithms Investigated

The approach to the distinct node simulations of cylinders was very similar to that for distinct node rings. In this case three sections of code were used, one for the processor and one for each of the output links, these taking data from either of the input link and passing it on to the relevant data sink.

The address of the processor at which processing was required to take place was 'encoded' in the data in a similar fashion to that for the rings but to uniquely identify a processor in a cylinder two values were required. These were combined to form one value that could be passed around the simulated topology; since it was slightly 'easier' to generate the 'encoded' value than to perform the reverse process and

generate the values of R and L from the encoded value the test for a match of the processor identity and the data was performed by generating the 'encoded' identity of the processor and comparing this with the data.

Only one algorithm was tested fully in this configuration, the algorithm giving priority to taking new data into the system in preference to dealing with data already within the ring structure was found to produce apparent deadlock situations and could not be used therefore in the cylindrical arrangement.

```
begin (* COMMS1 *)
if processor_idle then
  if new_data then
    if new_data_right then
      take_new_data
    else
      if ring_data then
        if ring_data_right then
          take_ring_data
        else
          (* NULL *)
      else
        (* NULL *)
  else
    if ring_data then
      if ring_data_right then
        take_ring_data
      else
        (* NULL *)
    else
      (* NULL *)
else
  (* NULL *);
if ring_ready then
  if new_data then
    if level_right_data and
       not column_right_data then
      new_data_on
    else
      if ring_data then
        if level_right_ring and
           not column_right_ring then
          ring_data_on
        else
          (* NULL *)
      else
        (* NULL *)
  else
    if ring_data then
      if level_right_ring and
         not column_right_ring then
        ring_data_on
      else
        (* NULL *)
    else
      (* NULL *)
else
  (* NULL *);
if down_ready then
  if new_data then
    if not level_right_data then
      new_data_down
    else
      if ring_data then
        if not level_right_ring then
          ring_data_down
        else
          (* NULL *)
```

```
begin (* COMMS2 *)
if processor_idle then
  if ring_data then
    if ring_data_right then
      take_ring_data
    else
      if new_data then
        if new_data_right then
          take_new_data
        else
          (* NULL *)
      else
        (* NULL *)
  else
    if new_data then
      if new_data_right then
        take_new_data
      else
        (* NULL *)
    else
      (* NULL *)
else
  (* NULL *);
if ring_ready then
  if ring_data then
    if level_right_ring and
       not column_right_ring then
      ring_data_on
    else
      if new_data then
        if level_right_data and
           not column_right_data then
          new_data_on
        else
          (* NULL *)
      else
        (* NULL *)
  else
    if new_data then
      if level_right_data and
         not column_right_data then
        new_data_on
      else
        (* NULL *)
    else
      (* NULL *)
else
  (* NULL *);
if down_ready then
  if ring_data then
    if not level_right_ring then
      ring_data_down
    else
      if new_data then
        if not level_right_data then
          new_data_down
        else
          (* NULL *)
```

```
      else
        (* NULL *)
  else
    if ring_data then
      if not level_right_ring then
        ring_data_down
      else
        (* NULL *)
    else
      (* NULL *)
else
  (* NULL *)
end;  (* COMMS1 *)
```

```
      else
        (* NULL *)
  else
    if new_data then
      if not level_right_data then
        new_data_down
      else
        (* NULL *)
    else
      (* NULL *)
else
  (* NULL *)
end;  (* COMMS2 *)
```

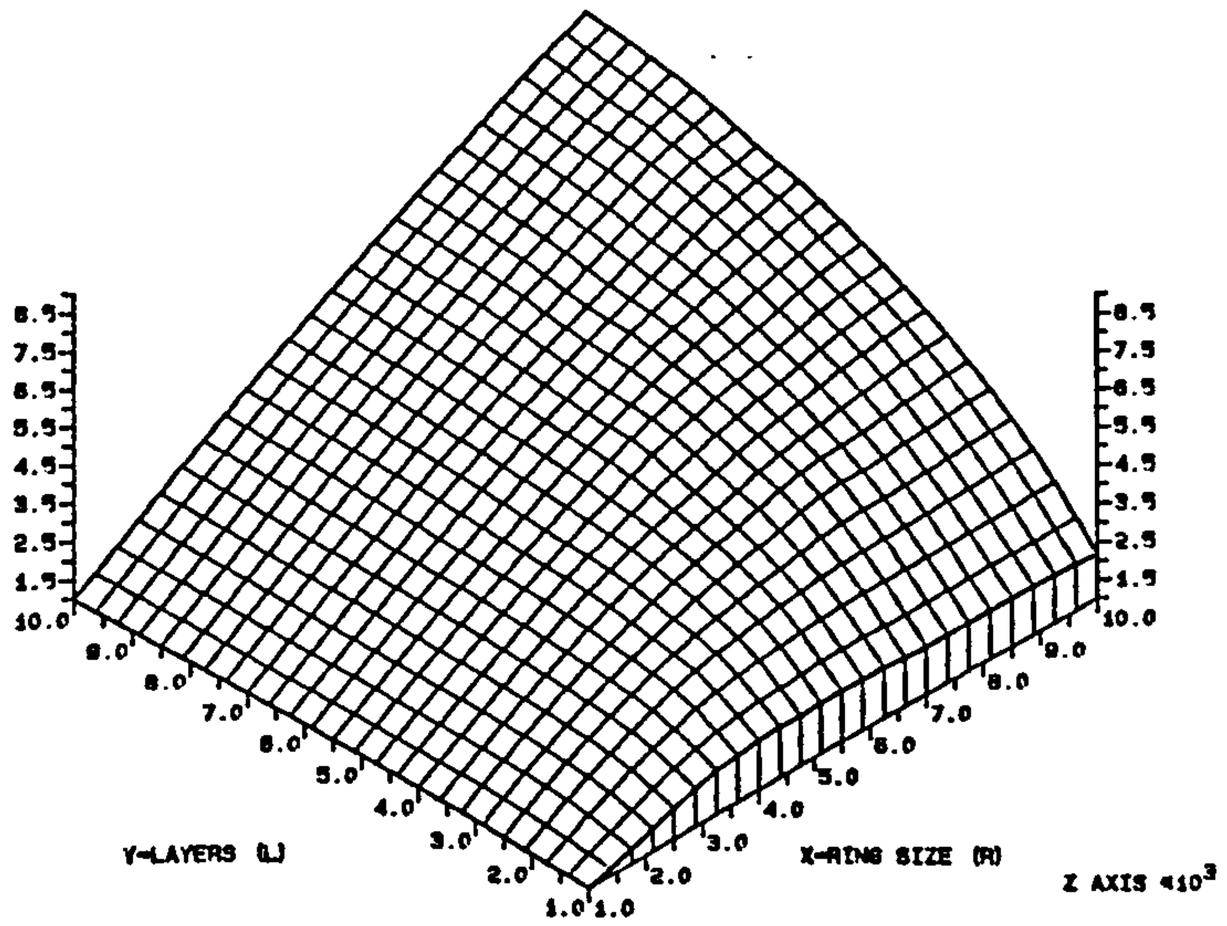
4.11.3.2 Performance of the Algorithms

4.11.3.2.1 Communication Bound

A few sample runs were tried using a development of the first communication algorithm and as expected processing very quickly produced an apparent deadlock condition. This condition was assumed to take the same form as the deadlock of the ring structure and this scheme was not investigated further, the following results are for a system using the second algorithm presented.

In the case of a system fed with data of type 1 the system will be entirely communication bound, this is directly comparable to the communication bound ring structure (section 4.11.2.2.1). By similar reasoning to the case of the ring structure values of K_r=1/4 and β_{cmax} =1 were used to model this situation. The results of the simulated total processing are shown in fig 4.7.

The curve along the line L=1 is, as expected, the same as that for the case of the ring with a maximum reached at approximately R=5 and an asymptotic value of 2 events/iteration as R becomes large.



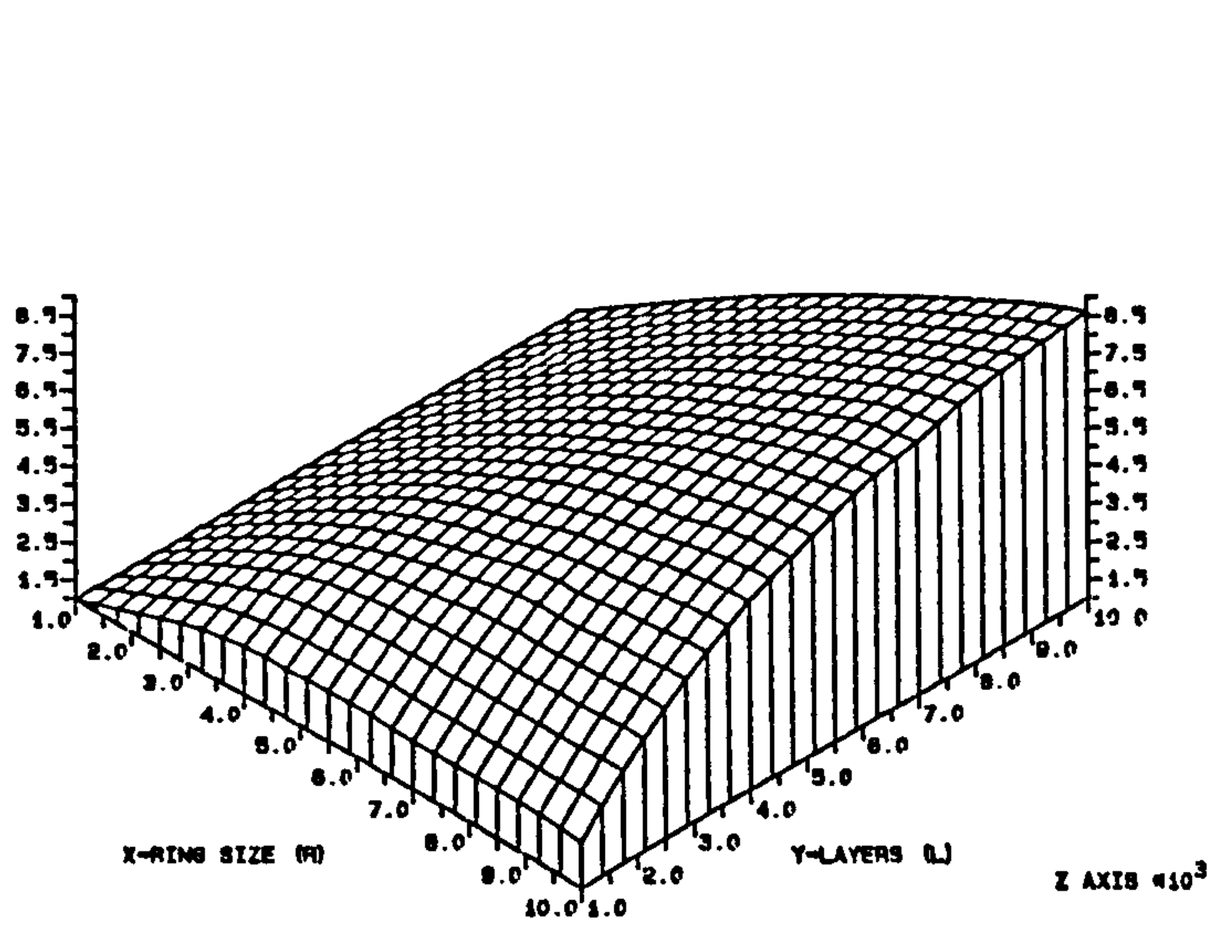
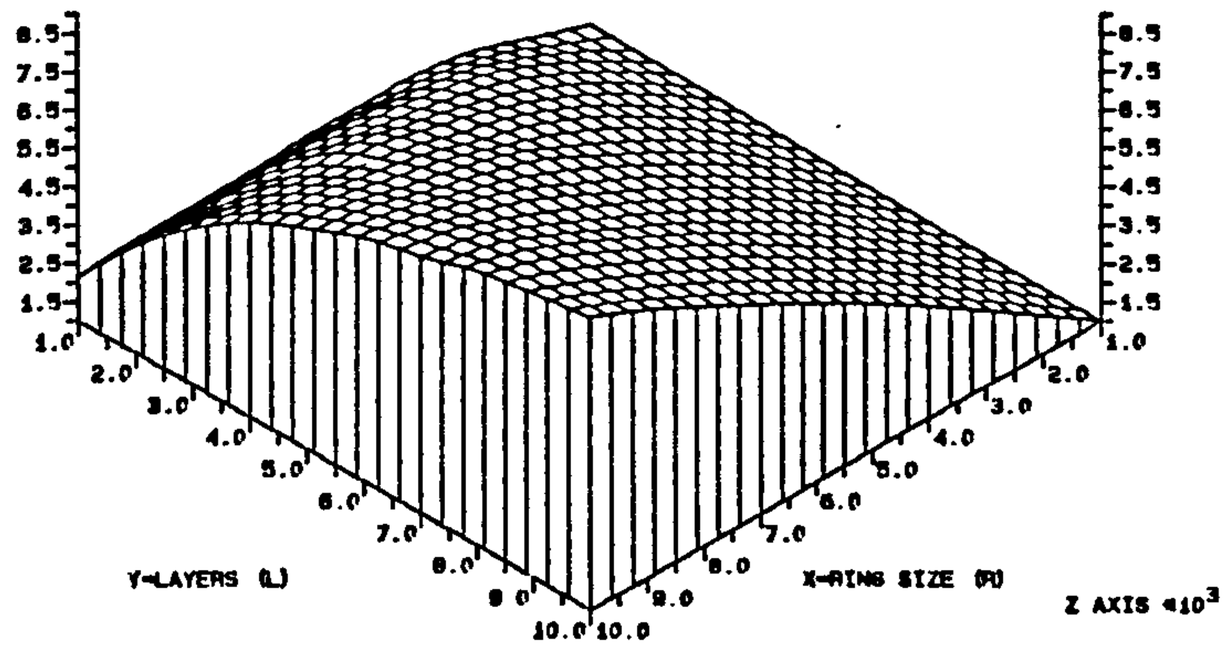
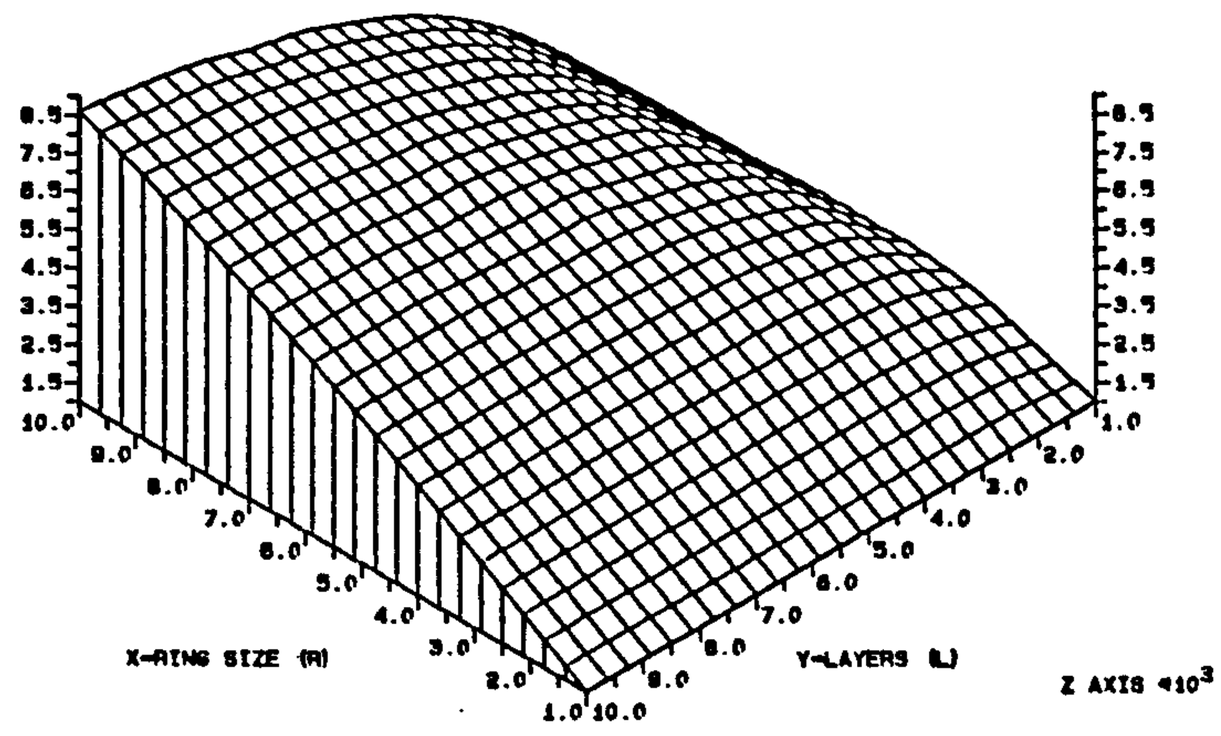


FIG. 4-7 DISTINCT NODE COMMS2 FED AT ALL NODES WITH DATA OF TYPE 1. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





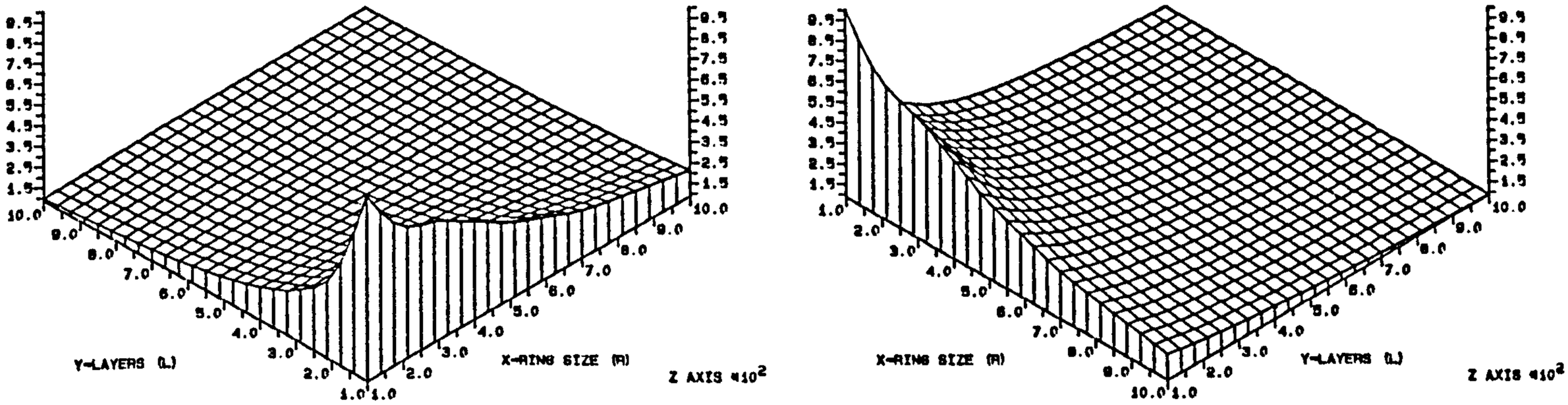
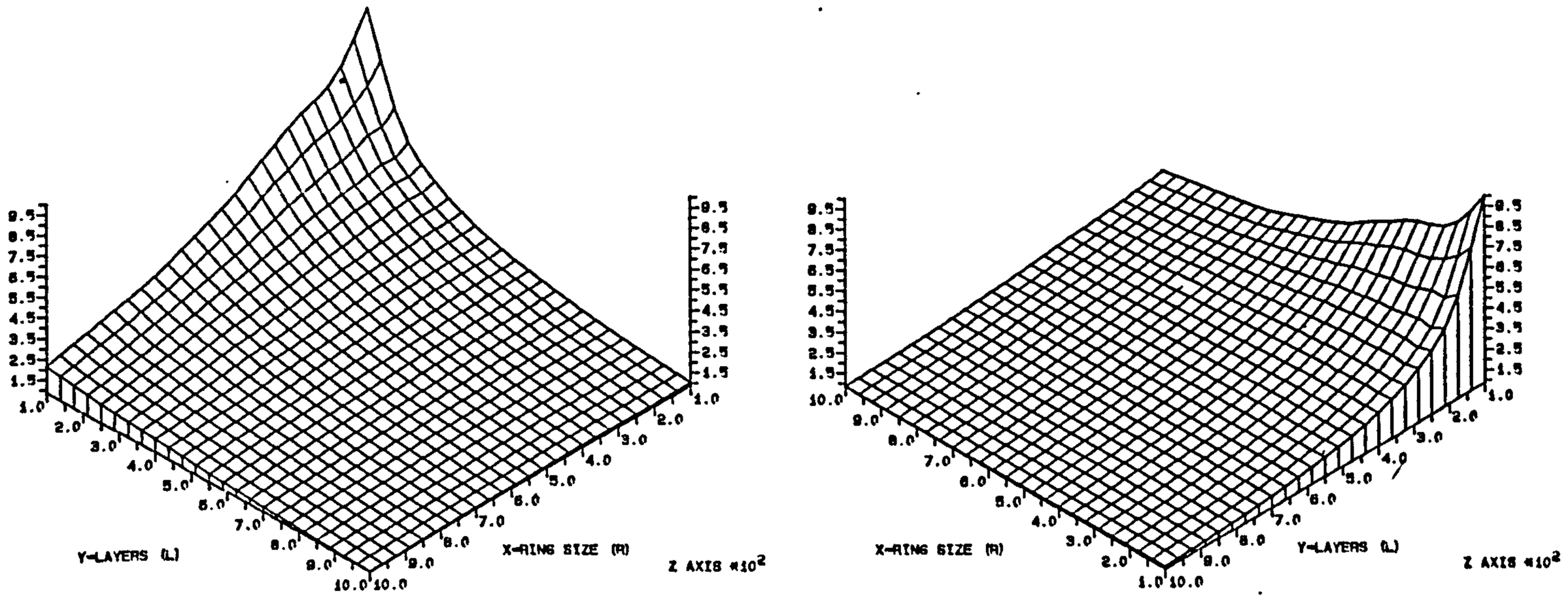


FIG. 4-8 DISTINCT NODE COMMS2 FED AT ALL NODES WITH DATA OF TYPE 1. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



For values of R=1 and increasing L there is no improvement in throughput as there is only one source of data. There is a broad ridge in the surface, the top of which tends to follow the R=2L + 1 optimum shape predicted by the model of section 3.4.2.6.2.

Fig 4.8 shows the weighted total processing ÷ the number of nodes in the system, this gives the mean processing applied towards event processing per node and does not have any component from the processing required for routing data. From this surface the most 'efficient' arrangement would appear to be a single processor however this is also the arrangement with the lowest throughput. The peak value of this surface is 1000 events/iteration/node and the surface decays down to a value of less than 100 which corresponds to less than 0.1 events/iteration per processor. Though this may sound poor this is the best that could be achieved since with β_n=1 and L=10 each processor would receive 1/10 events/iteration to process.

4.11.3.3 Larger Data Types

Figs 4.9 - fig 20 show the simulation results of distinct node cylinders with varying input data and the corresponding plots of processing per node. The surfaces of weighted total processing show all of the main features of the theoretically derived graphs of weighted total processing, figs 3.9 - 3.11.

There is a slight asymmetry in the results from the simulations when compared to those of the theoretical model in that the weighted total processing increases more linearly with

R, at large L than predicted by the model and the increase with L at large R shows a more distinct rounding. This can be attributed to a combination of the number of events required to fill the cylinder on startup increasing with L (see section 4.9) and some degree of bias towards horizontal rather than vertical communication being present in the communication algorithm.

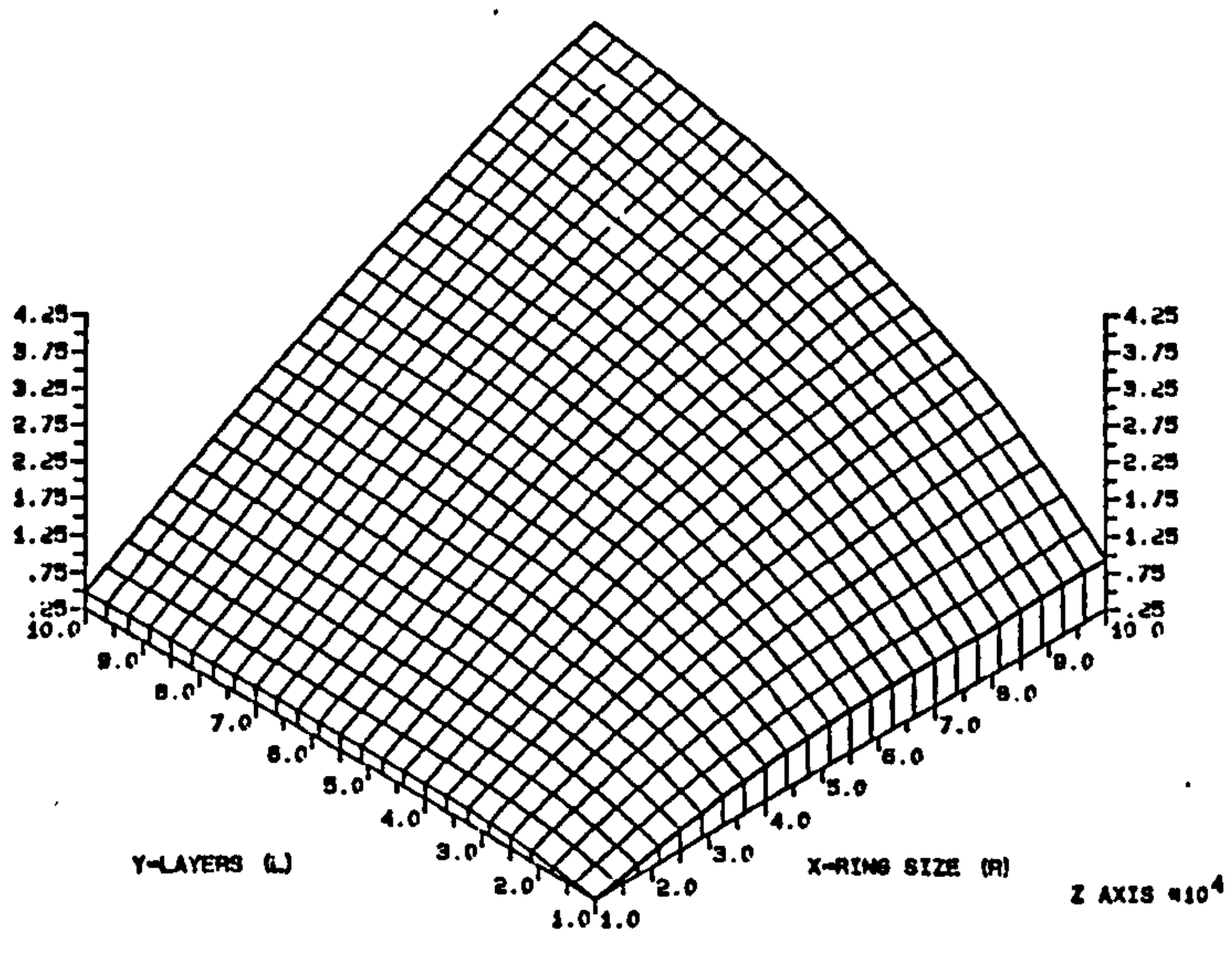
Comparing the peak values of the surfaces (ie at the point R=L=10) with the corresponding values of the theoretical model. The value of total processing in events per unit time (iteration) in the flow model is given by

$$\text{Total Processing} = \frac{R \cdot L \cdot \beta_{cmax}}{1 + K_r \cdot \left[\frac{(R-1)}{2} + \frac{(L+1)}{2} \right]}$$

which for a value of K_r=1/4 gives

$$\text{Total Processing} = 28.57 \times \beta_{cmax} \text{ events/iteration}$$

For the cylinder fed with data of type 5, (giving $\beta_{cmax} = 1/\lceil 5/4 \rceil = 1/2$) this gives a peak value of 14.29 events/iteration which represents a value of 71430 units of total processing over 1000 iterations. However the number of events presented to the system cannot exceed 10 000 (R × Iterations) giving a maximum total processing of 50 000, clearly this is the main factor constraining the system throughput.



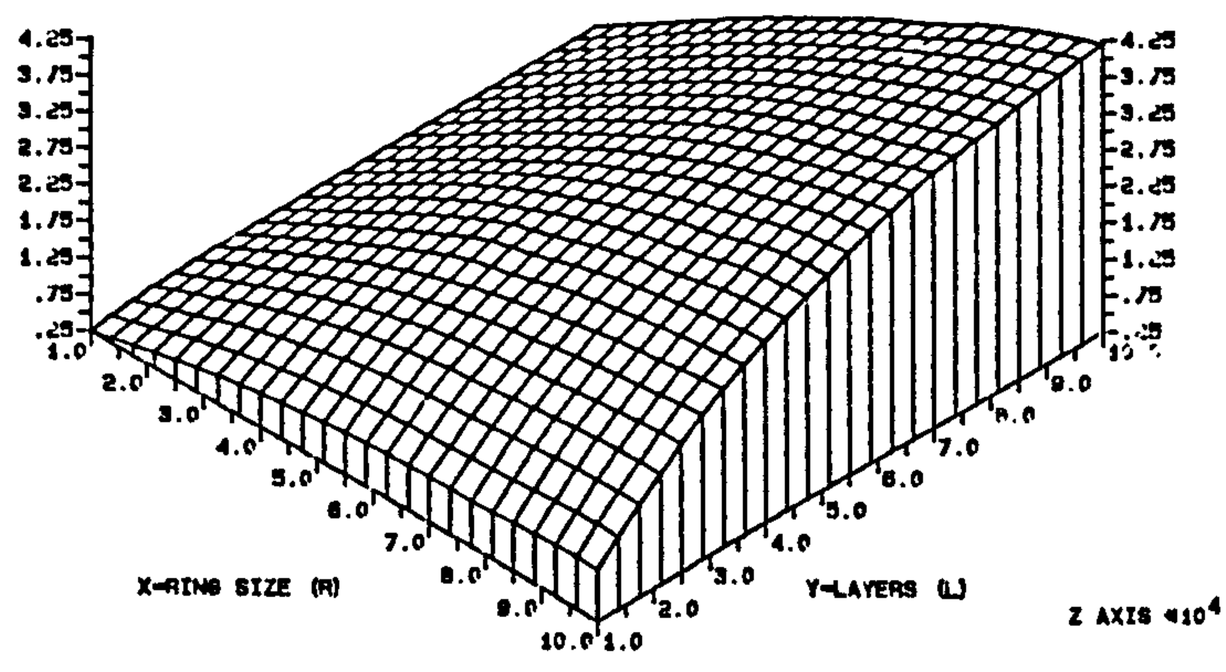
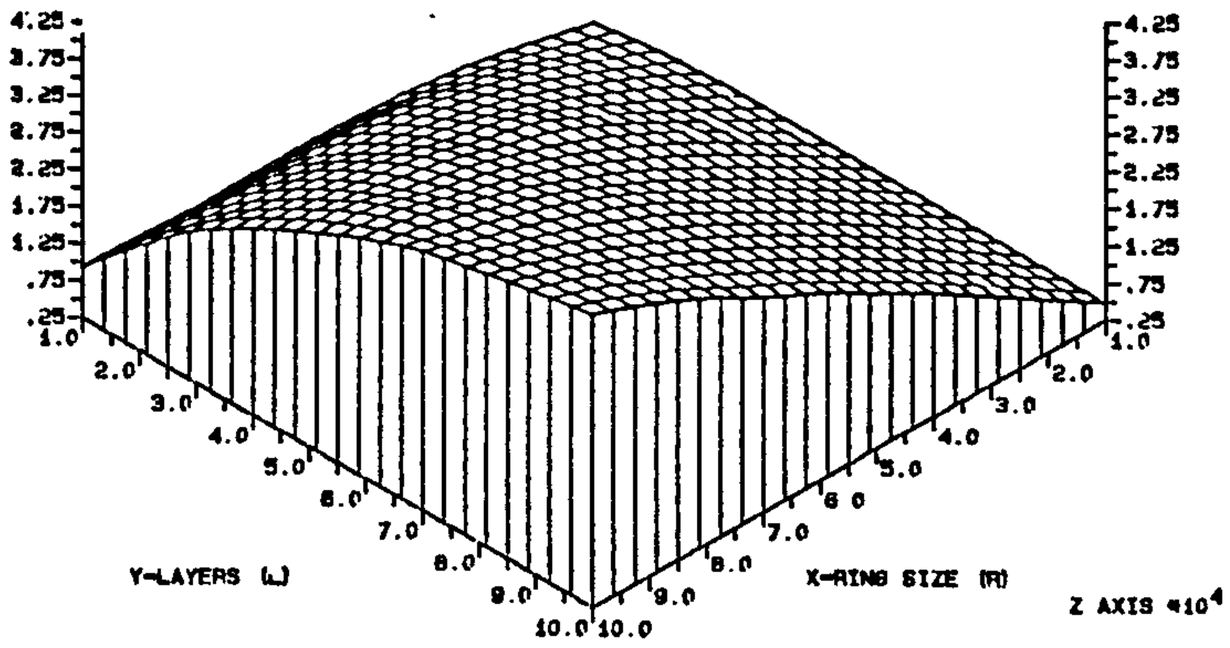
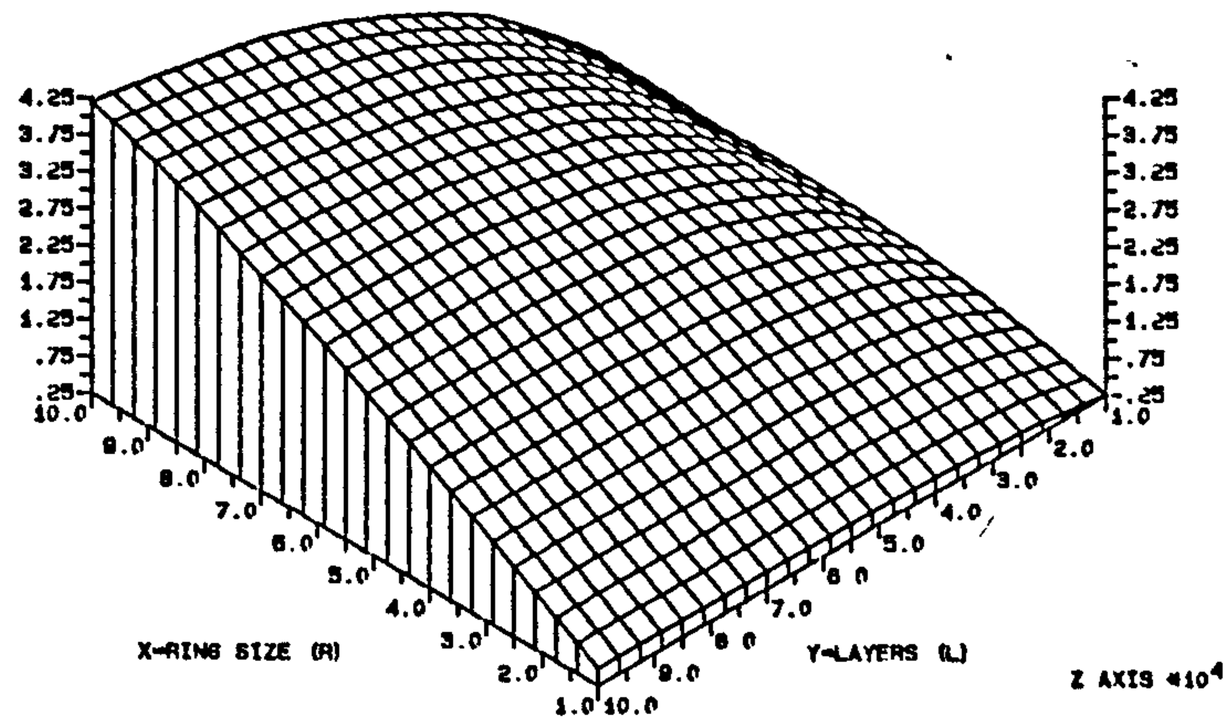
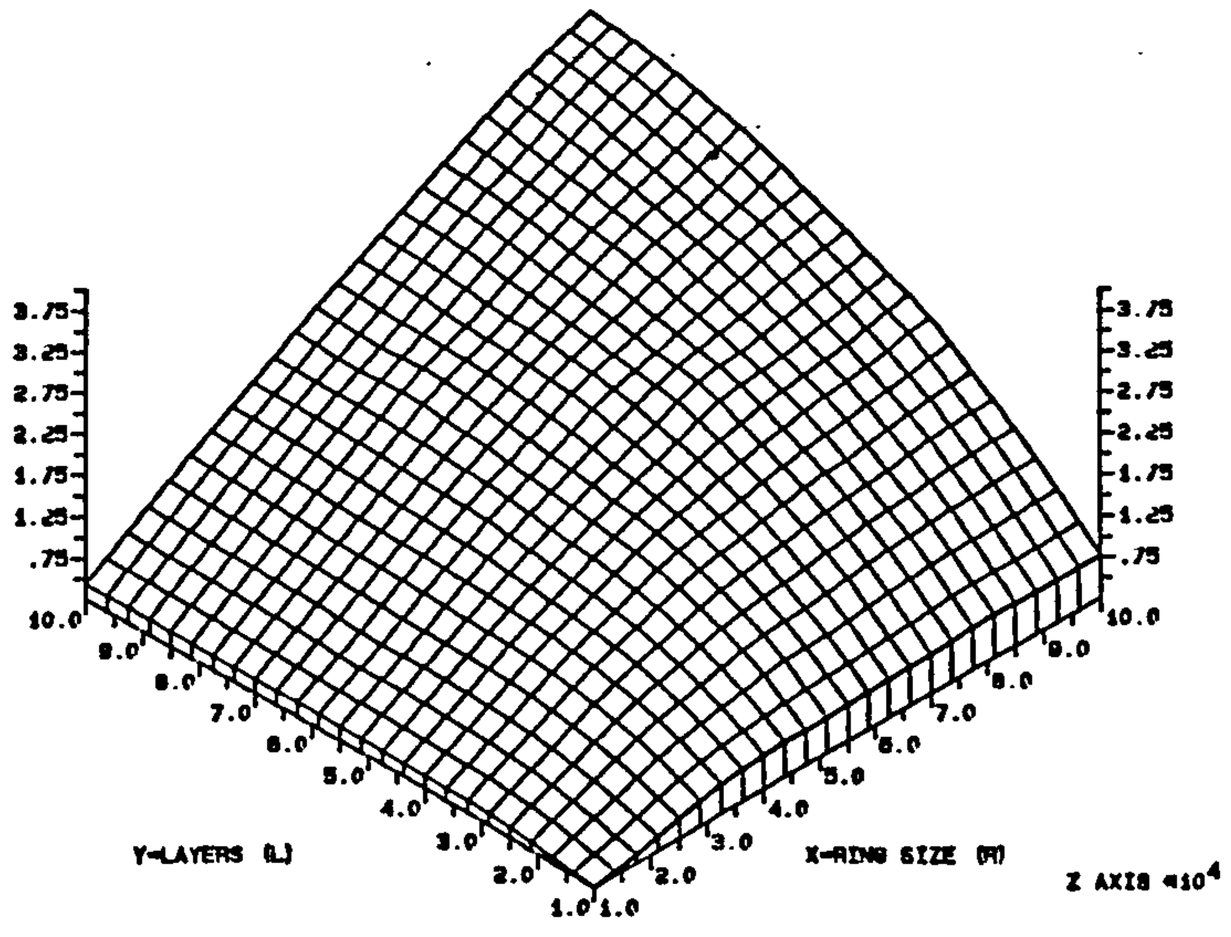


FIG. 4-9 DISTINCT NODE COMMS2 FED AT ALL NODES WITH DATA OF TYPE 5. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.







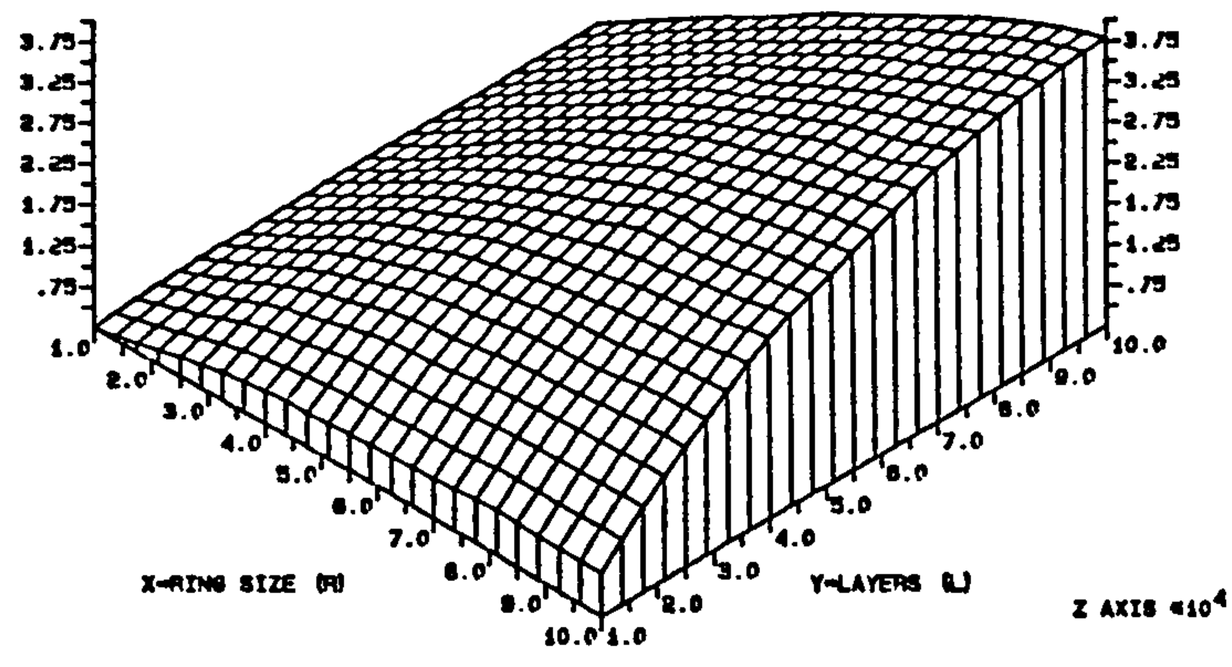
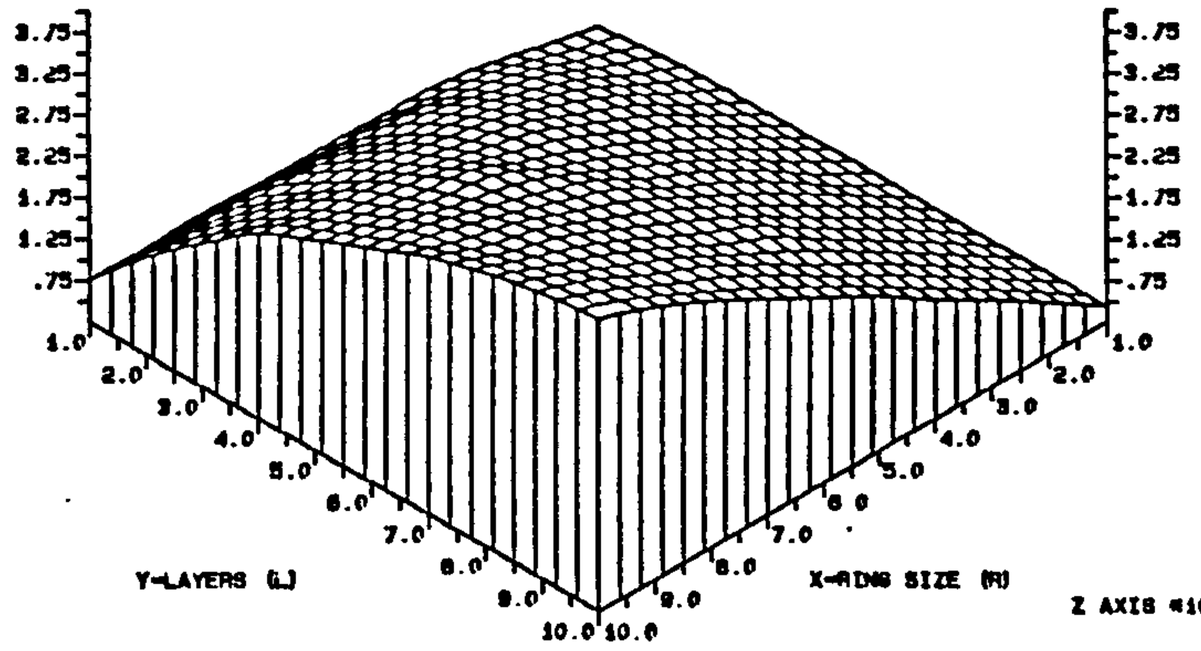
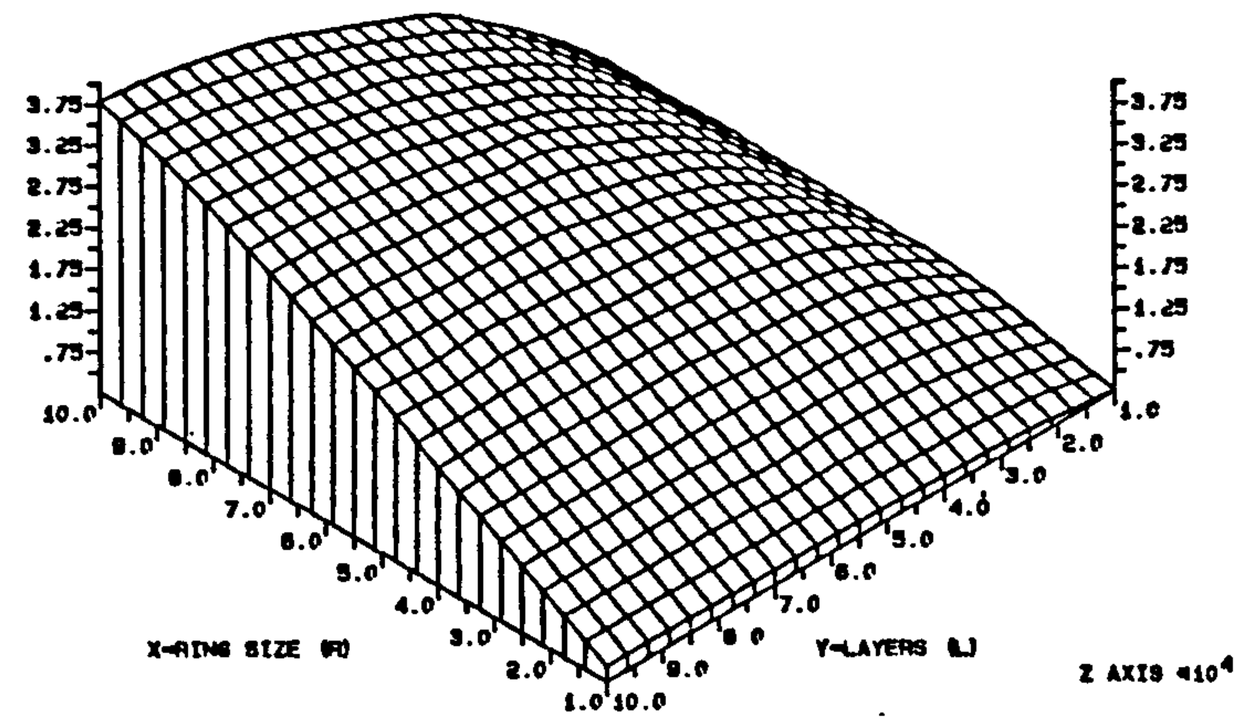
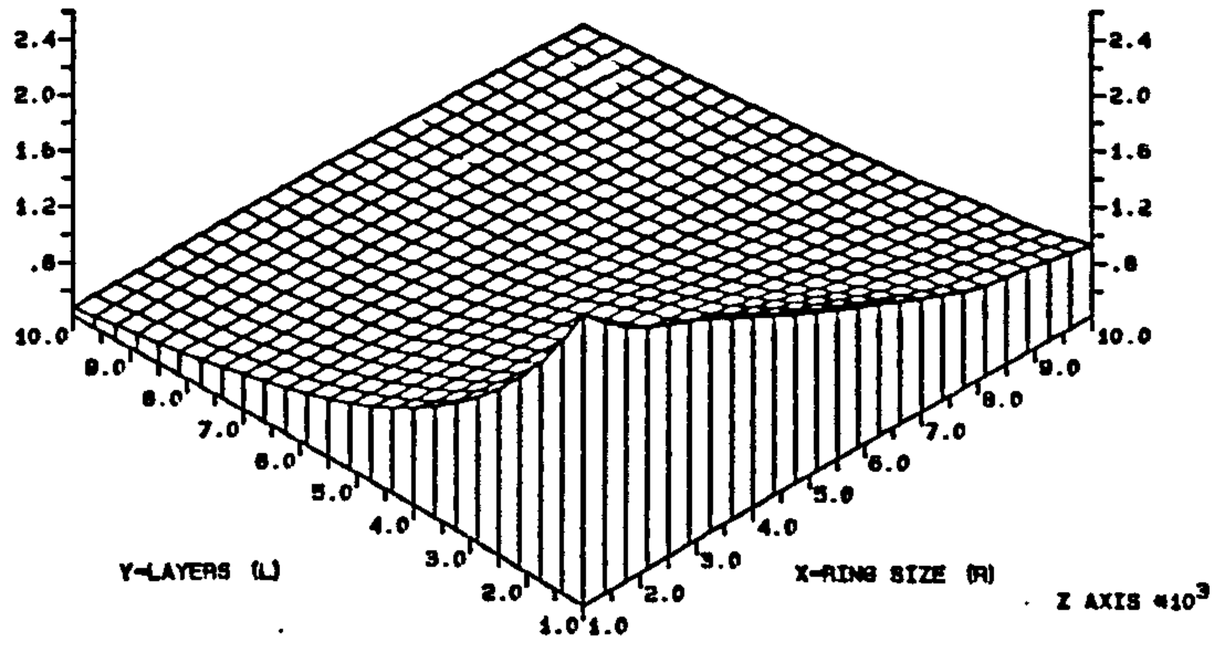


FIG. 4-10 DISTINCT NODE COMMS2 FED AT ALL NODES WITH DATA OF TYPE R10: Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.







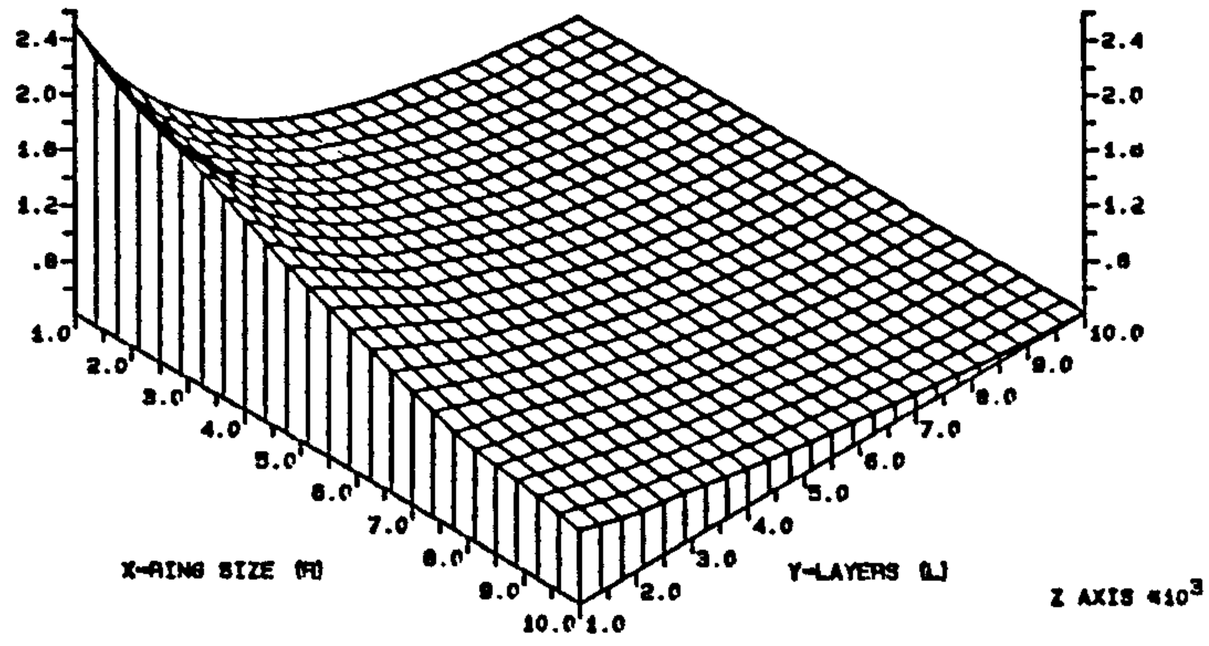
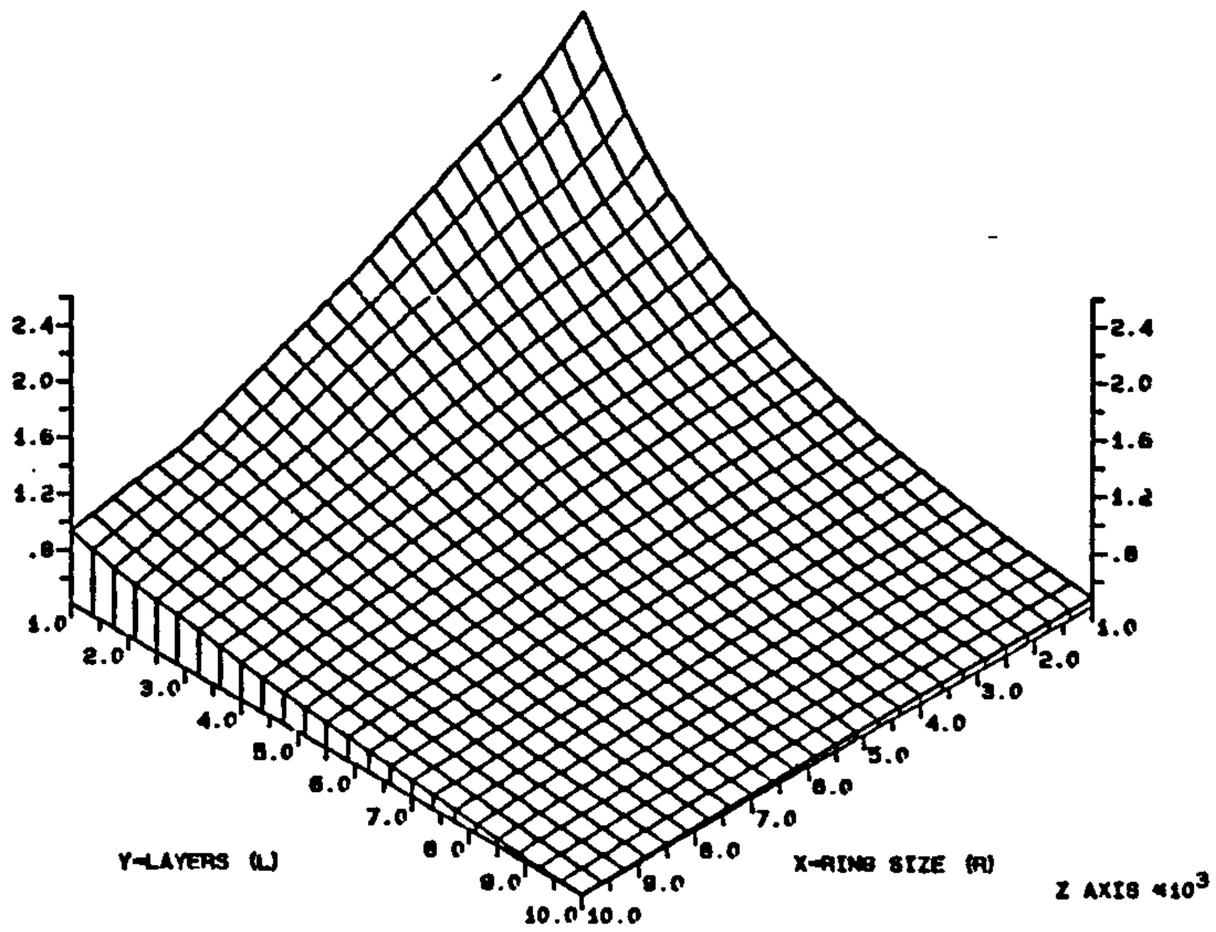
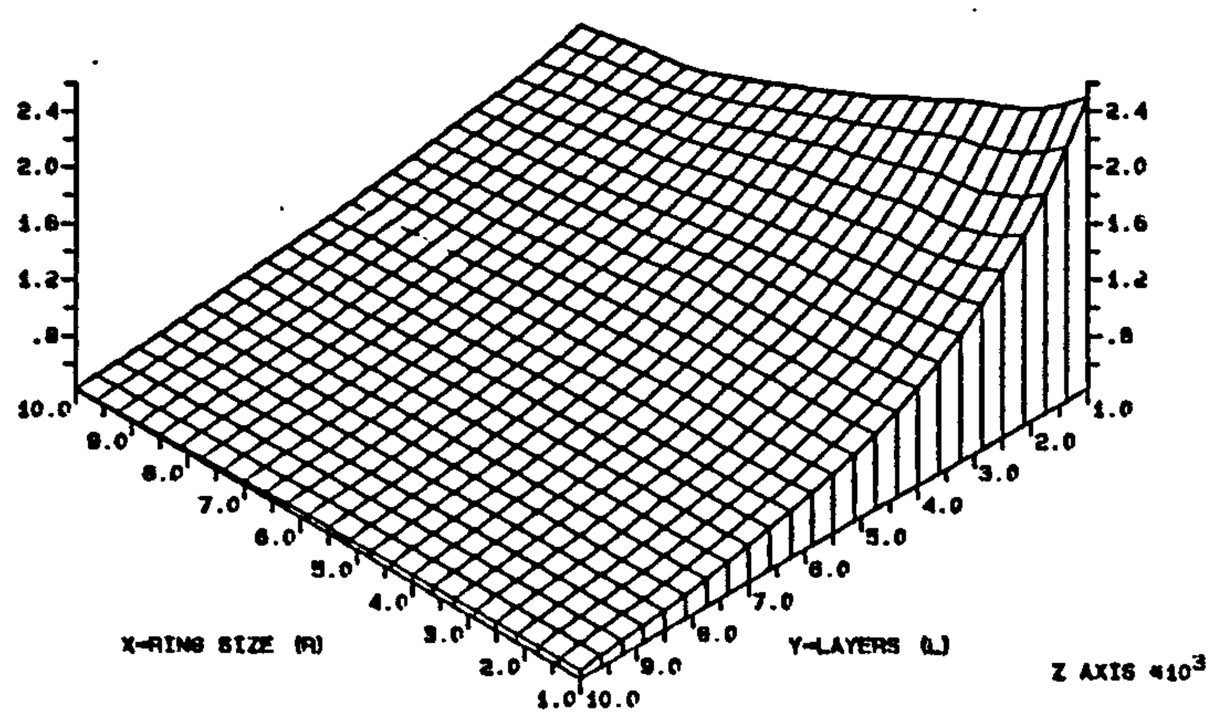


FIG. 4-11 DISTINCT NODE COMMS2 FED AT ALL NODES WITH DATA OF TYPE 5. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





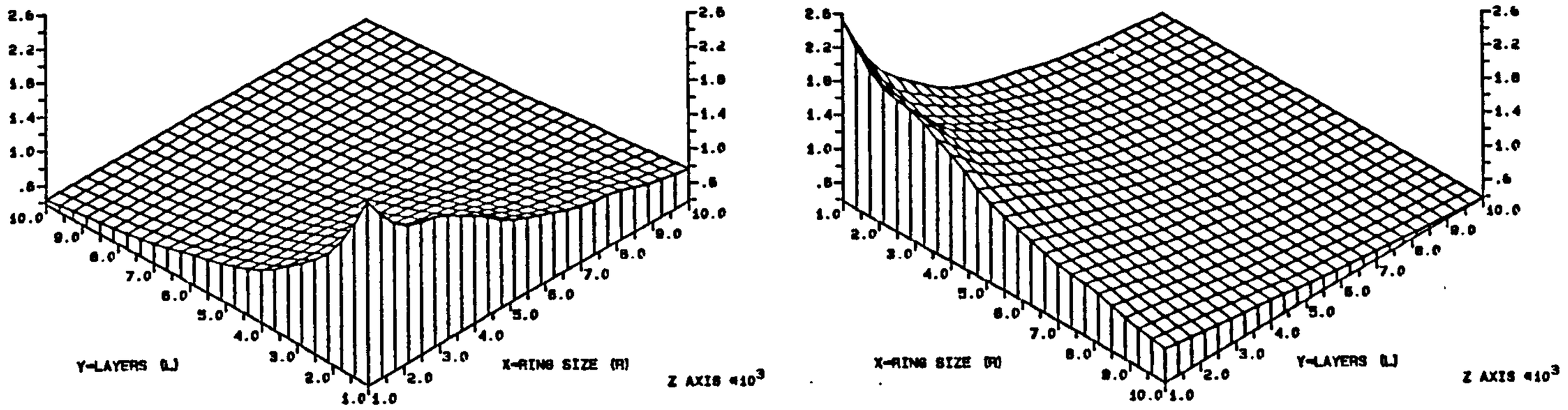
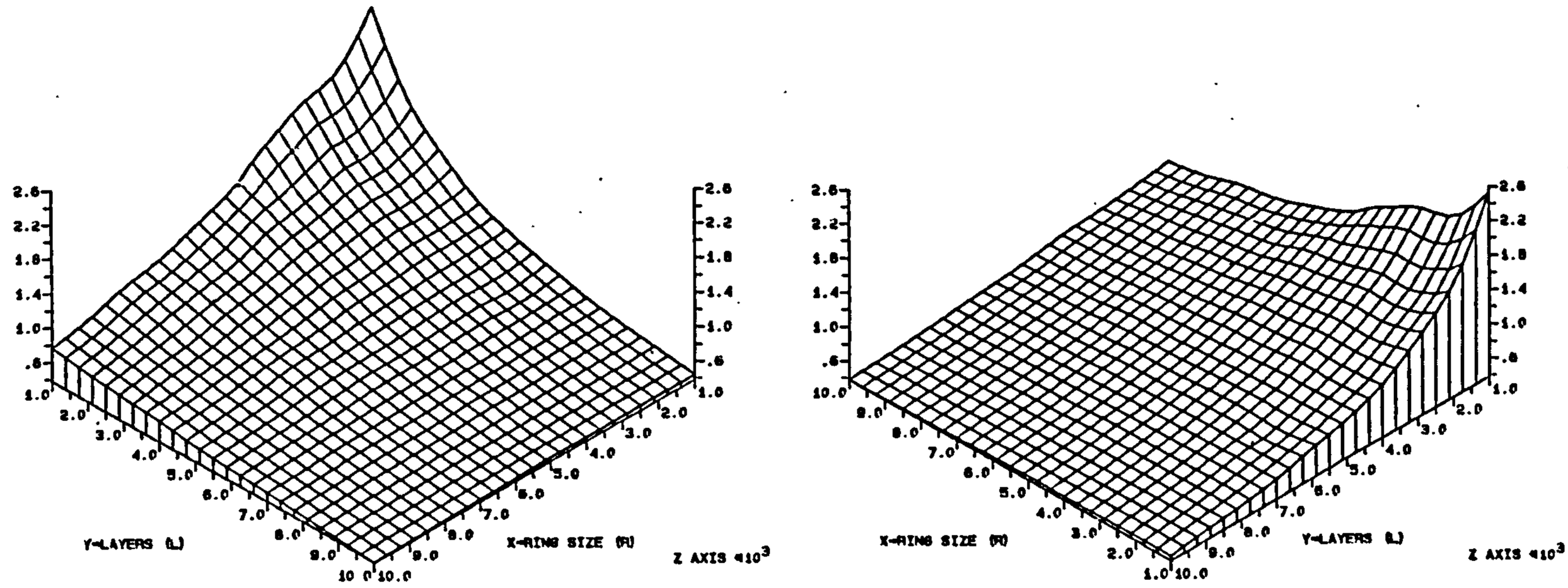


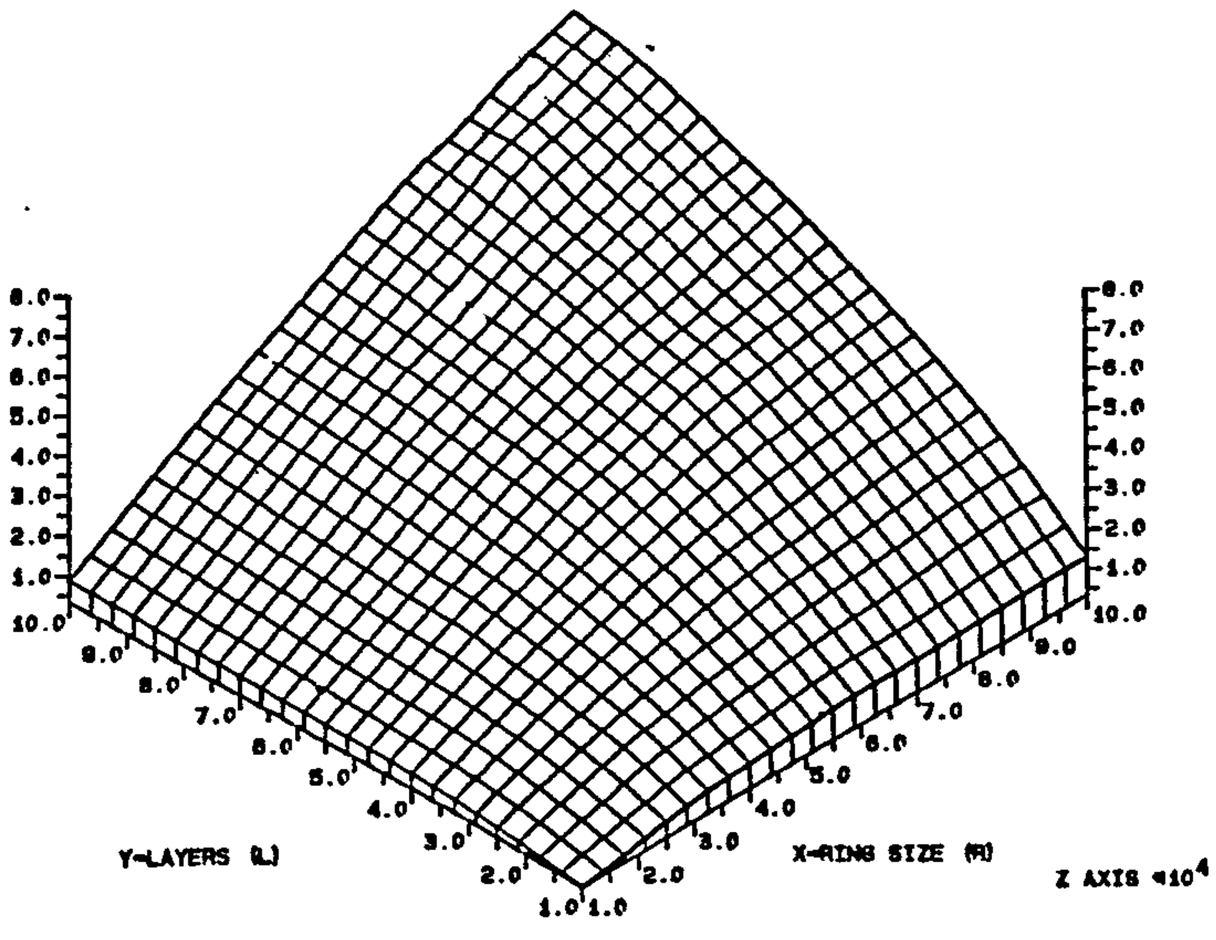
FIG. 4-12 DISTINCT NODE COMMS2 FED AT ALL NODES WITH DATA OF TYPE R10. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



This is still considerably greater than the 42 500 obtained by the simulation the results for which are shown in fig 4.9, obviously some effects other than included in the model have an influence on the processing. The simulation fed with randomly distributed data with a mean value of 5 produced a similar performance to the constant data type case, these results are shown in fig 4.10. The discrepancy between the calculated and simulated result is probably due to some processors remaining idle while the system initially fills with data.

Comparing the model/simulation discrepancy for simulations of data types requiring larger amount of processing shows an improvement in the model/simulation correlation for these data types.

Repeating the calculations for simulation performed for a system fed with data of type 10 gives a predicted throughput of 76 923 processing units over the simulation, this compares well with the 76 000 processing units achieved by the simulation (see fig 4.13). The simulation fed with randomly distributed data with mean value of 10 produced a similar performance to the constant data type case. This is shown in fig 4.14.



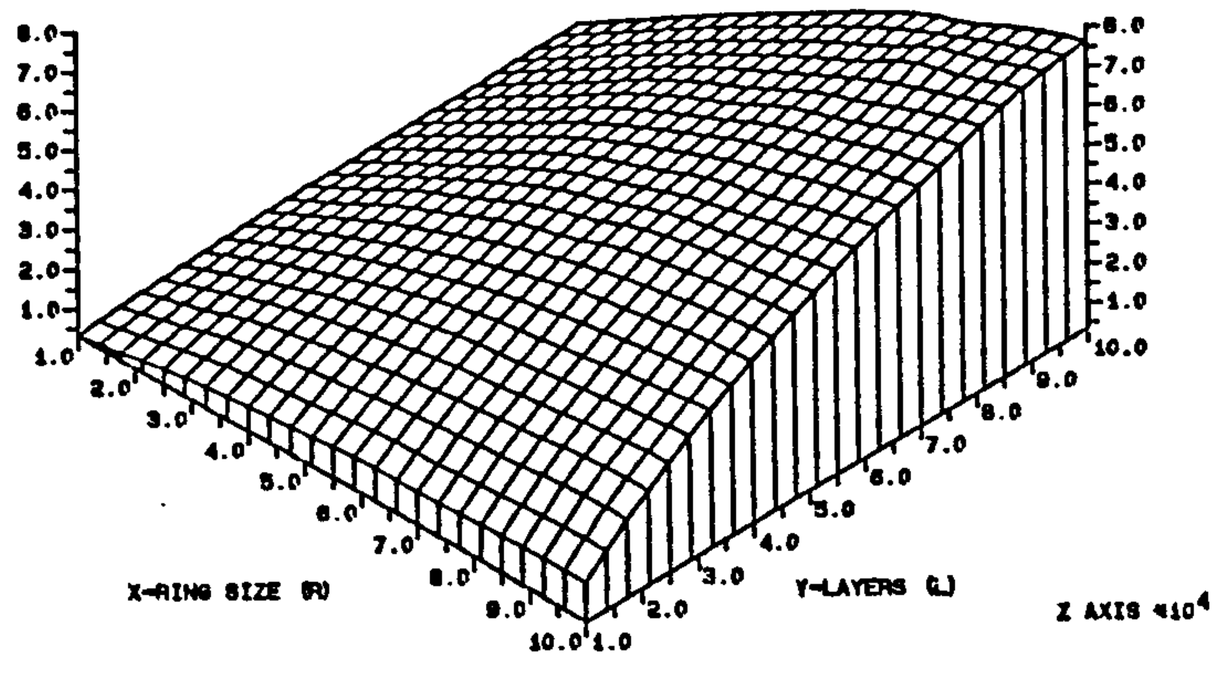
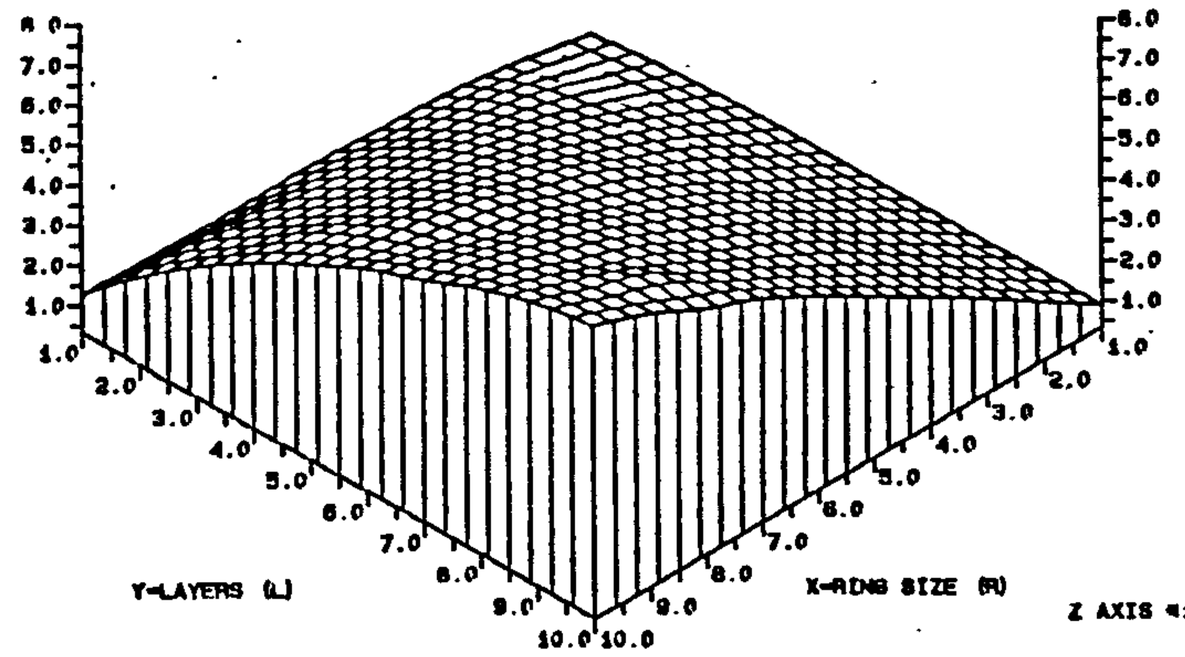
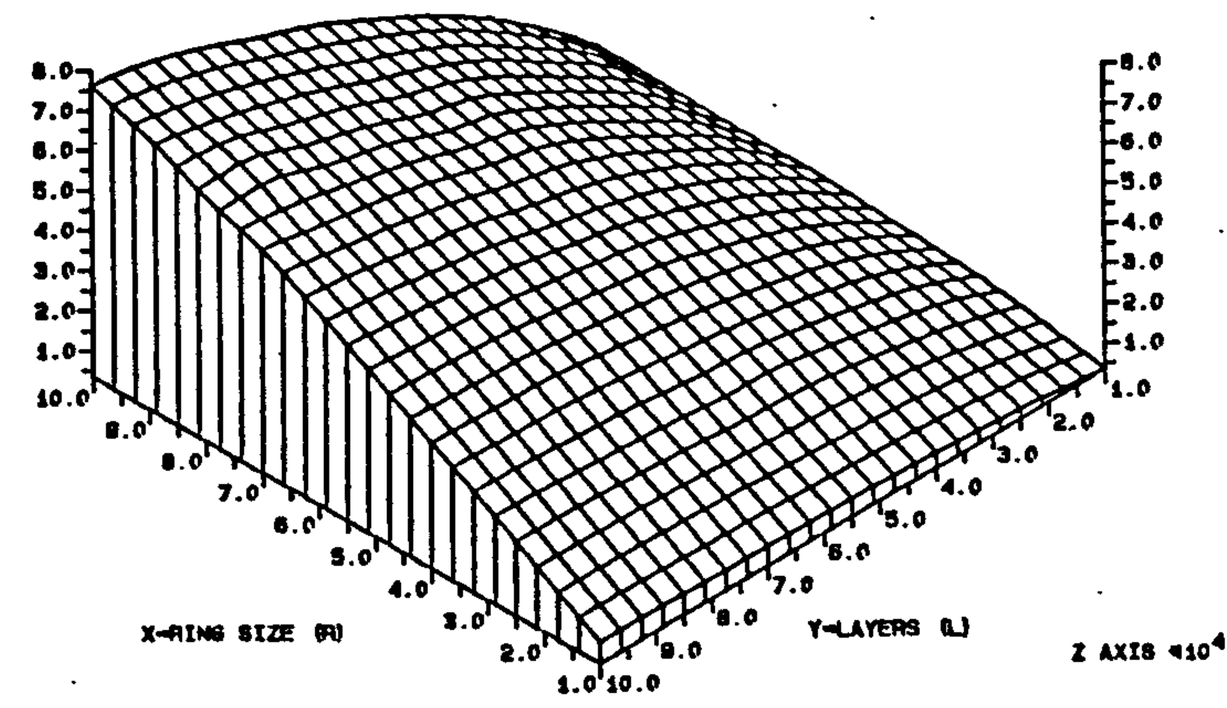
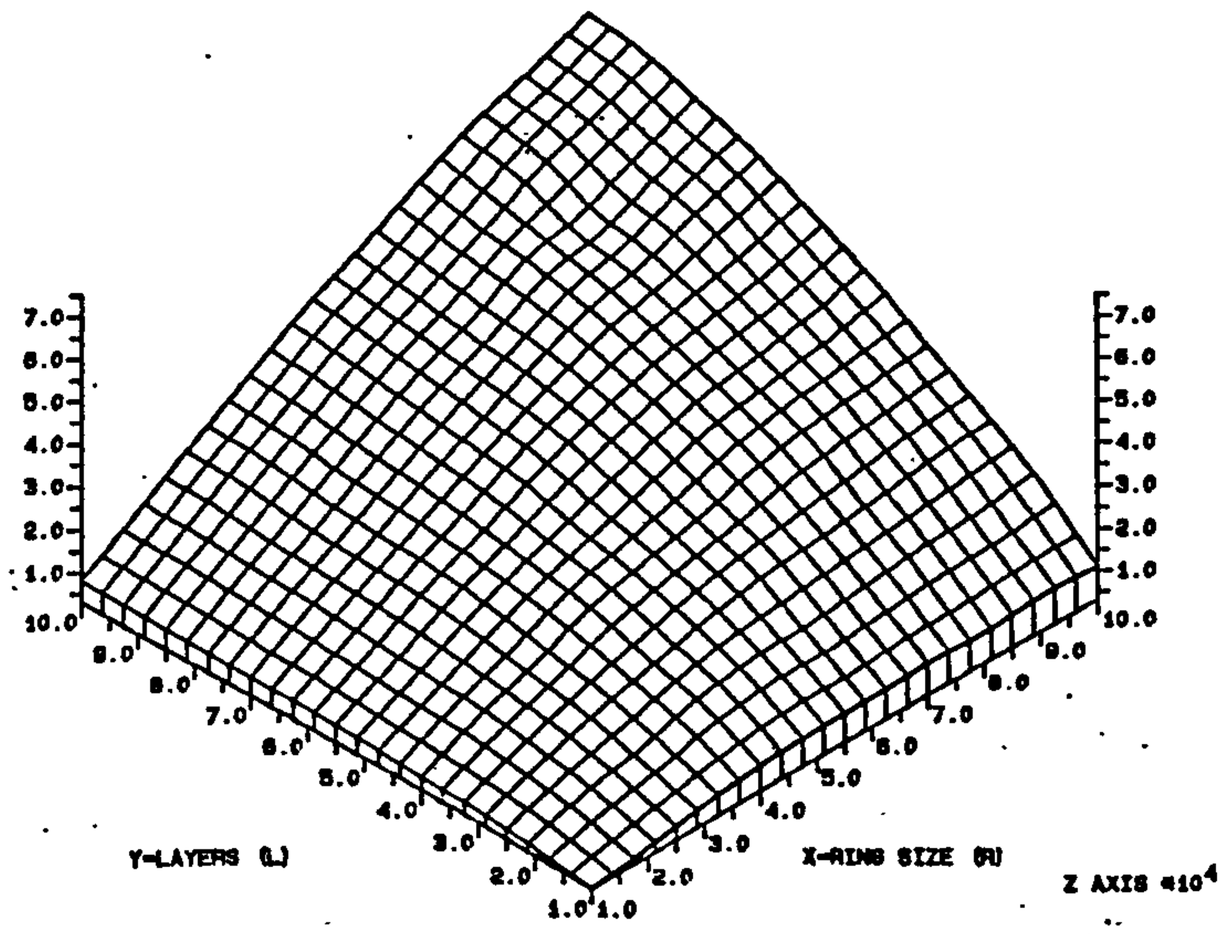


FIG. 4-13 DISTINCT NODE COMMS2 FED AT ALL NODES WITH DATA OF TYPE 10. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.







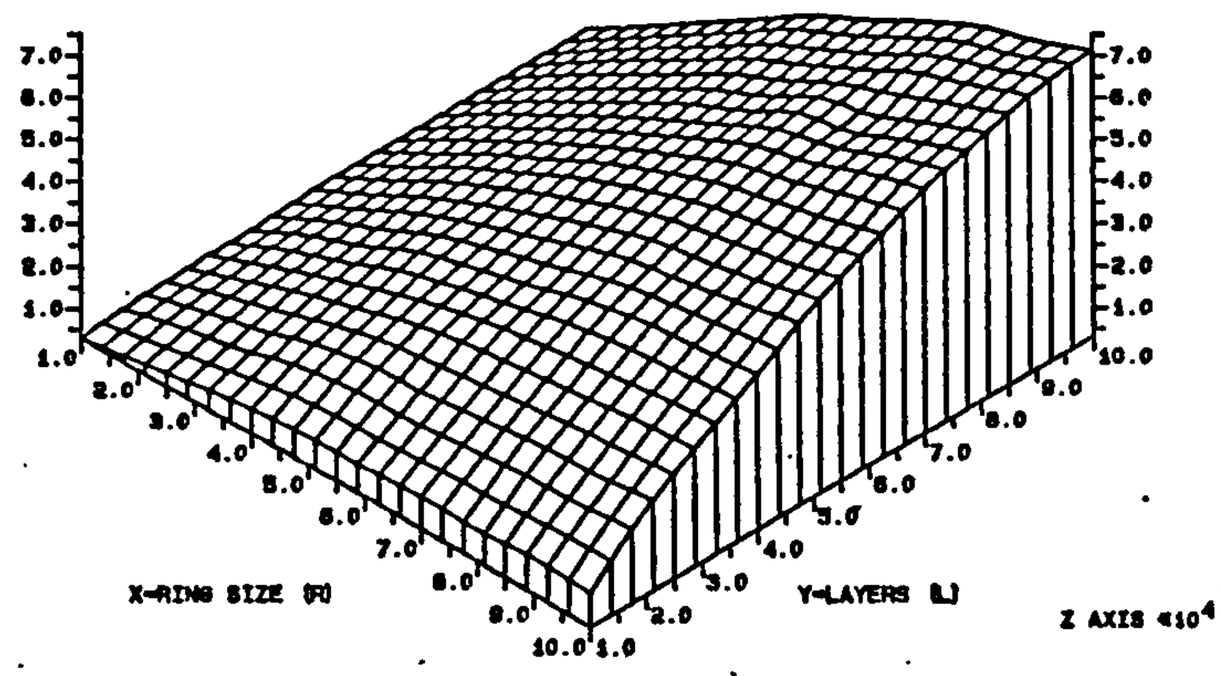
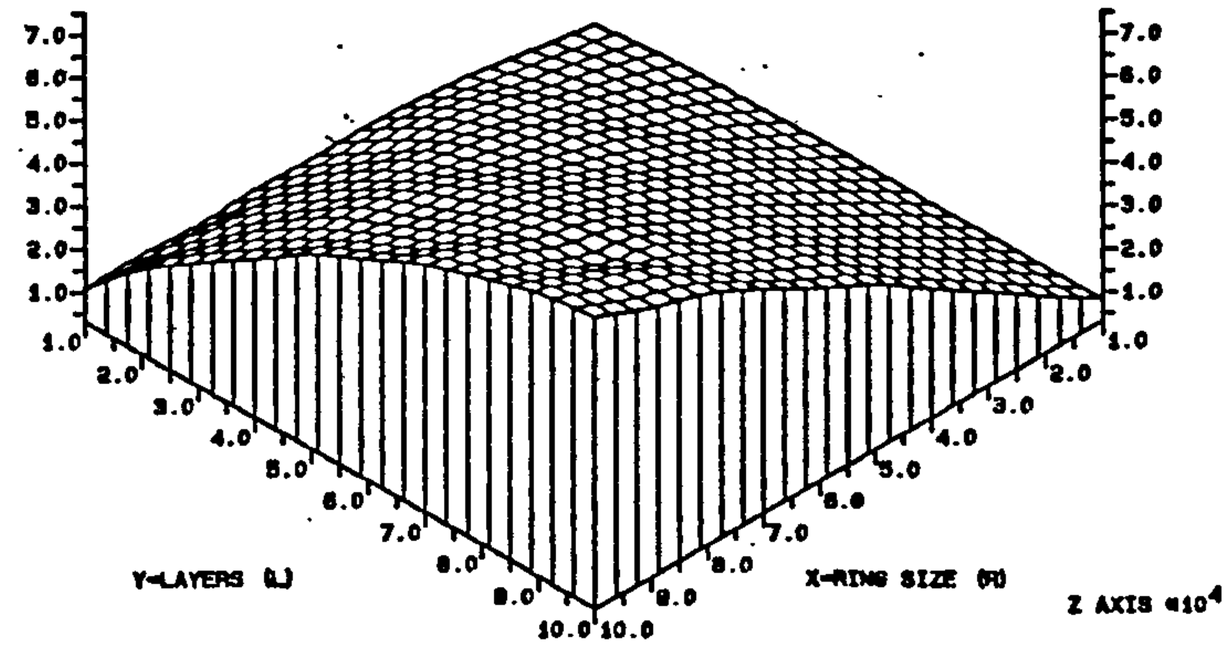
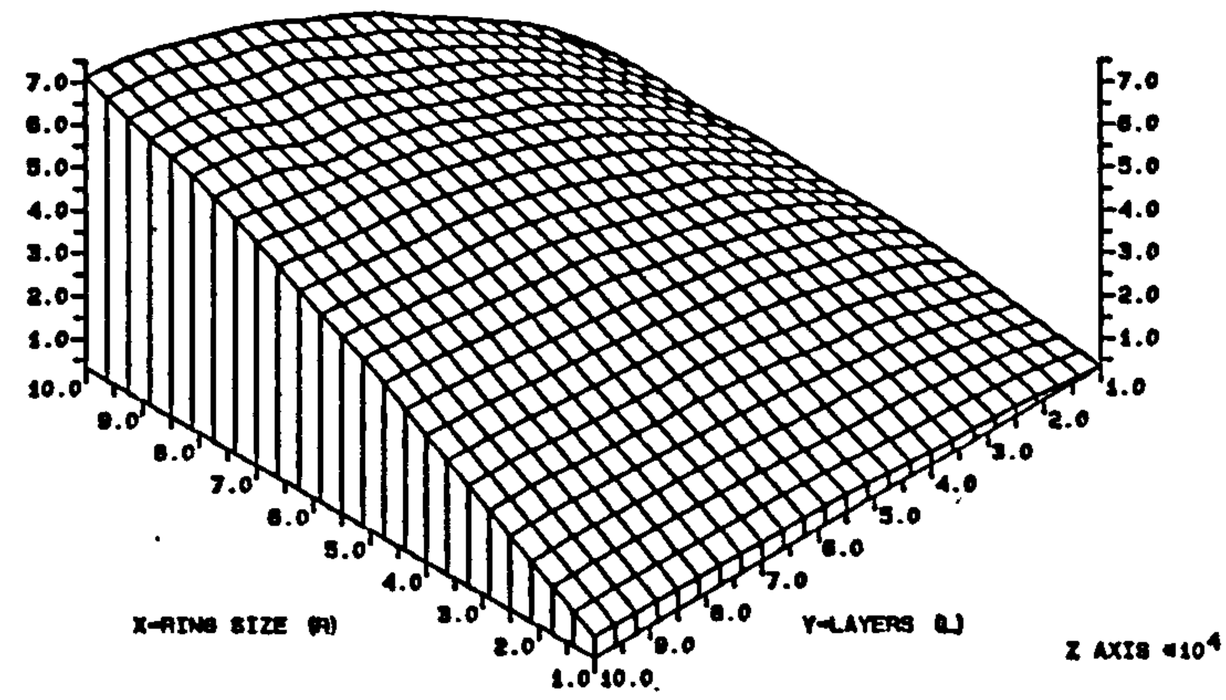


FIG. 4-14 DISTINCT NODE COMMS2 FED AT ALL NODES WITH DATA OF TYPE R20. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





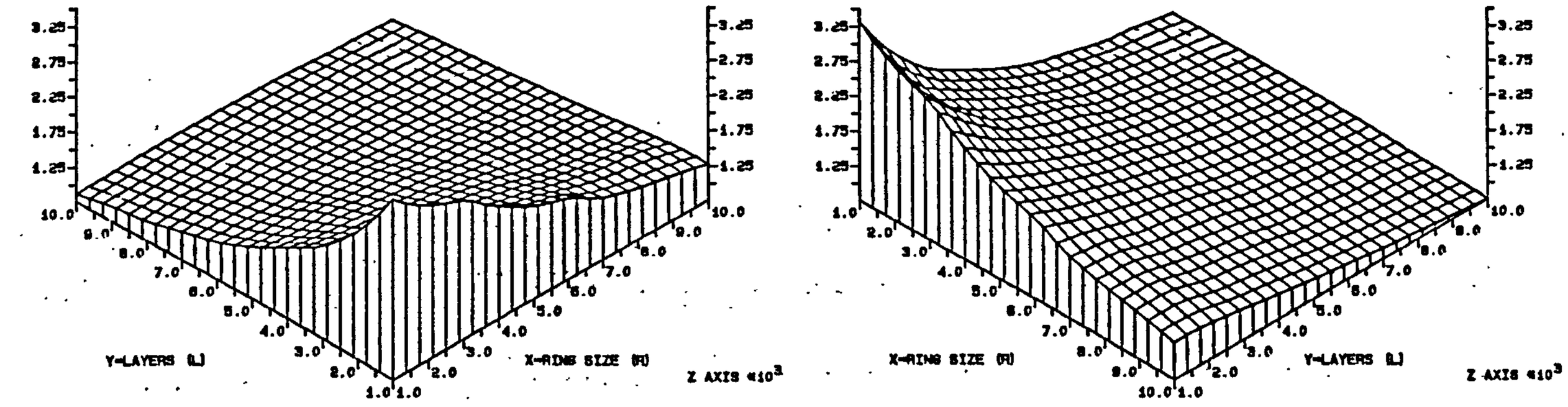
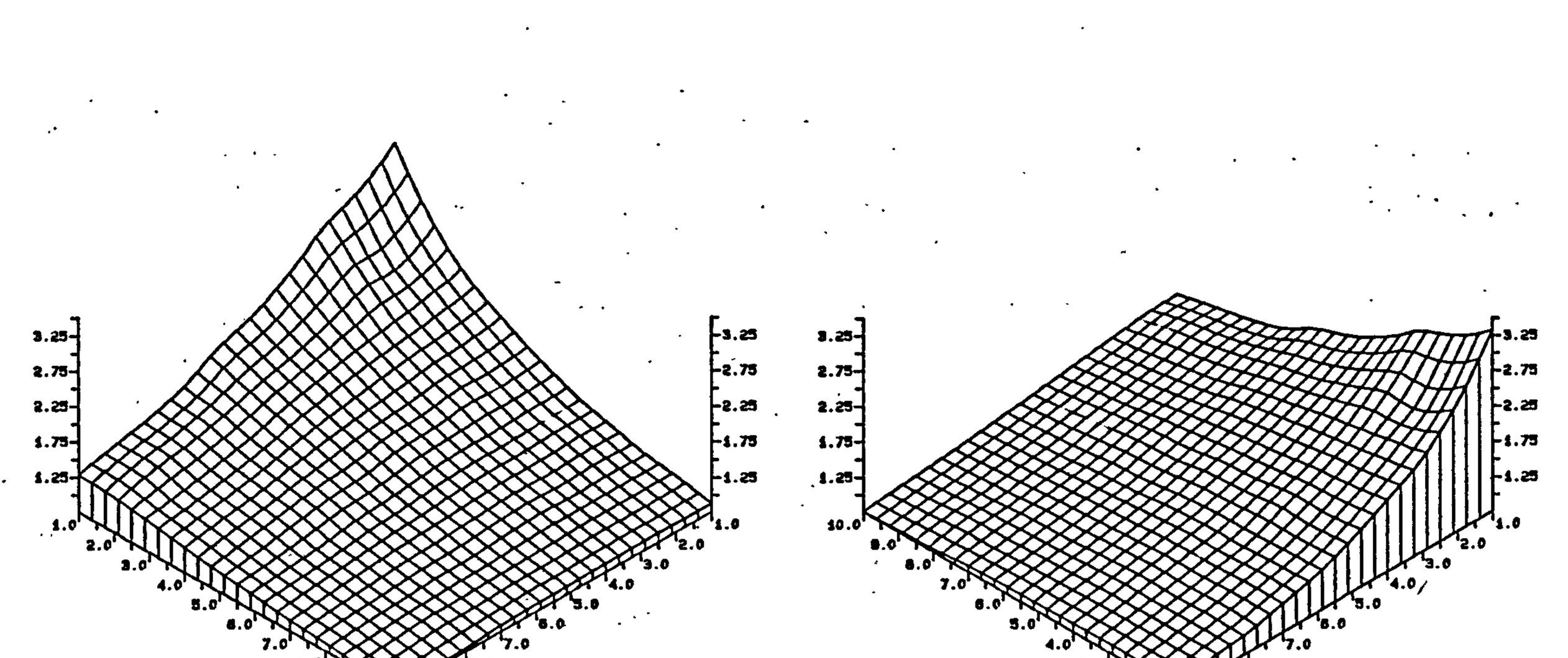


FIG. 4-15 DISTINCT NODE COMMS2 FED AT ALL NODES WITH DATA OF TYPE 10. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



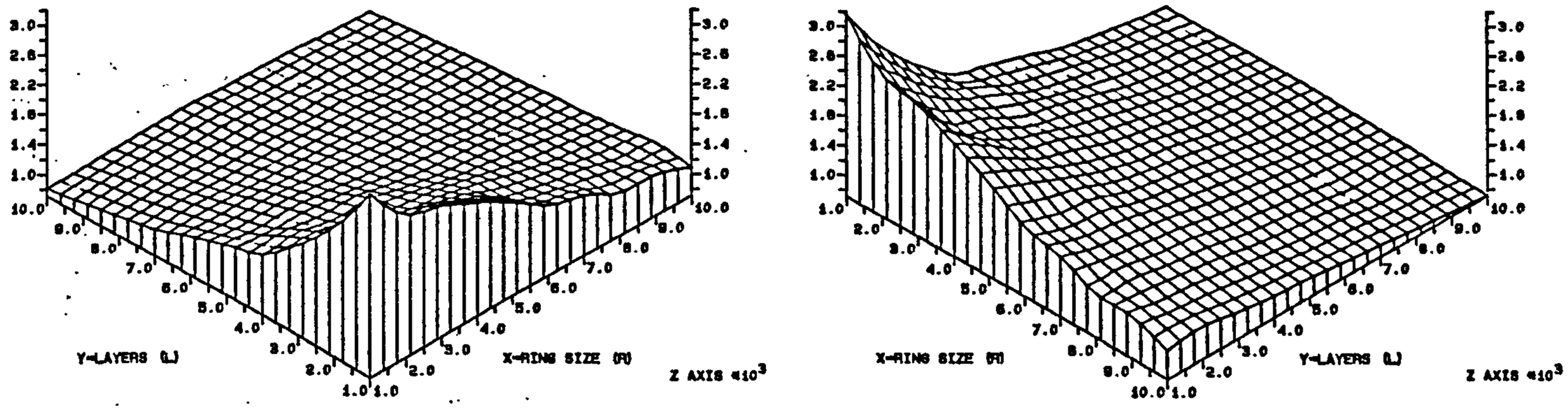
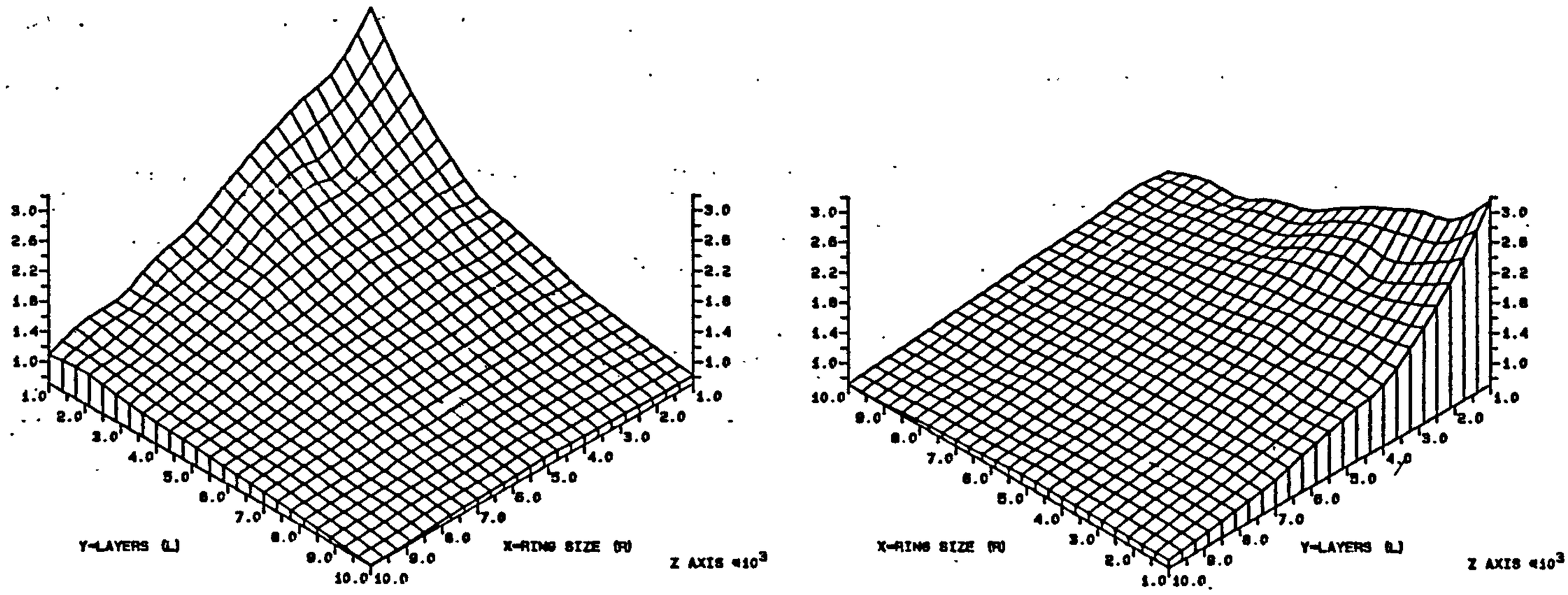
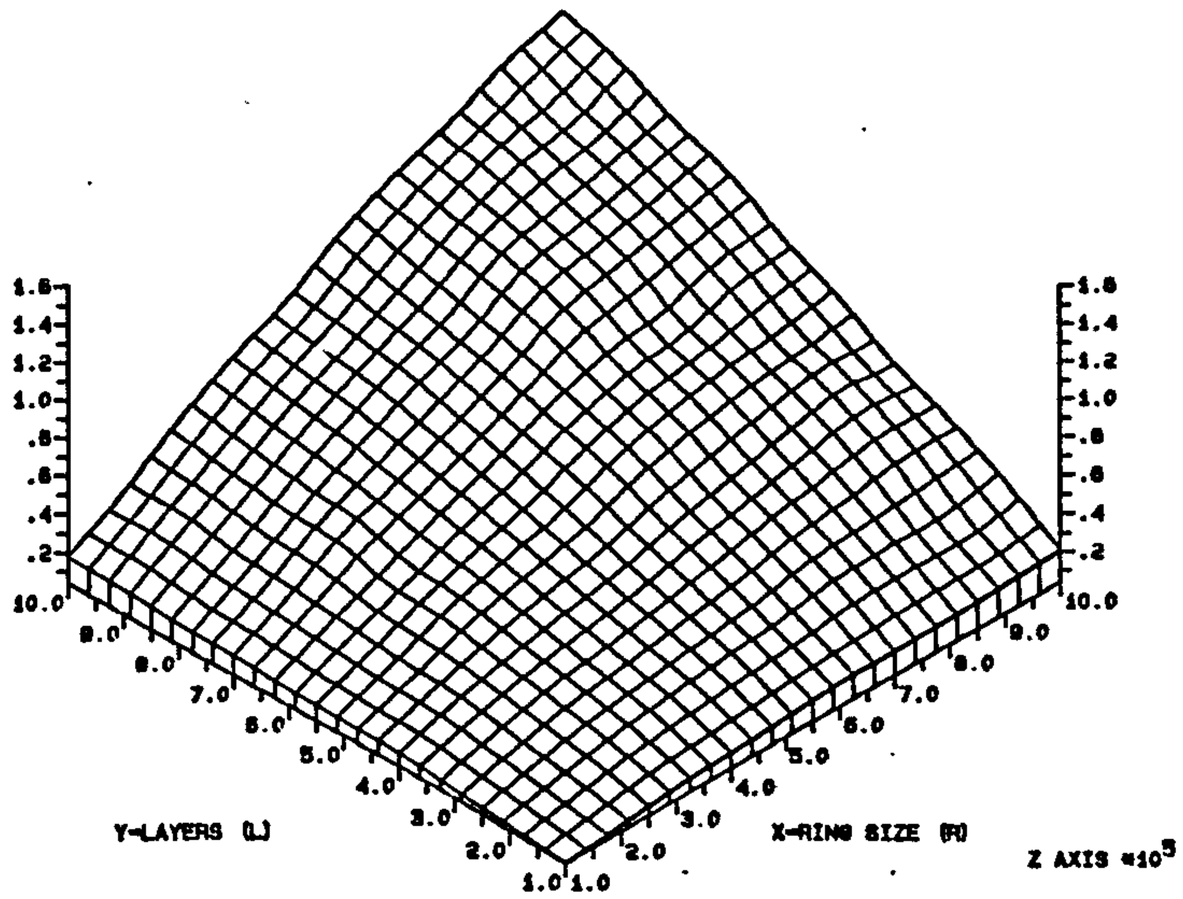


FIG. 4-16 DISTINCT NODE COMMS2 FED AT ALL NODES WITH DATA OF TYPE R20. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



With data of type 50 the value of total processing for R=L=10 over 1000 iterations predicted from the model is 2.198 × 50 × 1000 ≃ 110 000 events which is a reasonably close value (within 5%) of the performance achieved by the simulation (see fig 4.17). The simulation using randomly distributed data of mean value 50 showed a similar behaviour to that of the ring structure in that a very poor throughput was achieved with random data of large type and this was again attributed to wide range of the random numbers used and the simple buffering scheme of the simulation. The effect is not as serious as that in the case of the ring, because there are two output links to each node reducing the probability of data being help up by a busy node. Separating the queues of data to be processed and data to be forwarded to allow data to move through the system without being hindered by data awaiting processing would probably remove this effect. In the homogeneous cylindrical processor this effect does not occur as data is not held up by data awaiting processing. These simulations with large data type do not show the rounding off as L increases, the effect of the start up and algorithmic effects being less significant.

All of the plots of mean weighted total processing per node have the same form, the highest value being for a single processor, the plot decaying rapidly at first and then steadily as the number of processors is increased and the processing is distributed over a greater number of processors.



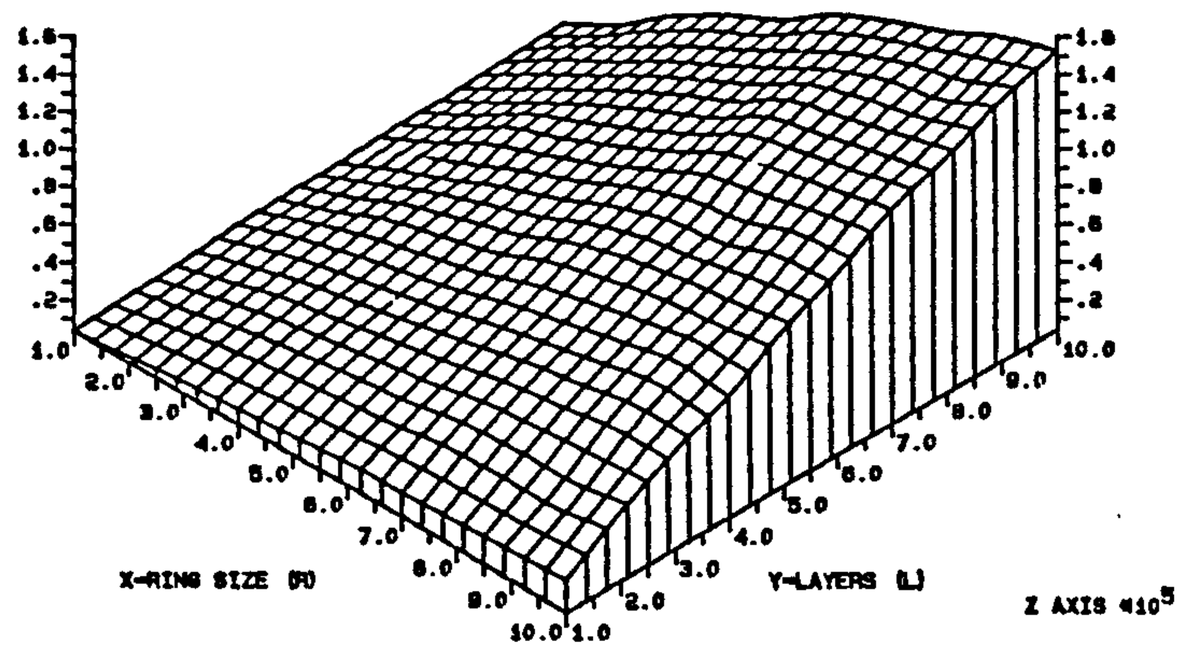
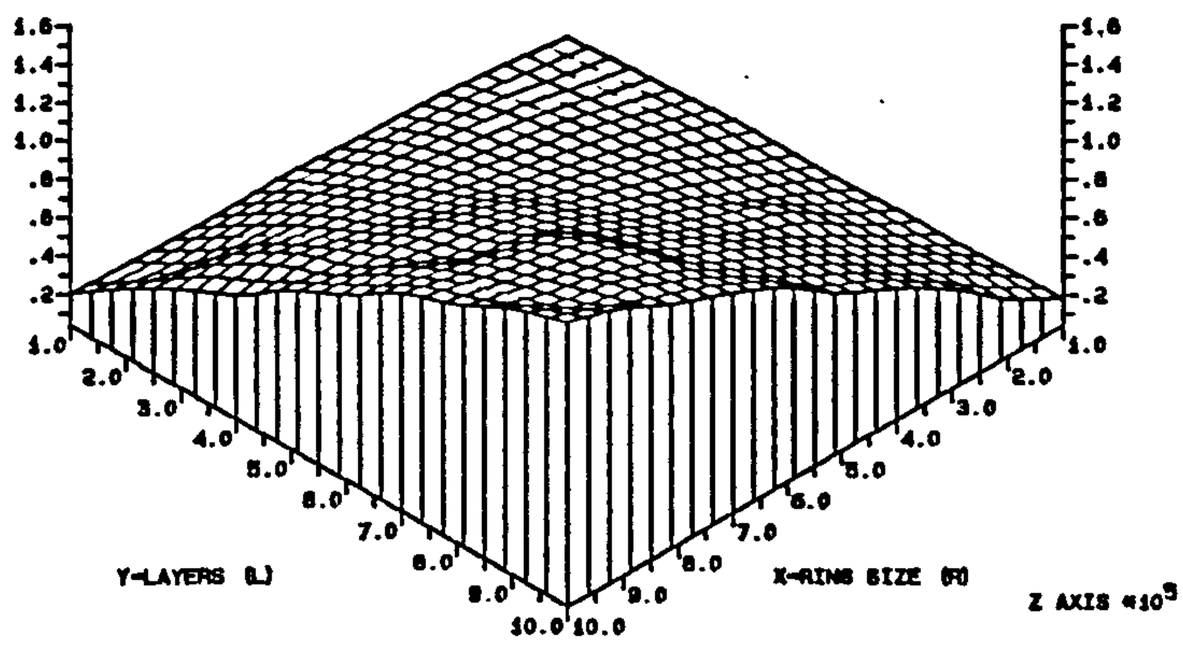
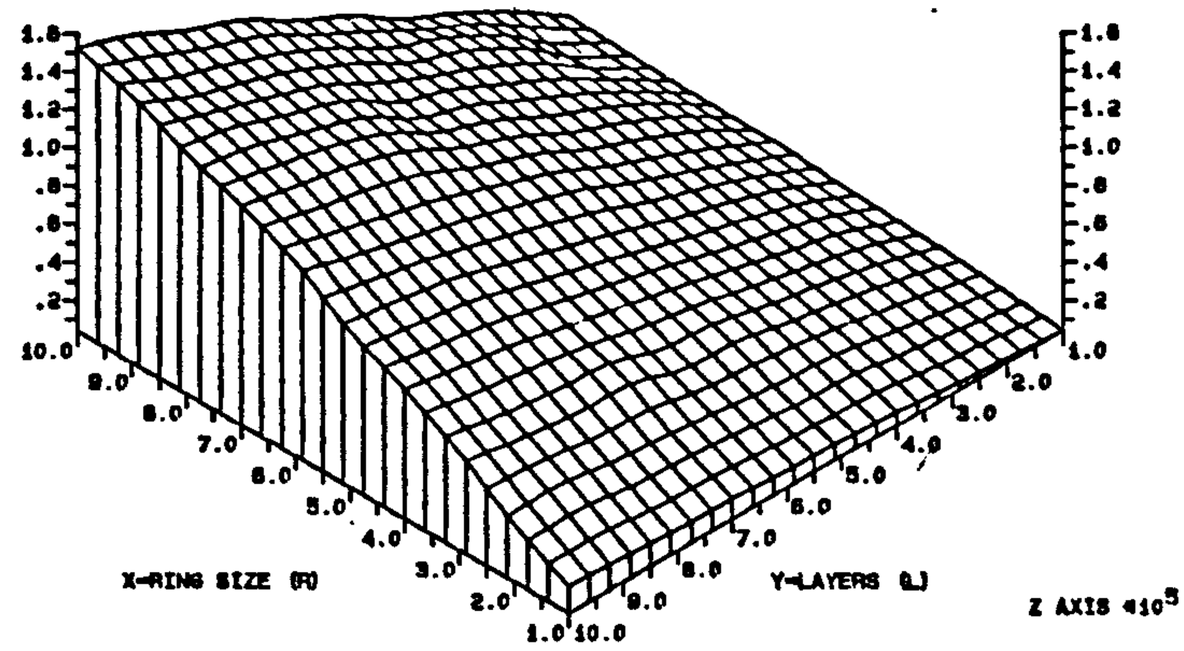
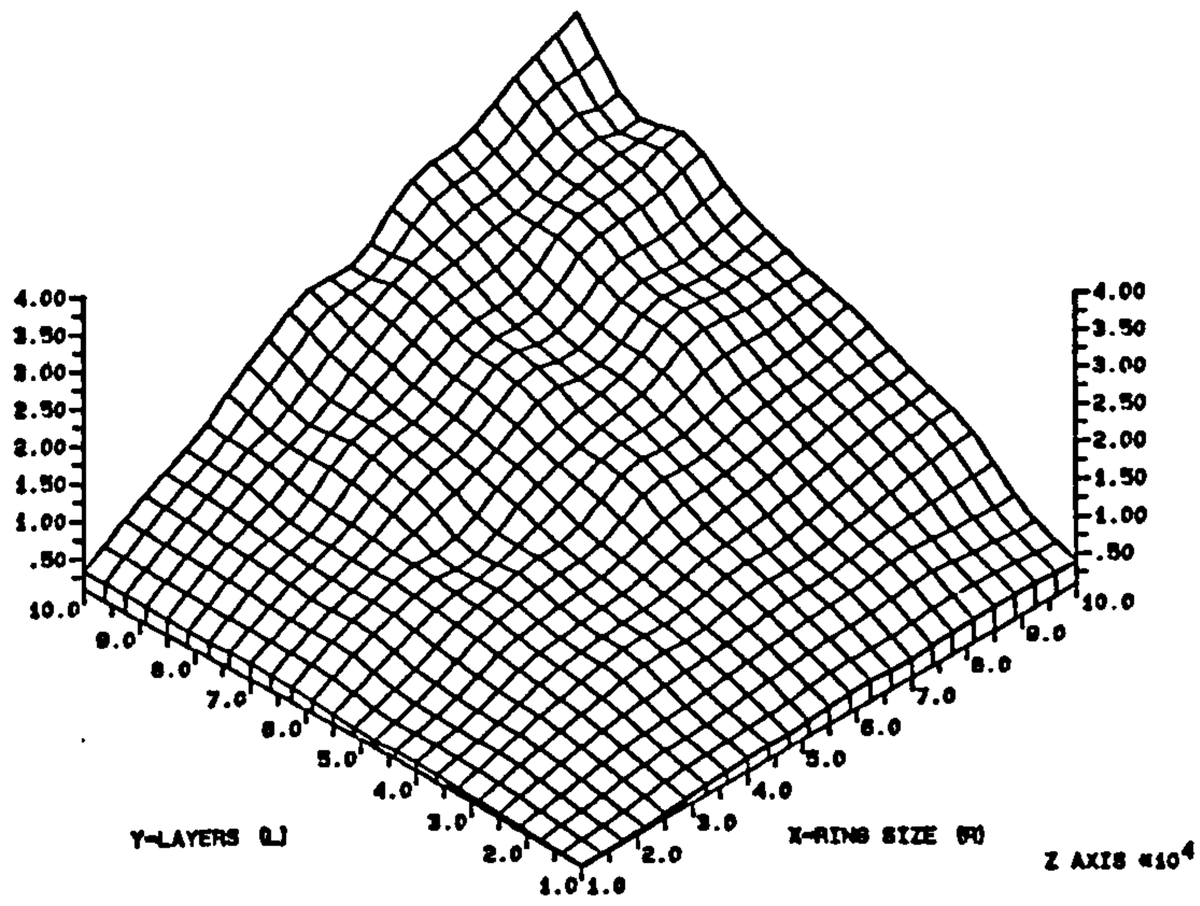


FIG. 4-17 DISTINCT NODE COMMS2 FED AT ALL NODES WITH DATA OF TYPE 50. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.







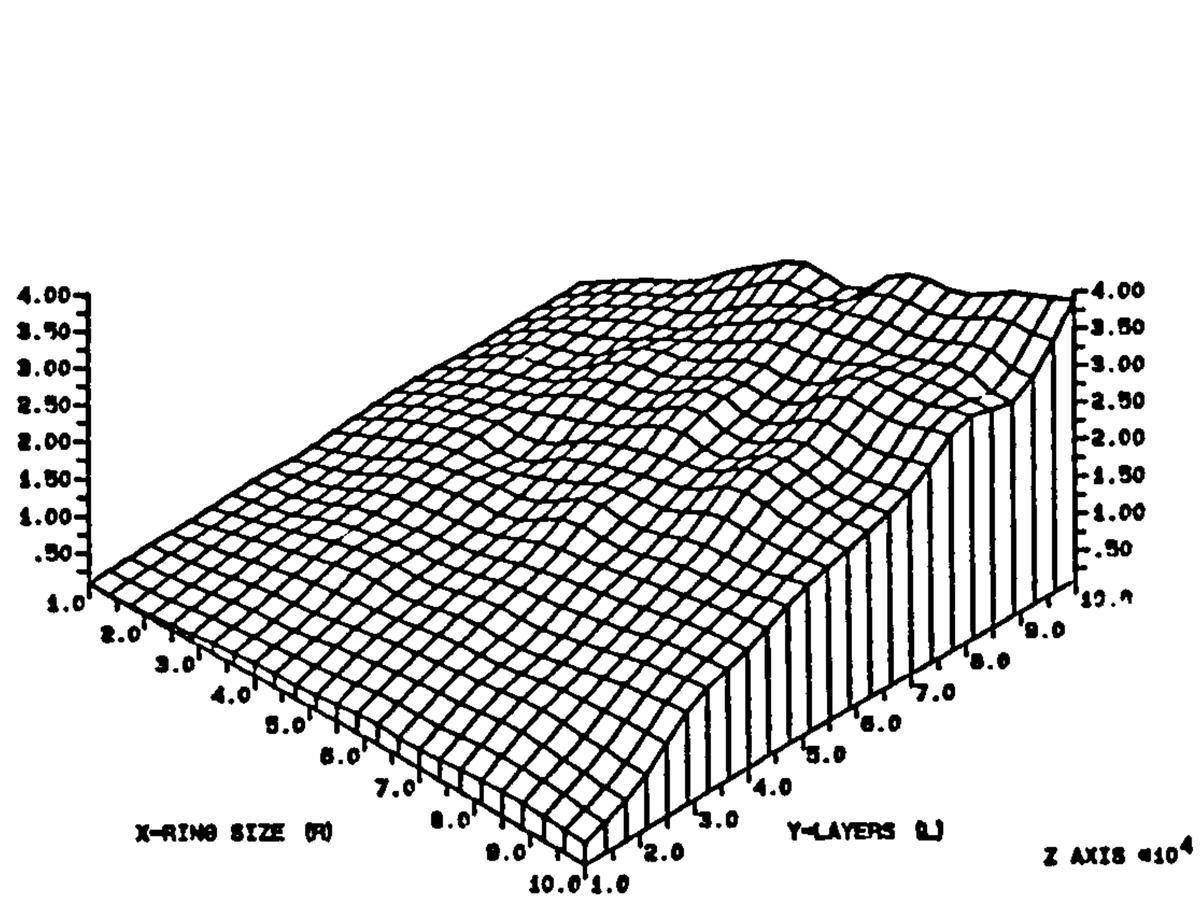
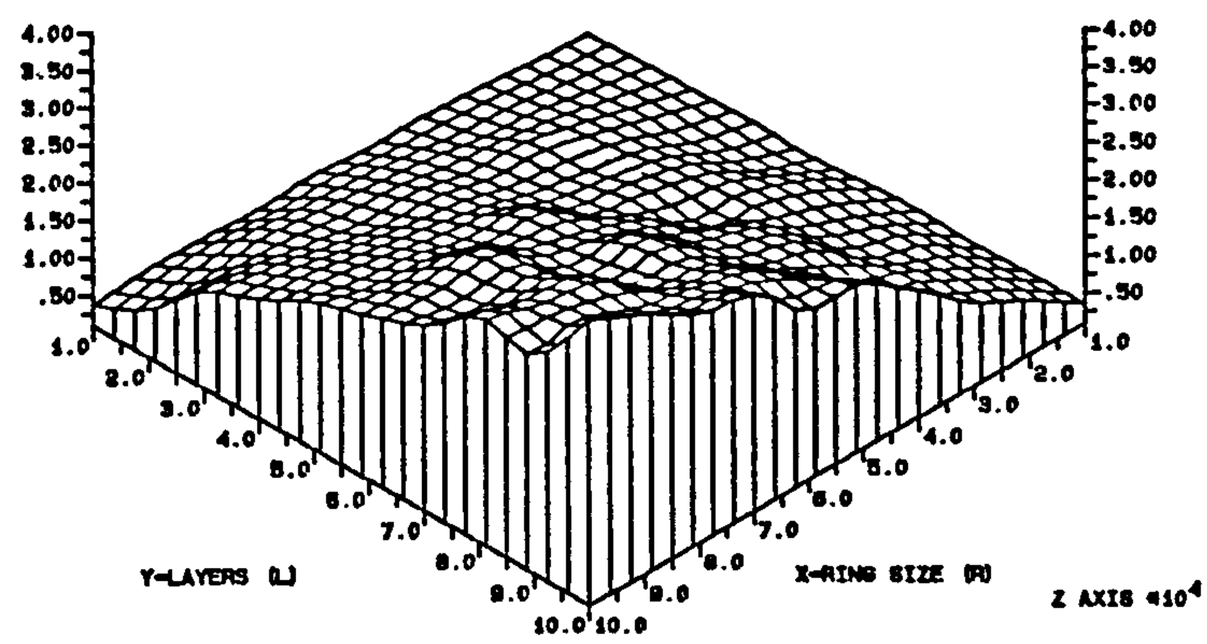
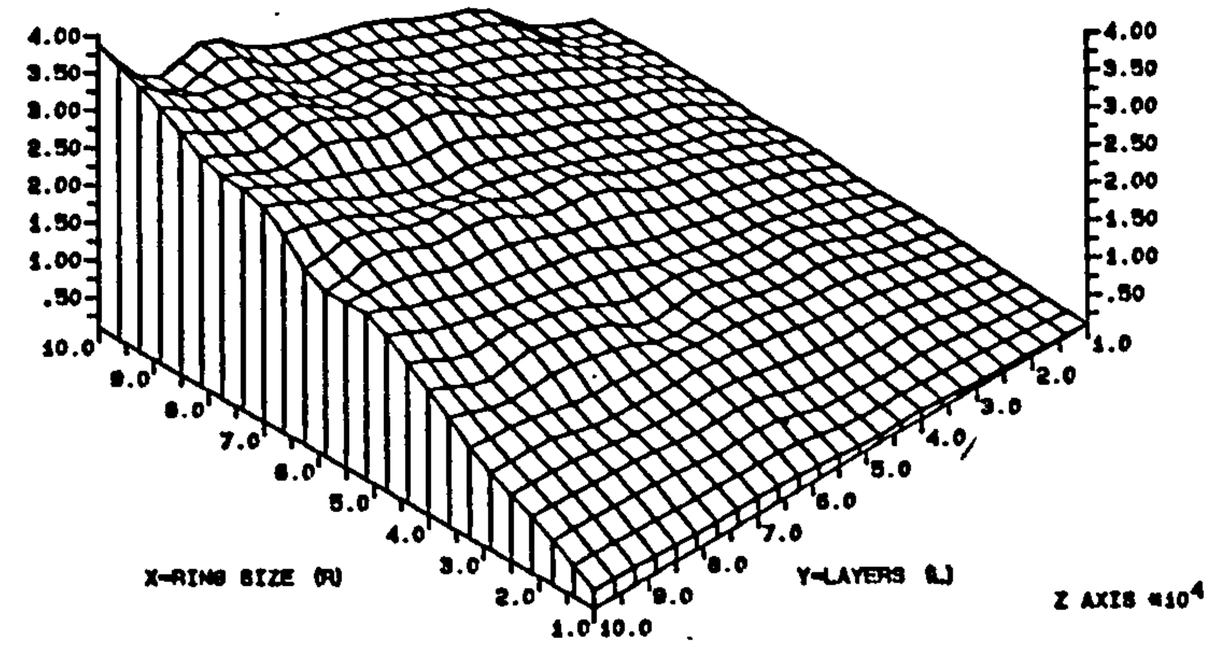


FIG. 4-18 DISTINCT NODE COMMS2 FED AT ALL NODES WITH DATA OF TYPE R100. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





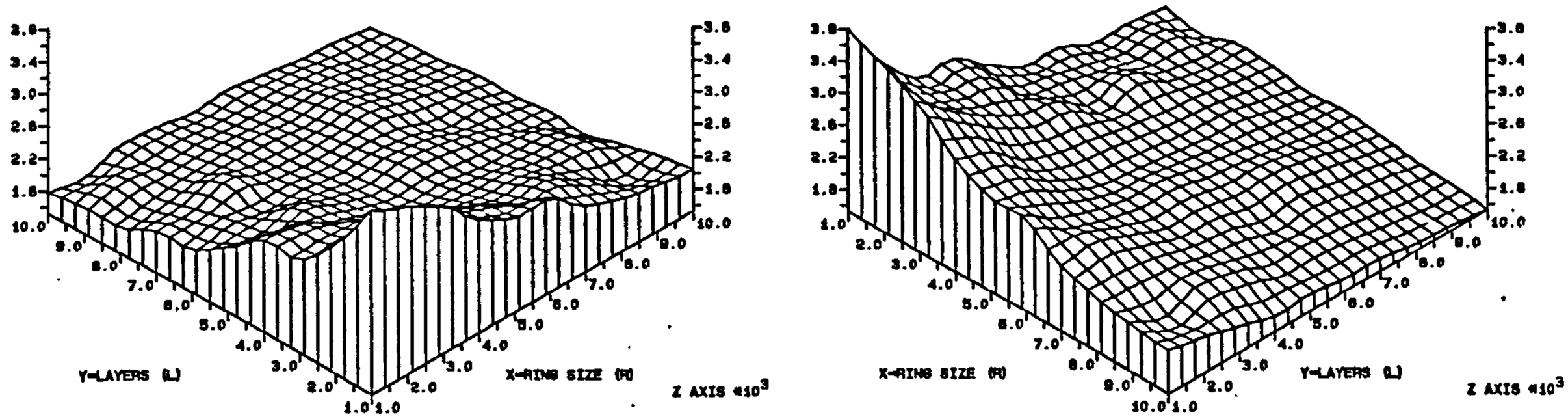
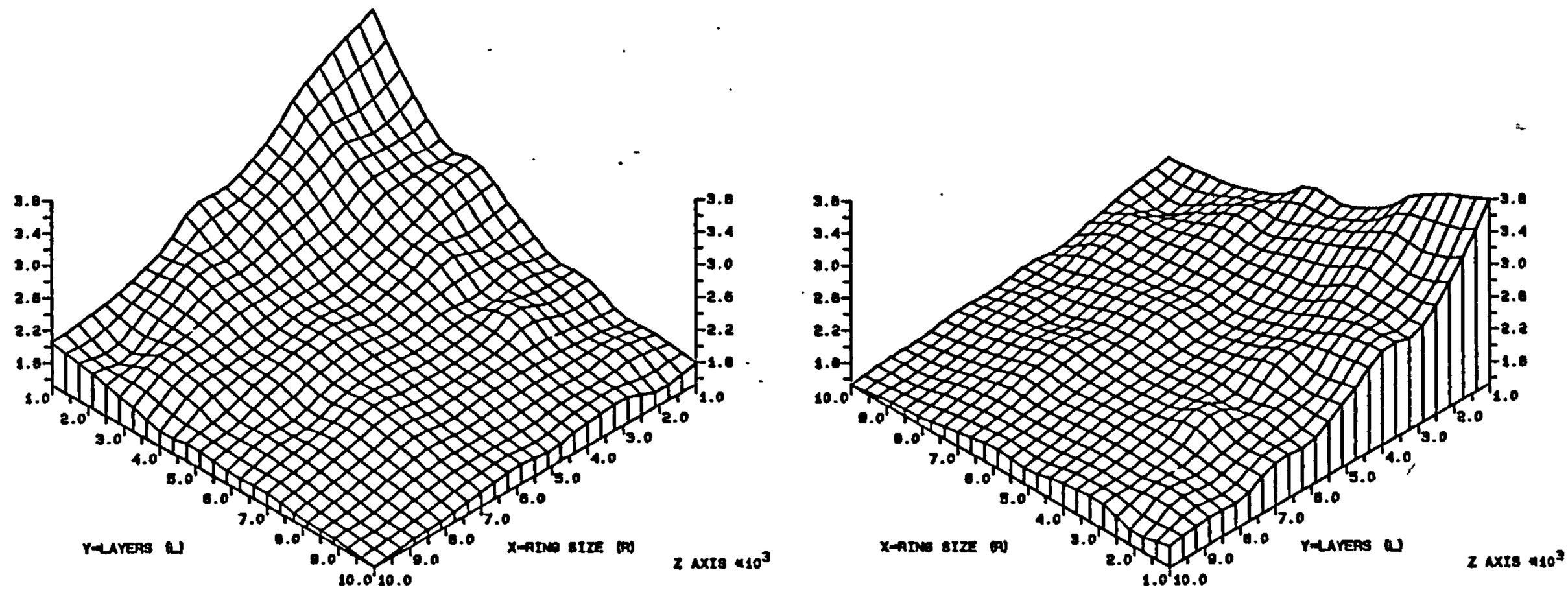


FIG. 4-19 DISTINCT NODE COMMS2 FED AT ALL NODES WITH DATA OF TYPE 50. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



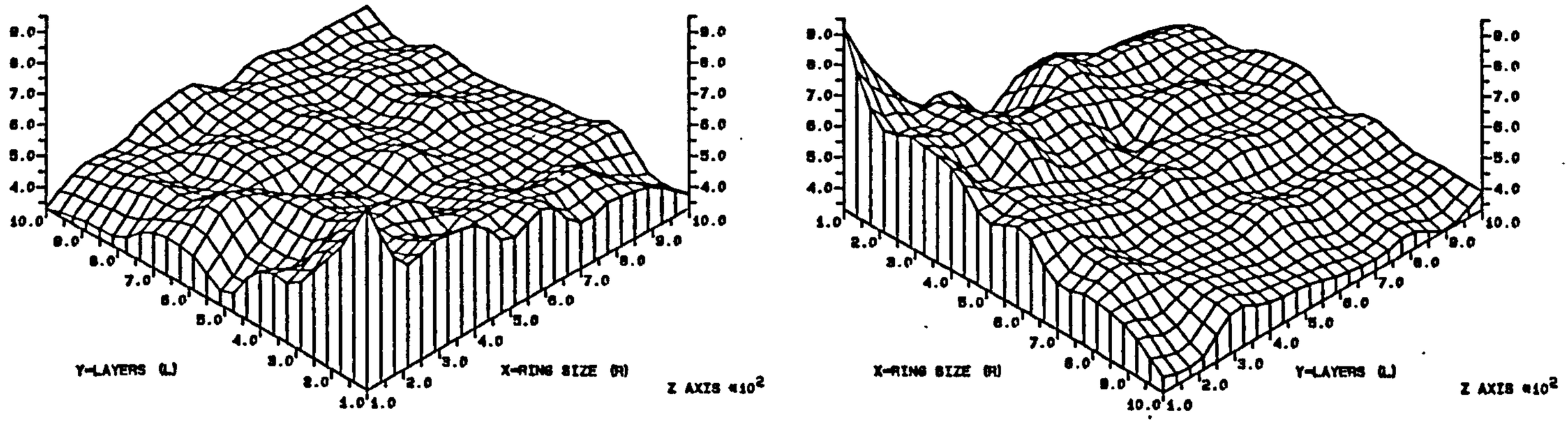
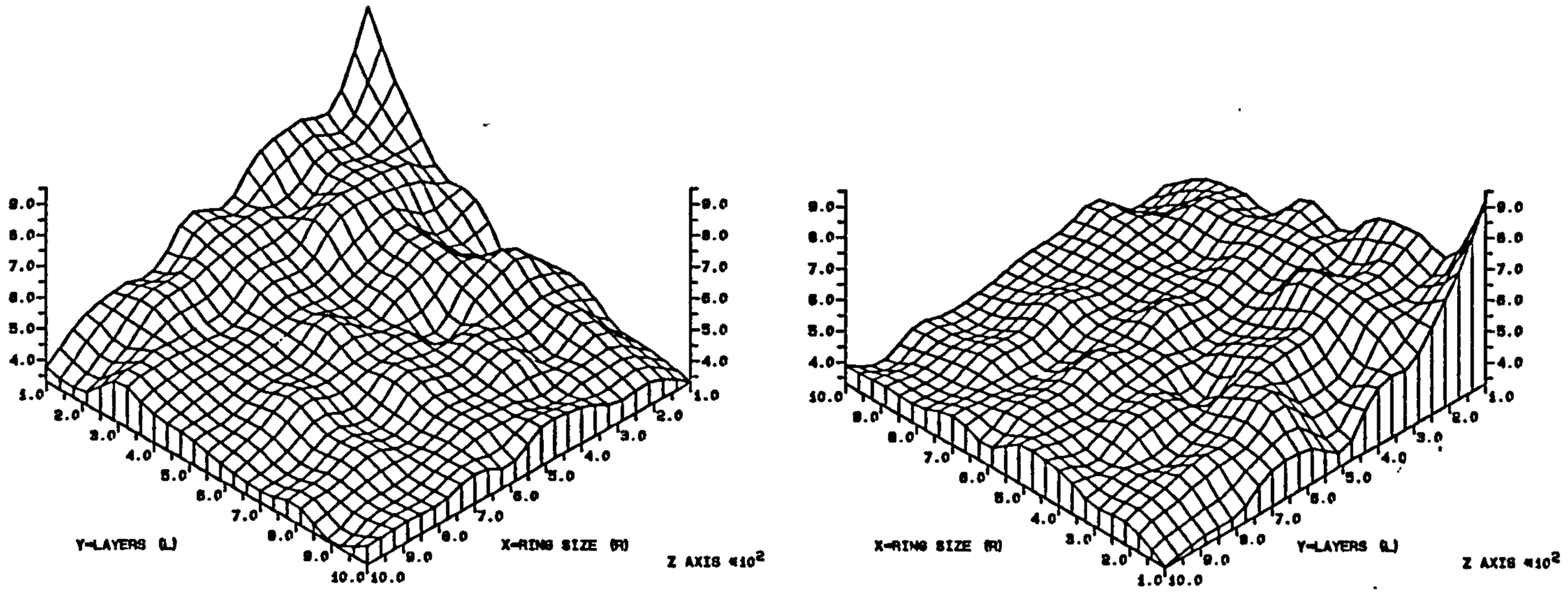


FIG. 4-20 DISTINCT NODE COMMS2 FED AT ALL NODES WITH DATA OF TYPE R100. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



4.12 Homogeneous Processing Simulation

4.12.1 Ring Simulation

4.12.1.1 Algorithms investigated

As with the case of distinct processing nodes a large number of possible communication schemes exist, these were reduced to two principle algorithms. The two algorithms used were essentially the same as those of the distinct node processing ring simulated, the statements required to compare the data with the identity of the processor could be omitted, providing a considerable simplification.

The two algorithms as they appear in the programs (which are to be found in appendix 3) are shown below, the same method of using short procedures and functions to allow an algorithmic style of programming was implemented.

```
(* COMMS1 *)
if processor_idle then
  if new_data then
    take_new_data
  else
    if ring_data then
      take_ring_data
    else
      (* NULL *)
else
  (* NULL *);
if ring_output_ready then
  if new_data then
    new_data_on
  else
    if ring_data then
      ring_data_on
     else
      (* NULL *)
else
  (* NULL *);
(* COMMS1 *)
```

```
(* COMMS2 *)
if processor_idle then
 if ring_data then
   take_ring_data
  else
     if new_data then
       take_new_data
    else
      (* NULL *)
else
  (* NULL *);
if ring_output_ready then
 if ring_data then
   ring_data_on
  else
     if new_data then
       new_data_on
    else
      (* NULL *)
else
  (* NULL *);
(* COMMS2 *)
```

4.12.1.2 Performance of the Algorithms

The nature of the homogeneous rings allowed the effect of different schemes of feeding data into the system such as only feeding a subset of the nodes with data and/or feeding different types of data into different nodes.

Simulations using data of type 1 were not performed as in the homogeneous ring data may be processed at any node that is free and data requiring such little processing as type 1 would be processed in the node to which it were fed immediately. For a ring where all nodes are fed with data the processing would be proportional to the number of fed processors in the ring. In the case of a ring fed at a single point only one processor would be kept busy and all but one of the nodes in the ring would remain idle, (this can be compared to the lower layers remaining idle in the case of a cylinder fed with data of type 1 as described in section 4.12.2.2).

Fig 4.21 shows the performance of a homogeneous ring of processing nodes fed at all nodes with data of type 10. Lines 1 and 2 show the behaviour of communication algorithms 1 and 2 fed with data of constant type 10 respectively and lines 3 and 4 show the behaviour of communication algorithms 1 and 2 fed with randomly distributed data with a mean value of 10 respectively. For such data, as for the distinct node simulations

$$\beta_{cmax} = 1/\lceil 10/4 \rceil = 1/3$$

and $K_r = 1/4$

from the discussion in chapter 3 of such a processing scheme the processing bandwidth is given by

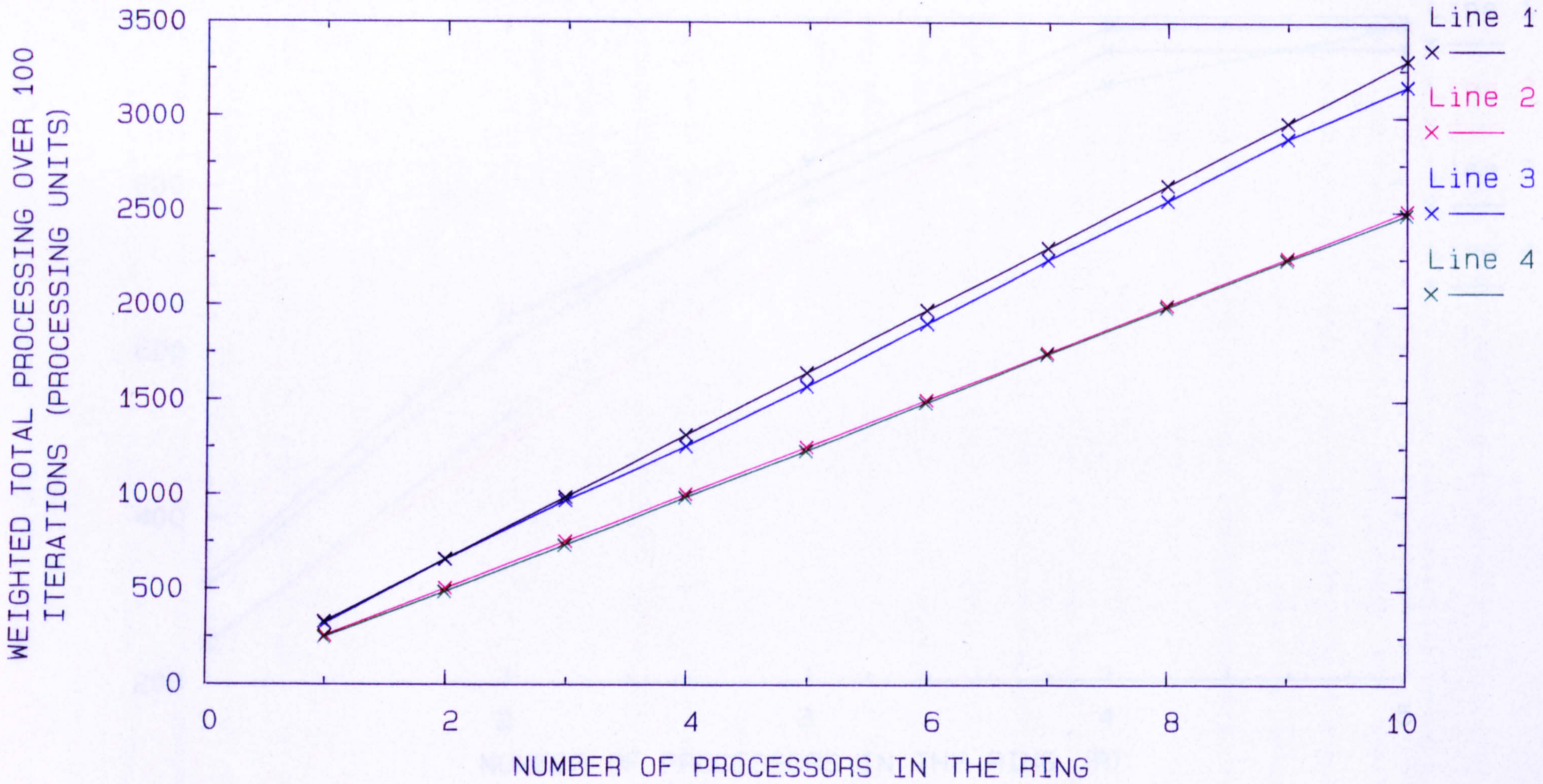
$$\text{Total Processing} = R \cdot \beta_c = \frac{R \cdot \beta_{cmax}}{1+K_r}$$

$$= R \times 0.266 \text{ events/iteration}$$

$$= R \times 2.66 \text{ processing units/iteration}$$

there is reasonably good agreement between this and the values in fig 4.21. The values for randomly distributed data being almost identical to those for data with uniform processing requirements illustrating some validity in the assumption made that horizontal communication is either insignificant or mutually cancelling.

FIG. 4-21 HOMOGENEOUS RING STRUCTURE FED AT ALL POINTS WITH DATA OF TYPE 10



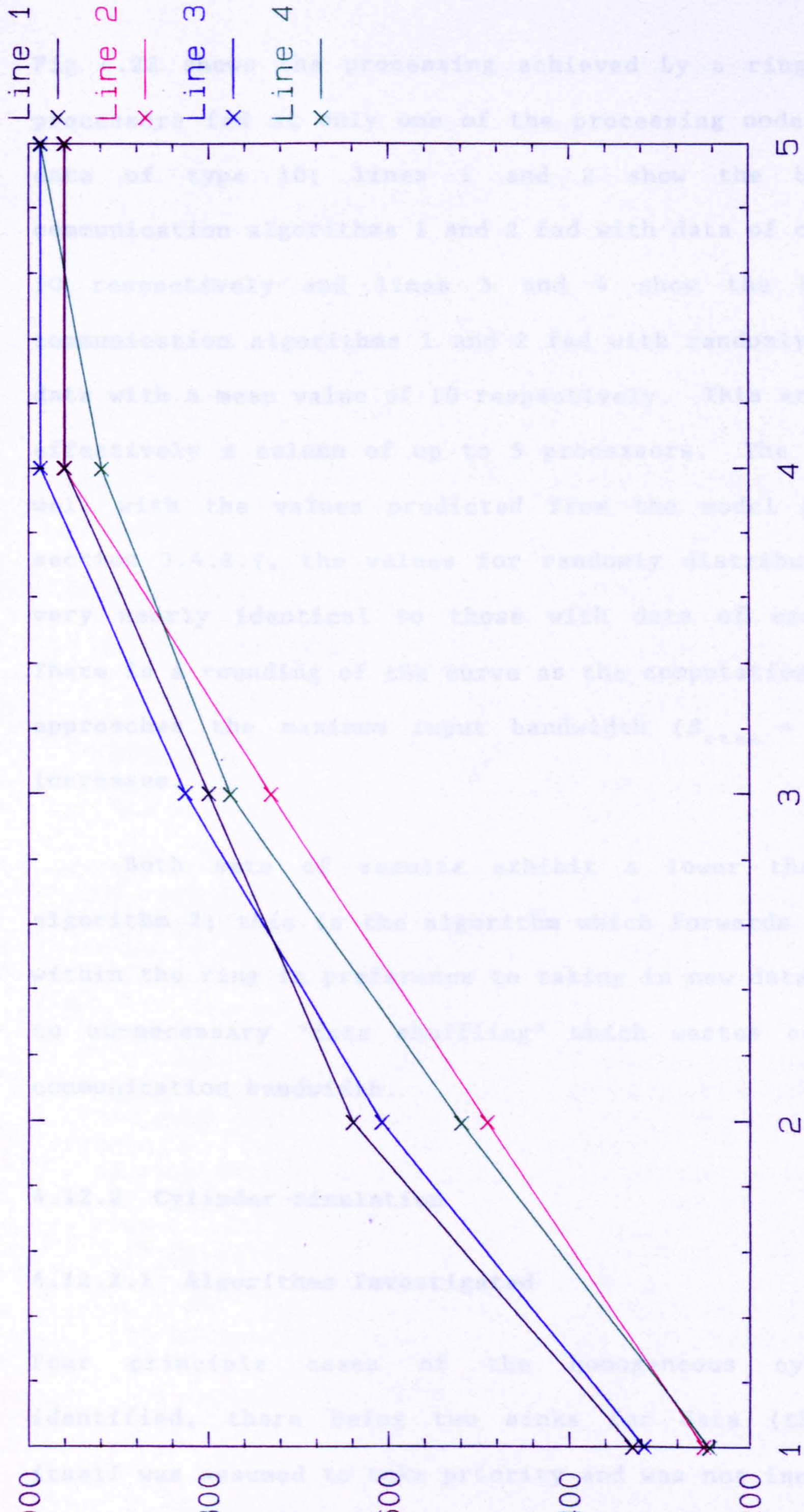


FIG. 4-22 HOMOGENEOUS RING STRUCTURE FED AT ONE POINT WITH DATA OF TYPE 10

Fig 4.22 shows the processing achieved by a ring of up to 5 processors fed at only one of the processing nodes with input data of type 10; lines 1 and 2 show the behaviour of communication algorithms 1 and 2 fed with data of constant type 10 respectively and lines 3 and 4 show the behaviour of communication algorithms 1 and 2 fed with randomly distributed data with a mean value of 10 respectively. This arrangement is effectively a column of up to 5 processors. The values agree well with the values predicted from the model presented in section 3.4.2.7, the values for randomly distributed data are very nearly identical to those with data of constant type. There is a rounding of the curve as the computational bandwidth approaches the maximum input bandwidth ($\beta_{ctot} \rightarrow \beta_{phys}$) as R increases.

Both sets of results exhibit a lower throughput for algorithm 2; this is the algorithm which forwards data already within the ring in preference to taking in new data. This lead to un-necessary 'data shuffling' which wastes computing and communication bandwidth.

4.12.2 Cylinder Simulation

4.12.2.1 Algorithms Investigated

Four principle cases of the homogeneous cylinder were identified, there being two sinks for data (the processor itself was assumed to take priority and was not included as one of the permutations) and two sources for data. In keeping with the distinct node case three sections of code were used, corresponding to the three data sinks.

The same algorithmic style of programming as before was used, in addition to the two cases explored for the distinct node cylinder the effect of changing the order in which the sections of code to send data to the two output sinks (the downward and horizontal outputs) were placed was examined. The four algorithms thus defined are shown below, as they appear in the simulation programs, copies of which appear in appendix 4.

```
begin (* COMMS1 *)
if processor_idle then
  if new_data then
    take_new_data
  else
    if ring_data then
      take_ring_data
    else
      (* NULL *)
else
  (* NULL *);
if ring_ready then
  if new_data then
    new_data_on
  else
    if ring_data then
      ring_data_on
    else
      (* NULL *)
else
  (* NULL *);
if down_ready then
  if new_data then
    new_data_down
  else
    if ring_data then
      ring_data_down
    else
      (* NULL *)
else
  (* NULL *);
end;  (* COMMS1 *)
```

```
begin (* COMMS2 *)
if processor_idle then
 if ring_data then
   take_ring_data
 else
    if new_data then
      take_new_data
   else
     (* NULL *)
else
  (* NULL *);
if ring_ready then
 if ring_data then
   ring_data_on
 else
    if new_data then
      new_data_on
   else
     (* NULL *)
else
  (* NULL *);
if down_ready then
 if ring_data then
   ring_data_down
 else
    if new_data then
      new_data_down
   else
     (* NULL *)
else
  (* NULL *);
end;  (* COMMS2 *)
```

```
begin (* COMMS3 *)
if processor_idle then
  if new_data then
    take_new_data
  else
    if ring_data then
      take_ring_data
    else
      (* NULL *)
else
  (* NULL *);
if down_ready then
  if new_data then
    new_data_down
  else
    if ring_data then
      ring_data_down
    else
      (* NULL *)
else
  (* NULL *);
if ring_ready then
  if new_data then
    new_data_on
  else
    if ring_data then
      ring_data_on
    else
      (* NULL *)
else
  (* NULL *);
end;  (* COMMS3 *)
```

```
begin (* COMMS4 *)
if processor_idle then
 if ring_data then
   take_ring_data
  else
     if new_data then
       take_new_data
    else
      (* NULL *)
else
  (* NULL *);
if down_ready then
 if ring_data then
   ring_data_down
  else
     if new_data then
       new_data_down
    else
      (* NULL *)
else
  (* NULL *);
if ring_ready then
 if ring_data then
   ring_data_on
  else
     if new_data then
       new_data_on
    else
      (* NULL *)
else
  (* NULL *);
end;  (* COMMS4 *)
```

4.12.2.2 Performance of the Algorithms

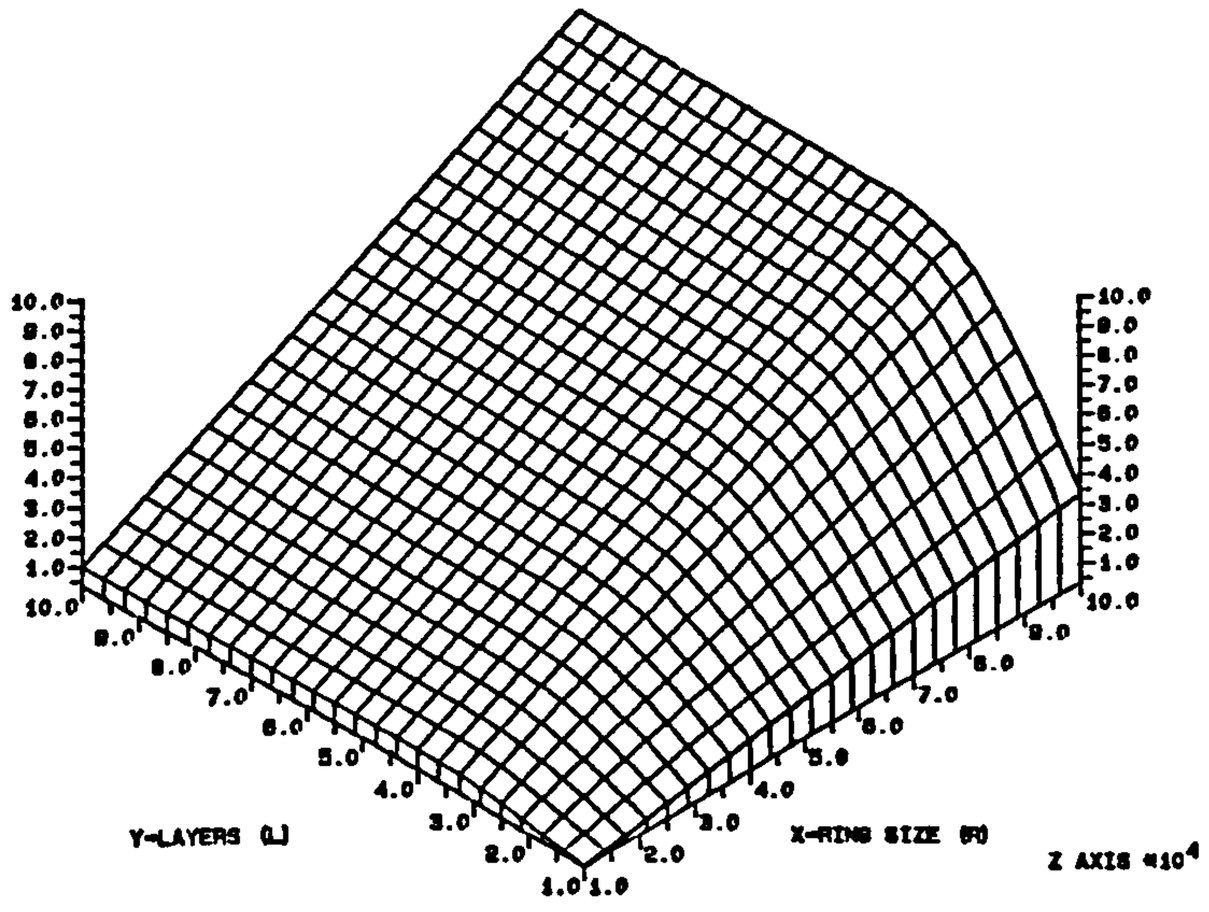
4.12.2.2.1 Processing Throughput

The overall pattern of behaviour of these algorithms with different data types was found to take the same form, the performance of the system when fed with randomly distributed data was almost identical to that of the constant data type case and the behaviour of algorithms 1 and 2 was found to correspond strongly with that of algorithms 3 and 4 respectively; therefore, in the interests of brevity only a few illustrative cases will be discussed here and a complete set of simulation results are to be found in appendix 5.

The performance of a cylinder fed at all of its uppermost nodes with data of type 10 shown in fig 4.23 clearly shows that there is a maximum useful value of L beyond which no improvement (in the absence of faults) is achieved with additional processors; once L has reached this value the performance improves with increasing R (apparently indefinitely). The performance of this system agrees well with that predicted from the model of section 3.4.2.7, there is a slightly lower throughput using algorithm 2 (and similarly algorithm 4 c.f. appendix 5) as a result of the algorithms preference for horizontal communication producing a tendency towards un-necessary 'data shuffling'. This lower processing throughput is also shown in fig 4.25, the weighted total processing per processor for the two systems, though other details are similar. As R increases there is no decrease of processing throughput as in the distinct processor case (see fig 4.15) because there is no corresponding increase in the distance data must travel as R increases. There is a slight decrease in processing per node, due to the effect of K_r, as L increases up to the maximum useful value followed by a steep decline when this value is exceeded.

The performance of a cylinder fed at only one point with data of type 10 shows the effect of saturation of the point at which data is fed into the system. There is a limited number of nodes (in this case 4) that can be usefully employed. For a linear chain of processors the methods described in section 3.4.2.7 may be directly applied and the system exhibits generally good agreement with these values. The corresponding results using data of type 50 shown in fig 4.27 exhibit a

similar overall behaviour, however the processing values do not correspond very well with those predicted from the model. This occurs as a result of the 'quantisation' of the simulation; with data of type 50 very few events will require re-transmission by a node while it is processing an event, though processing effort would be consumed for each action there is a likelihood in a large number of cases of the number of iterations required to finish the event remaining unchanged with a consequent reduction in the apparent value of K_r. The predicted and obtained values differ approximately by a factor of 2 and only a small change in the effective value of K_r could easily produce such a change in the predicted value.



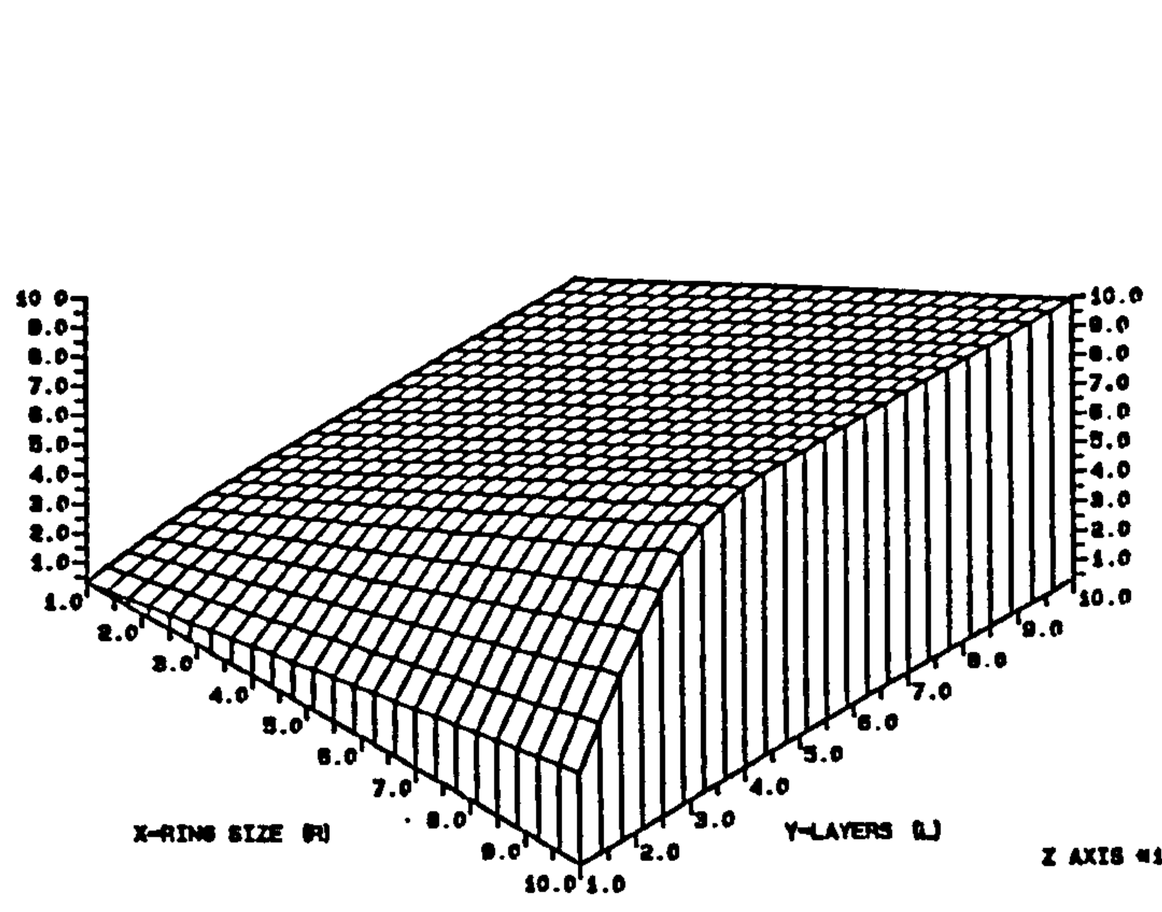
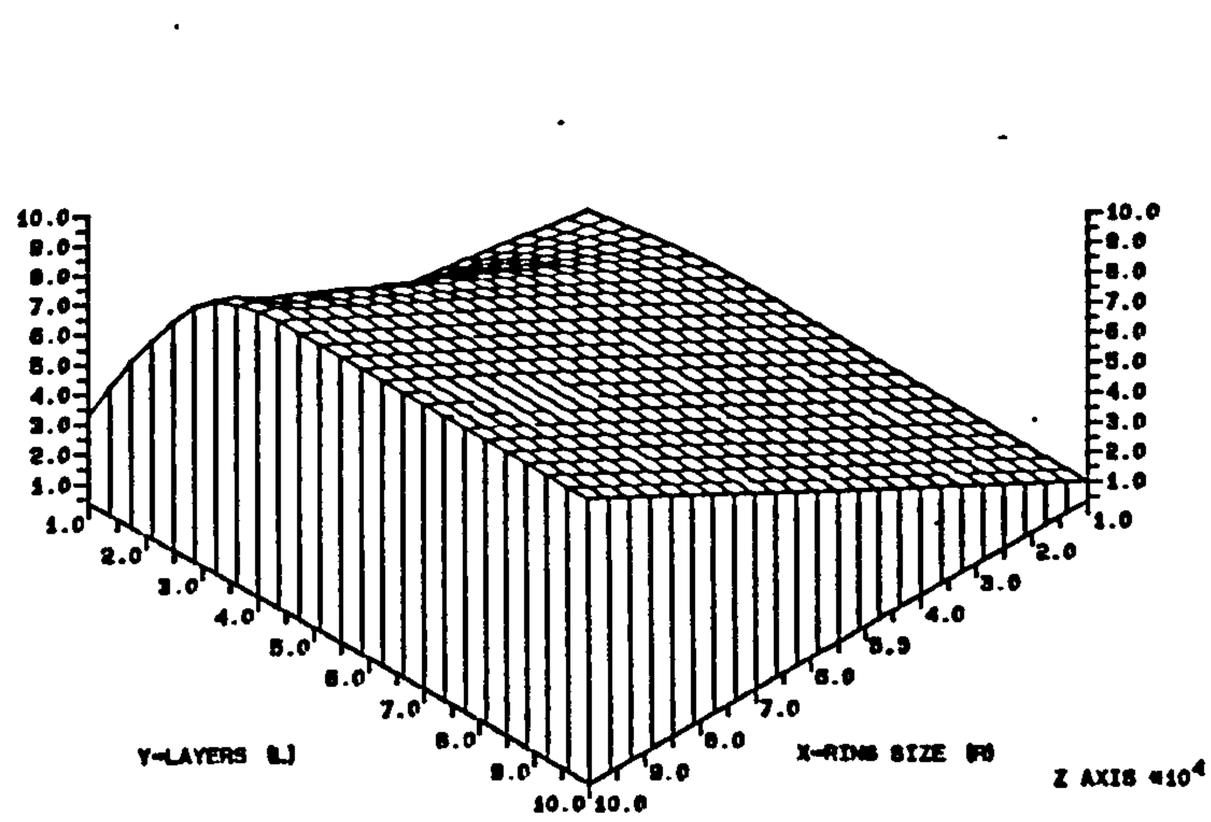
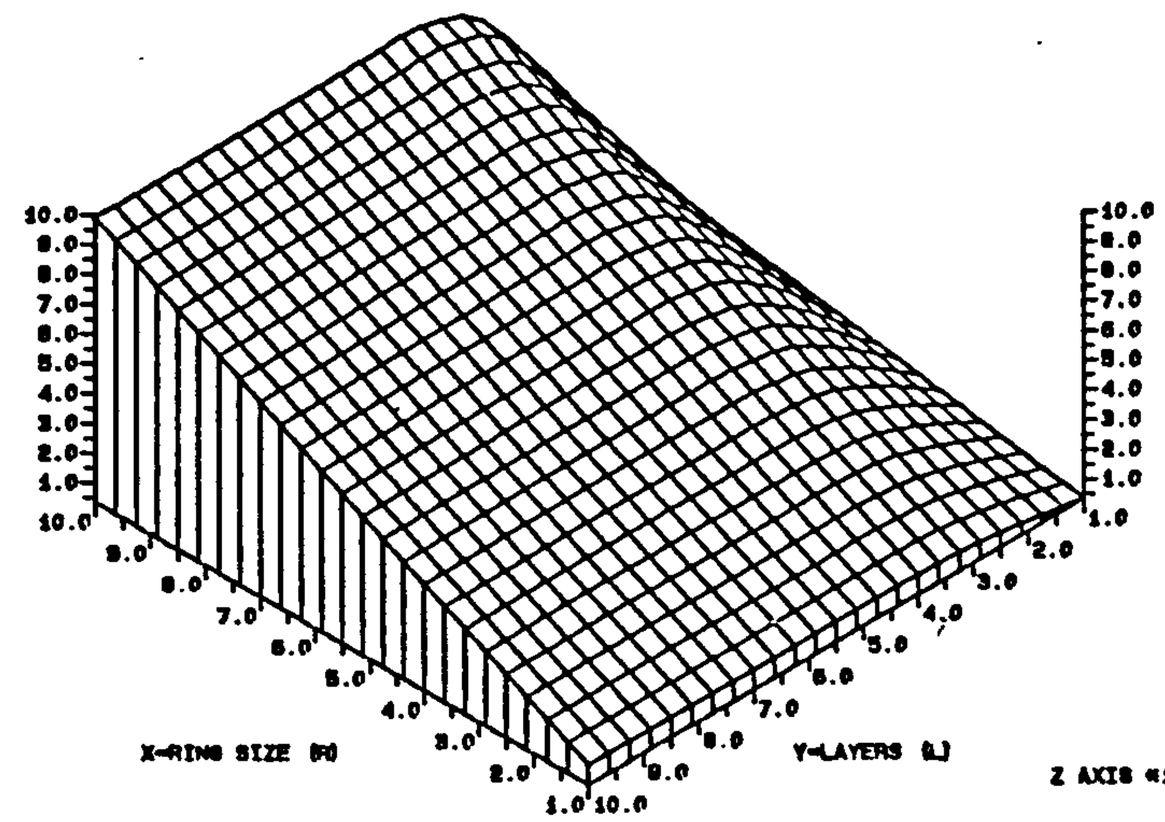
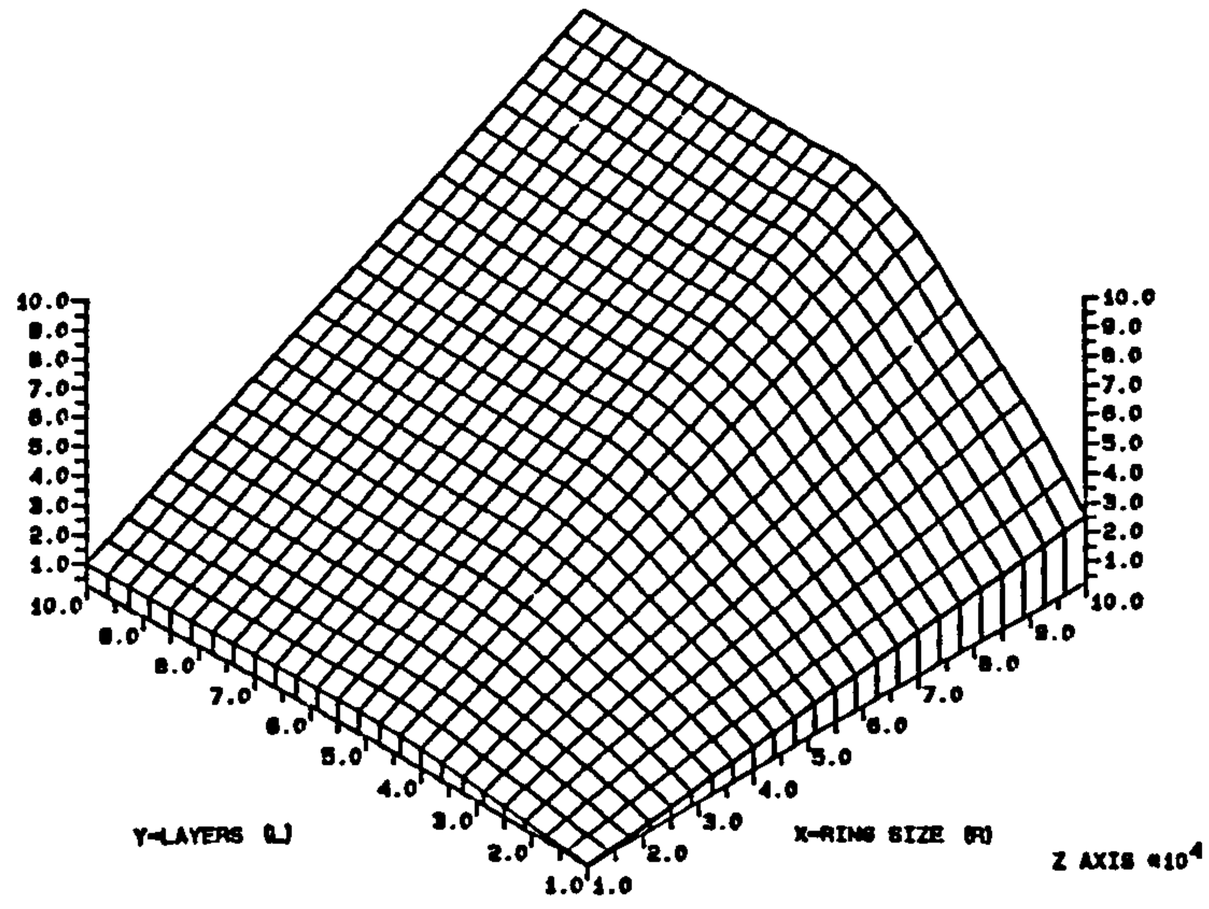


FIG. 4-23.1 HOMOGENEOUS COMMS1 FED AT ALL POINTS WITH DATA OF TYPE 10. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS







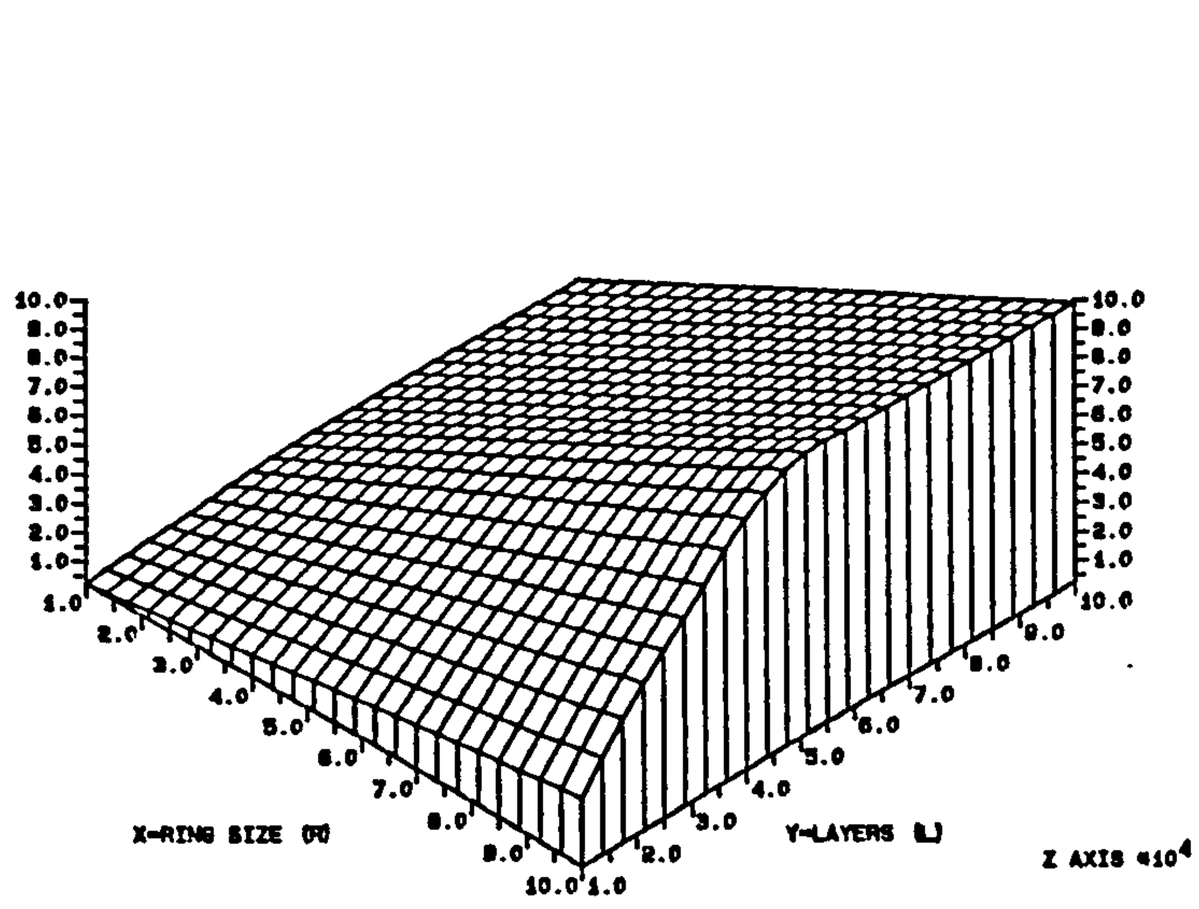
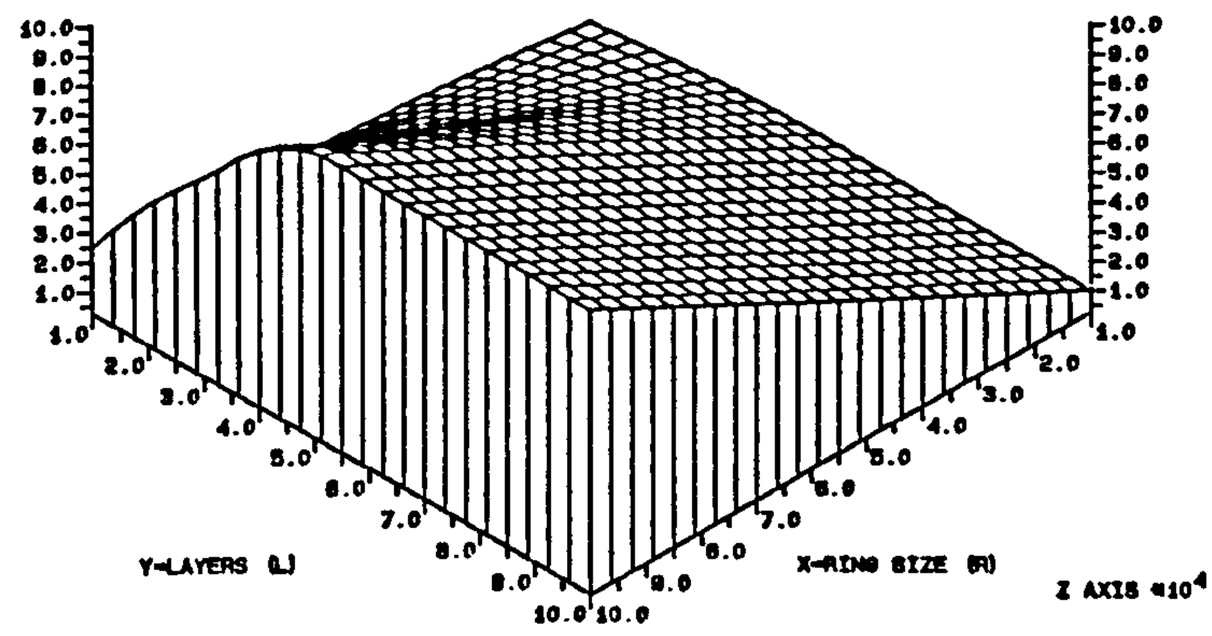
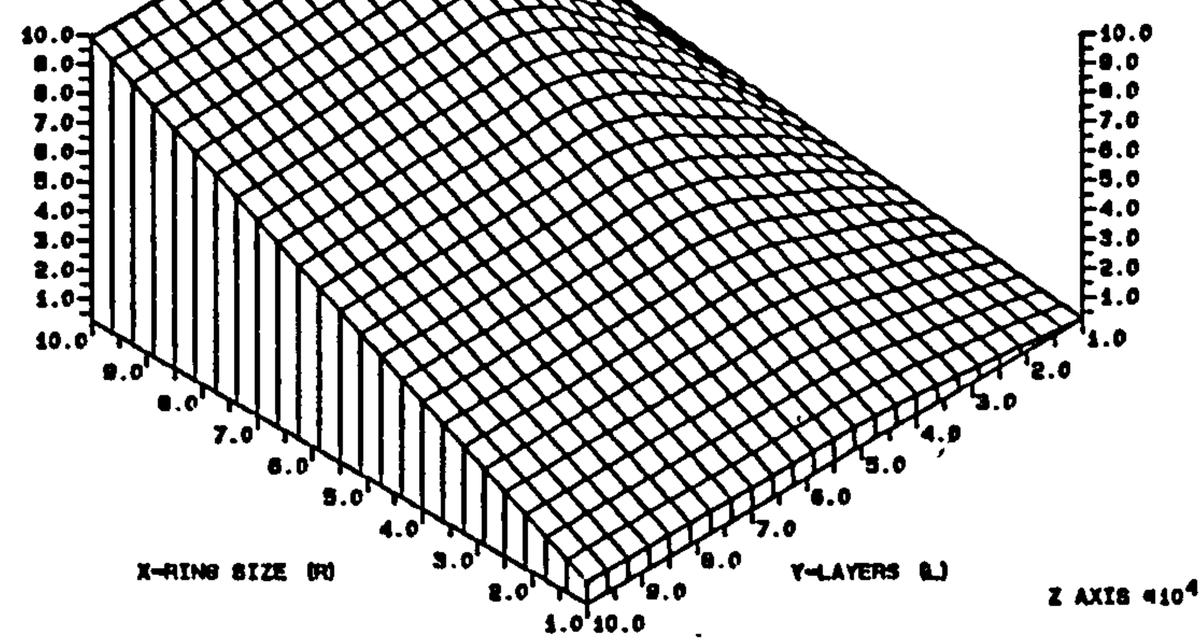
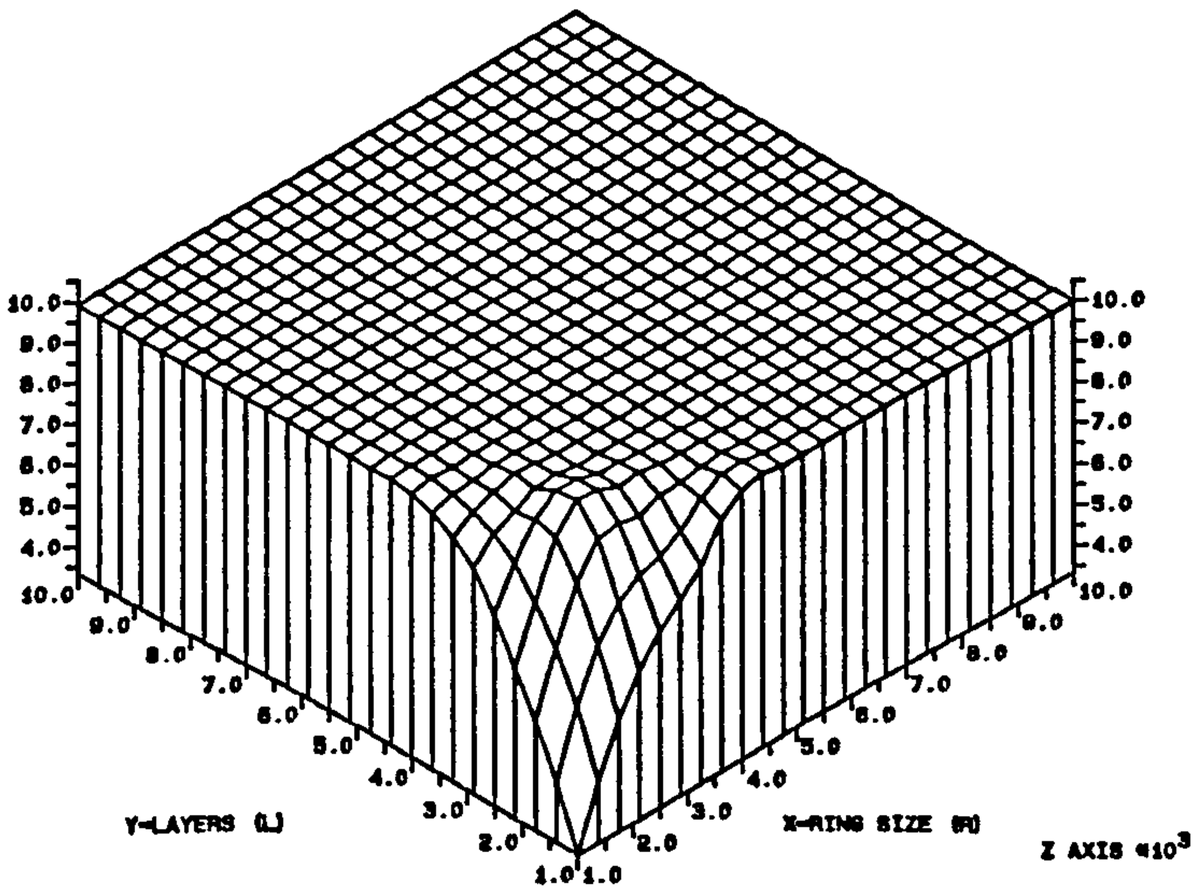


FIG. 4-23.2 HOMOGENEOUS COMMS2 FED AT ALL POINTS WITH DATA OF TYPE 10. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS







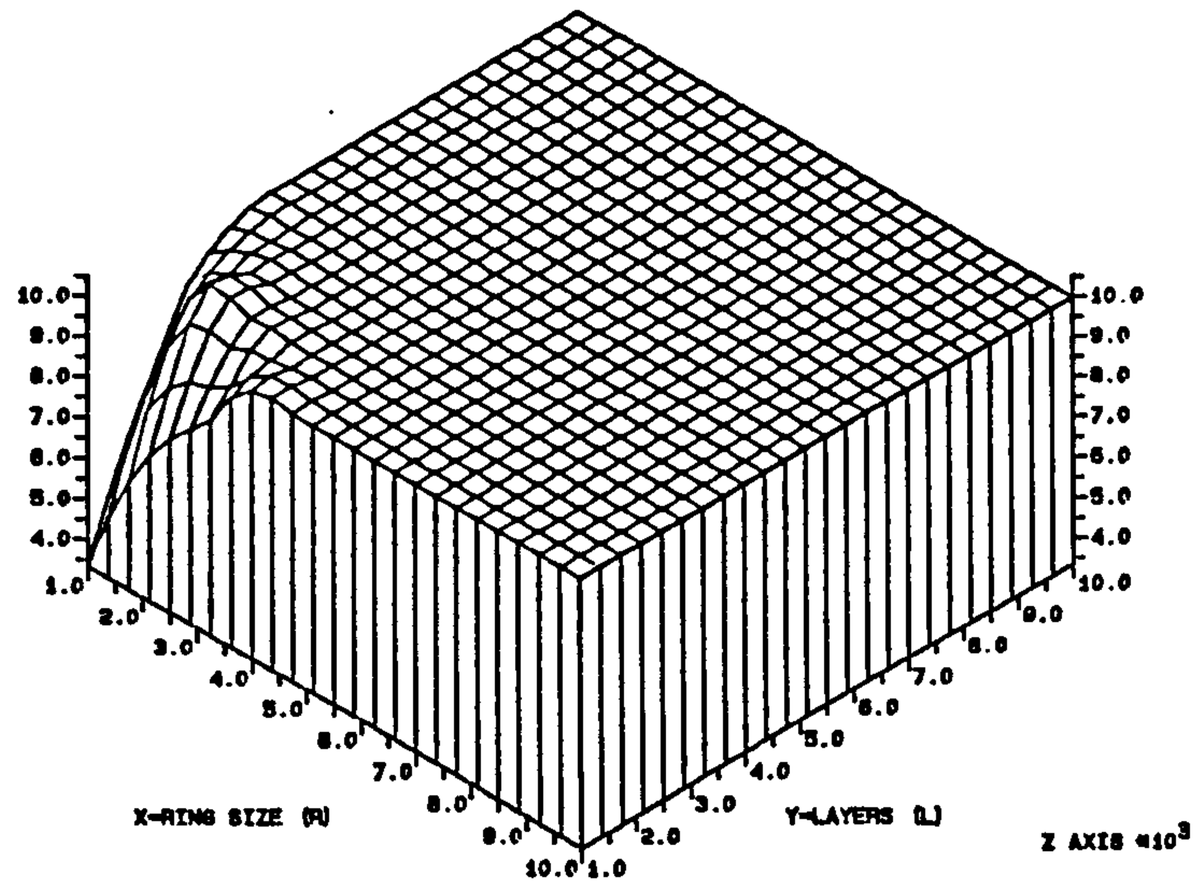
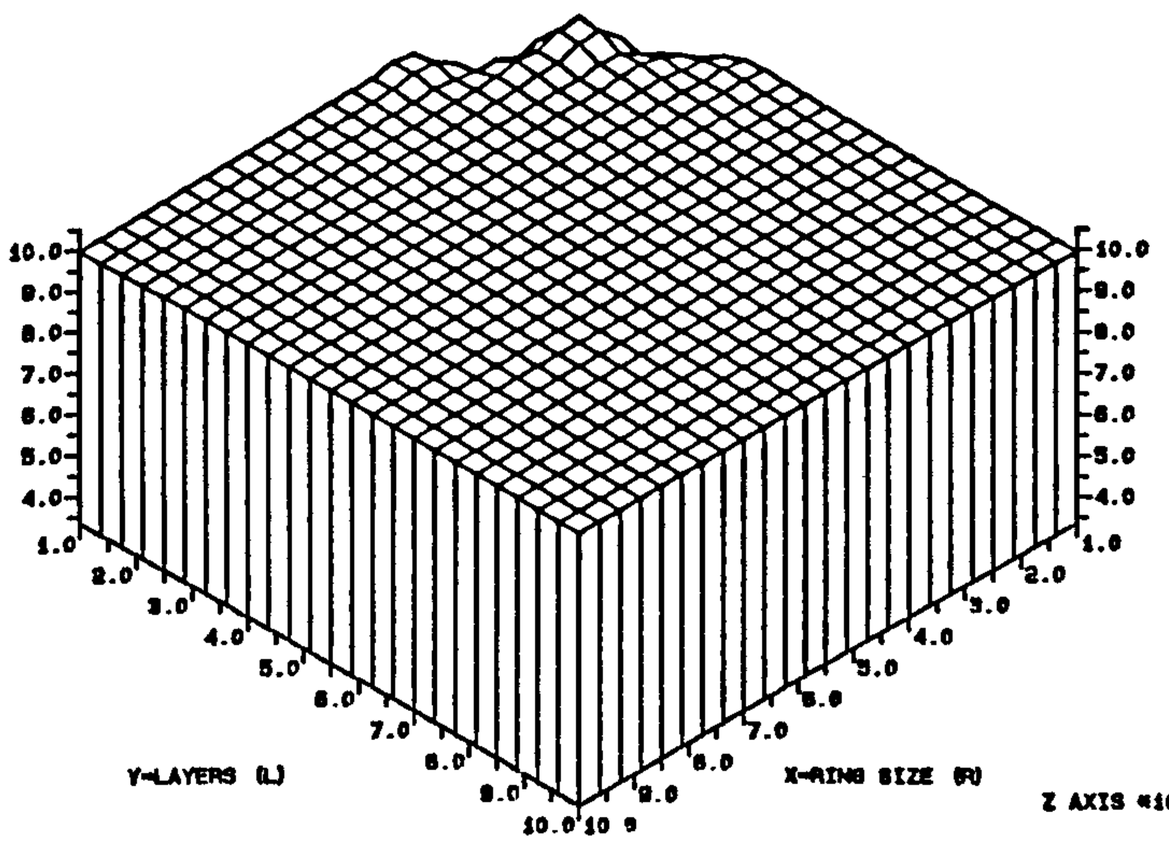
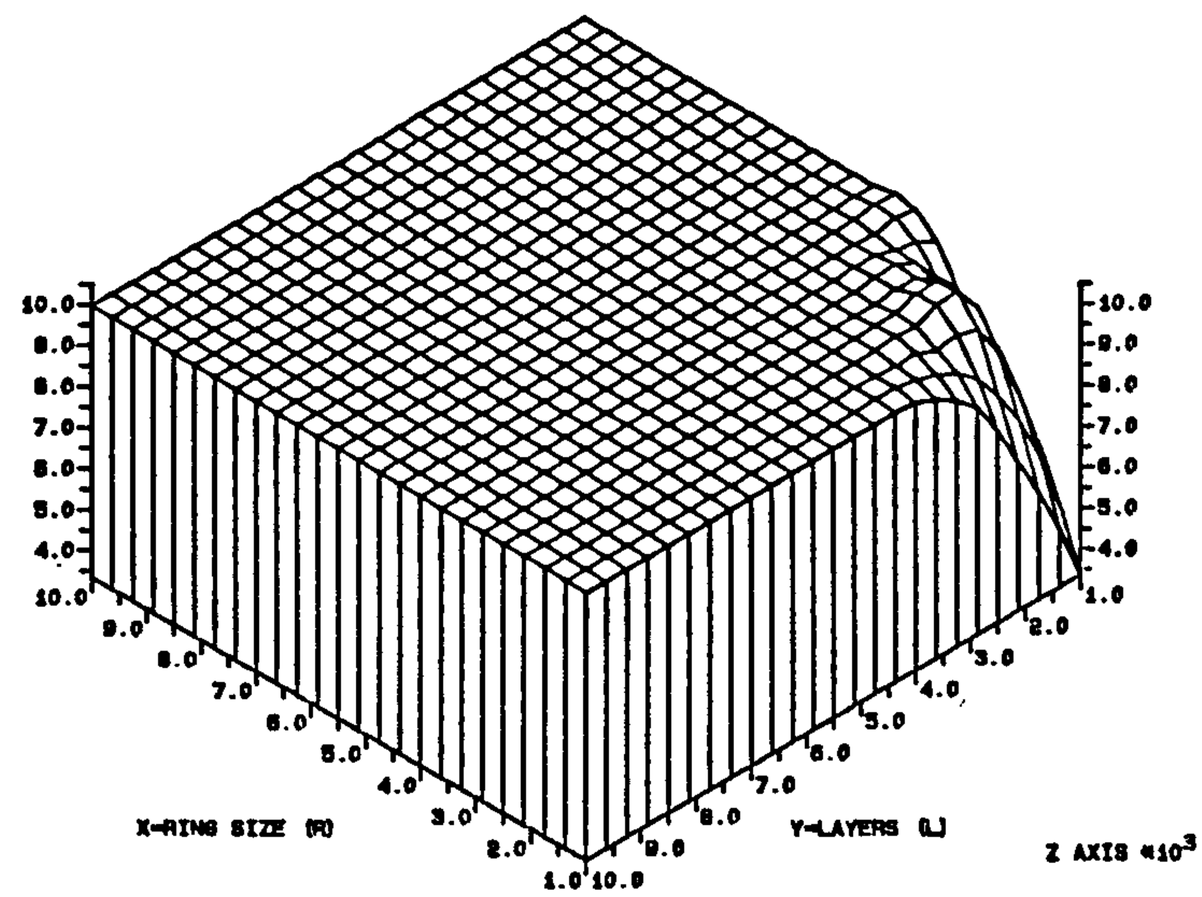
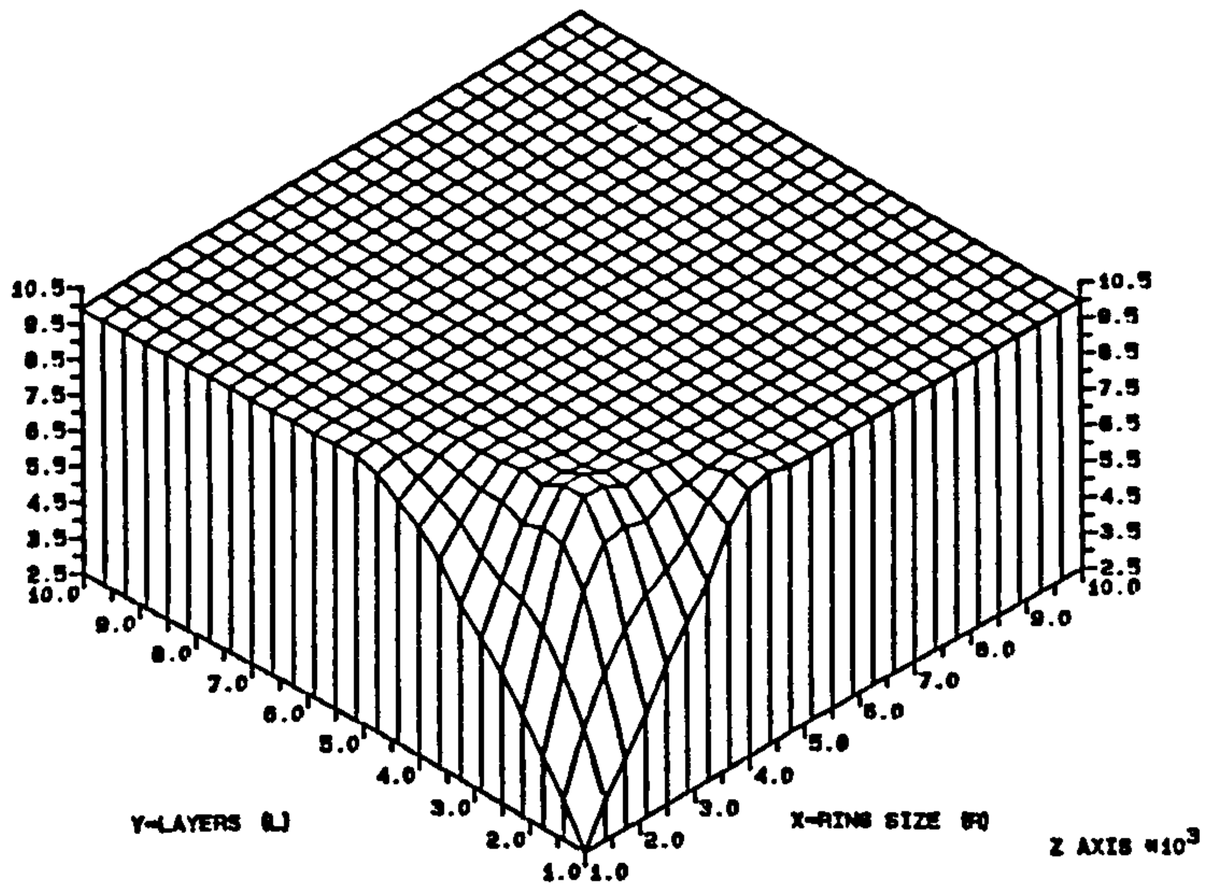


FIG. 4-24.1 HOMOGENEOUS COMMS1 FED AT ONE POINT WITH DATA OF TYPE 10. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.







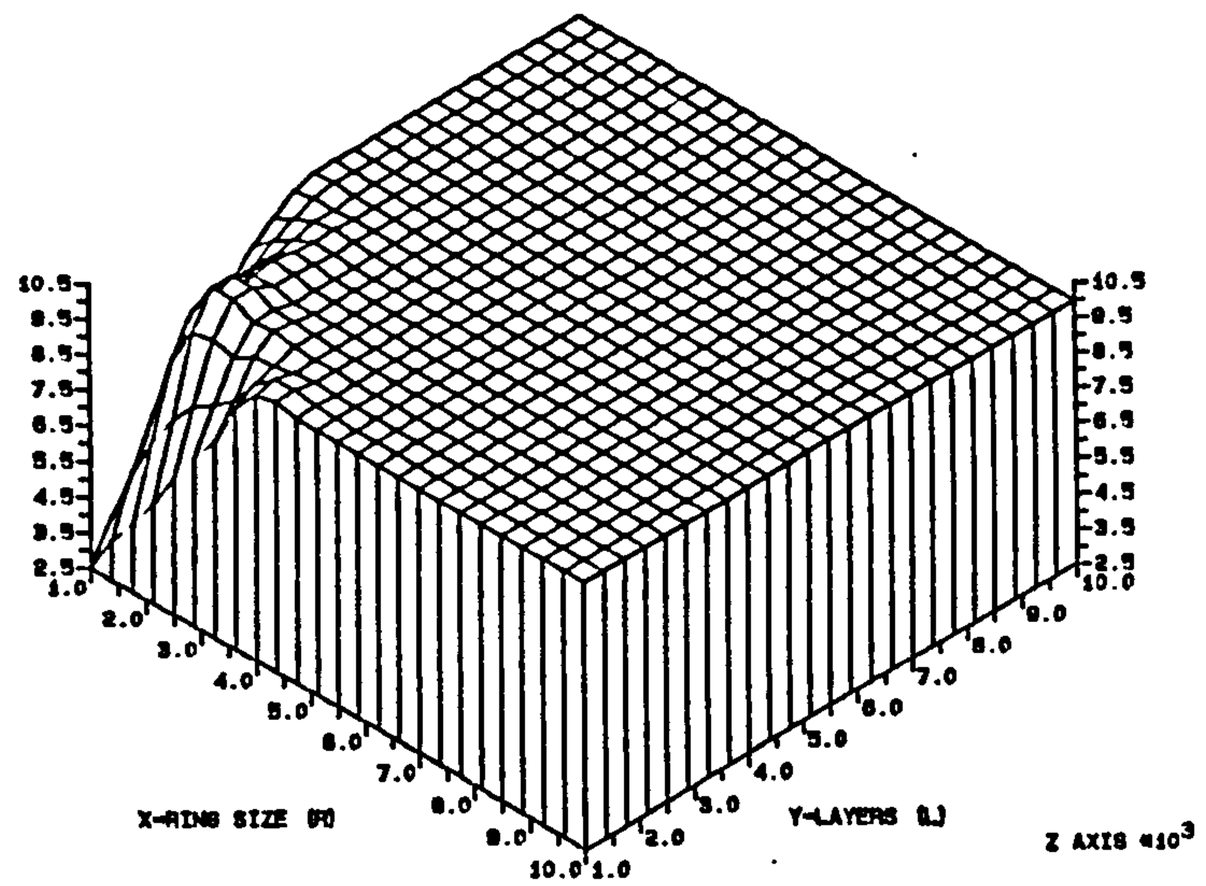
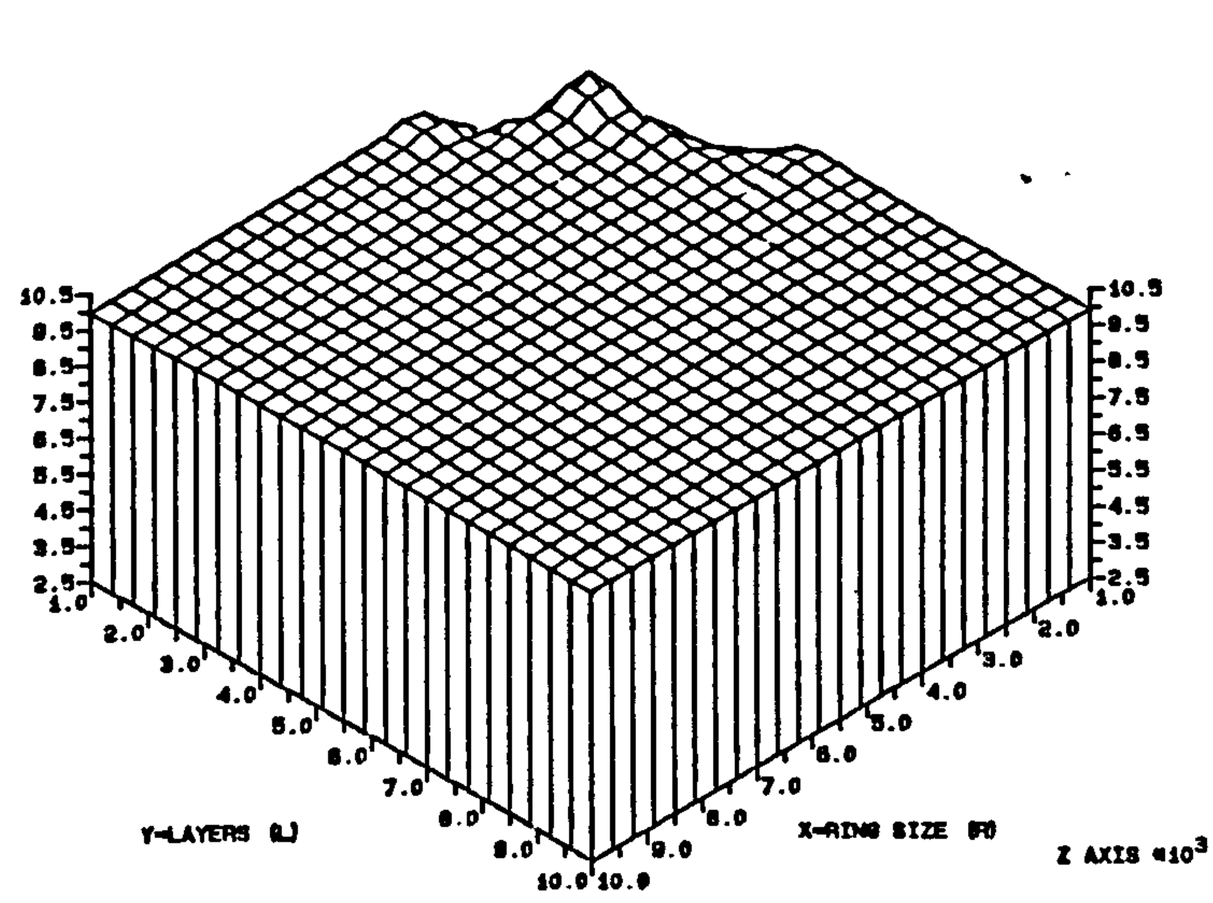
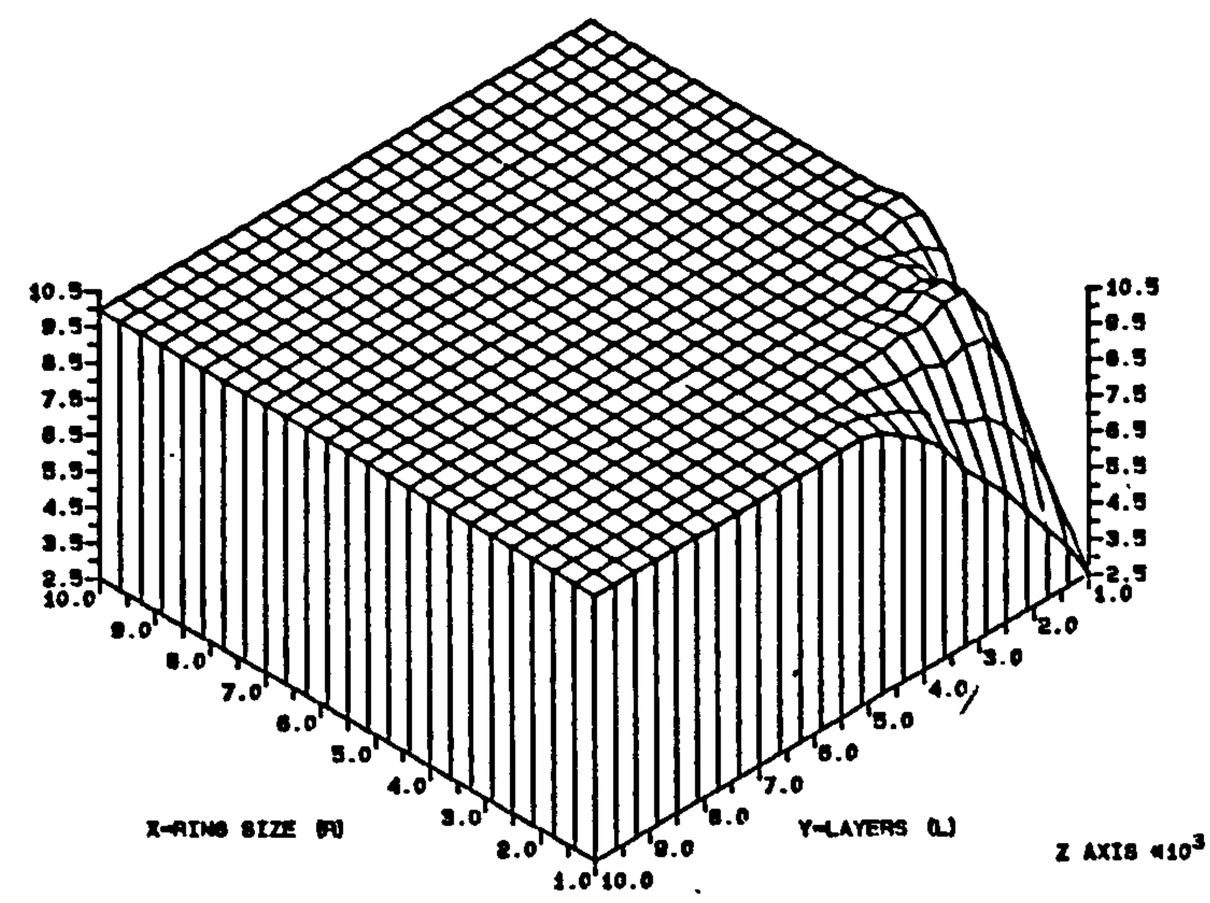


FIG. 4-24.2 HOMOGENEOUS COMMS2 FED AT ONE POINT WITH DATA OF TYPE 10. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





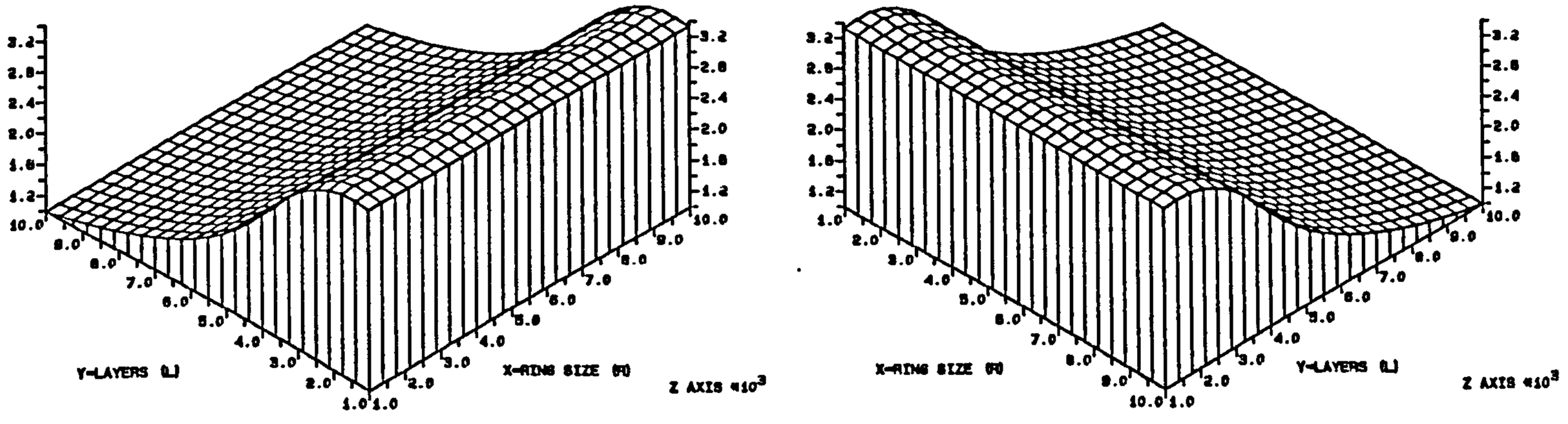
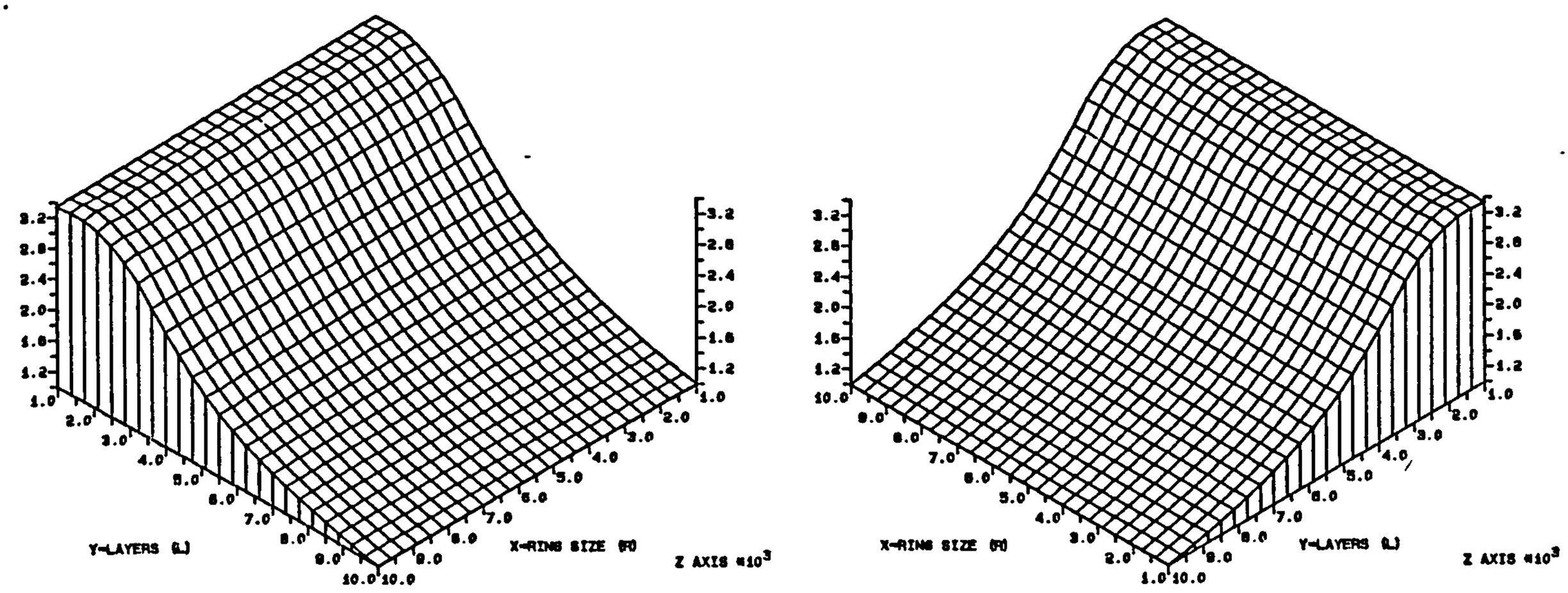


FIG. 4-25.1 HOMOGENEOUS COMMS1 FED AT ALL POINTS WITH DATA OF TYPE 10. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



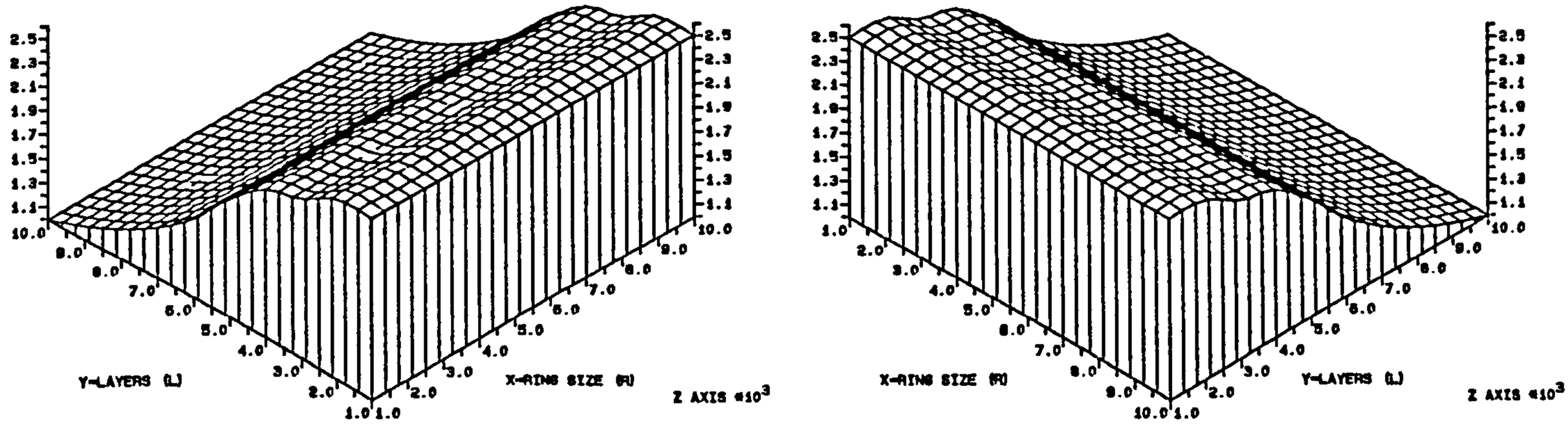
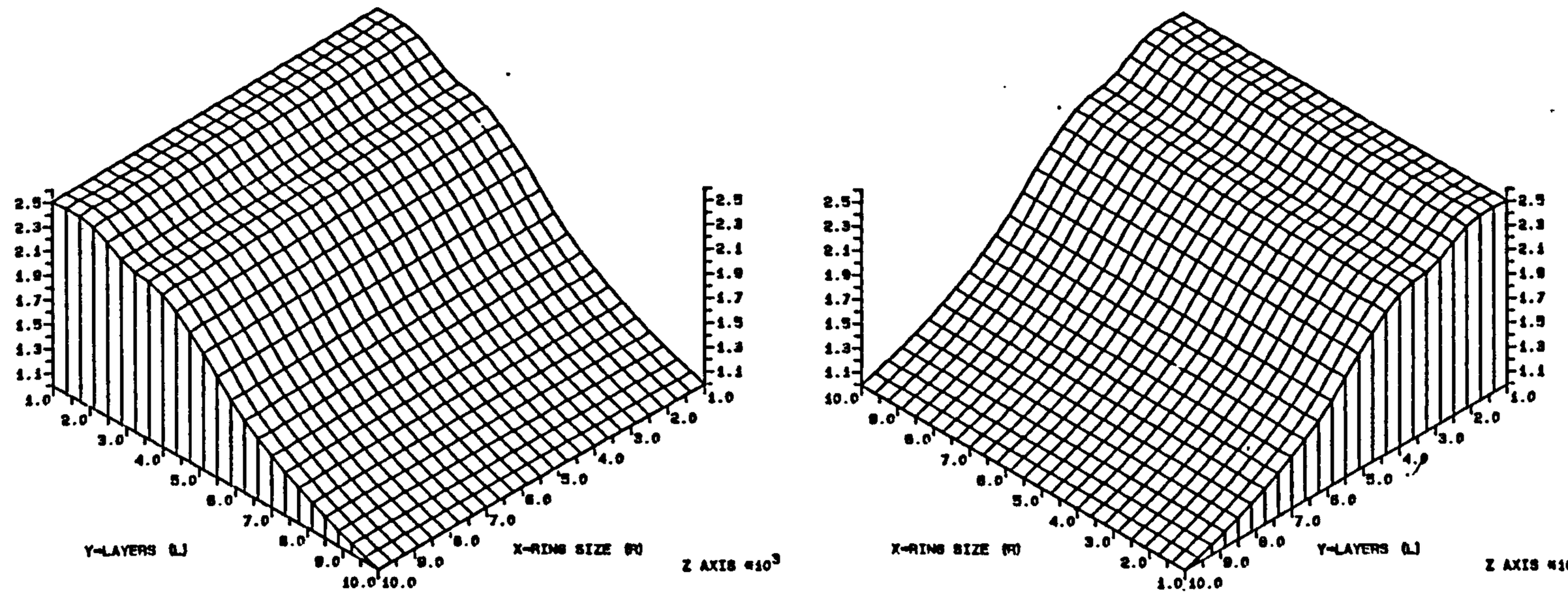


FIG. 4-25.2 HOMOGENEOUS COMMS2 FED AT ALL POINTS WITH DATA OF TYPE 10. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



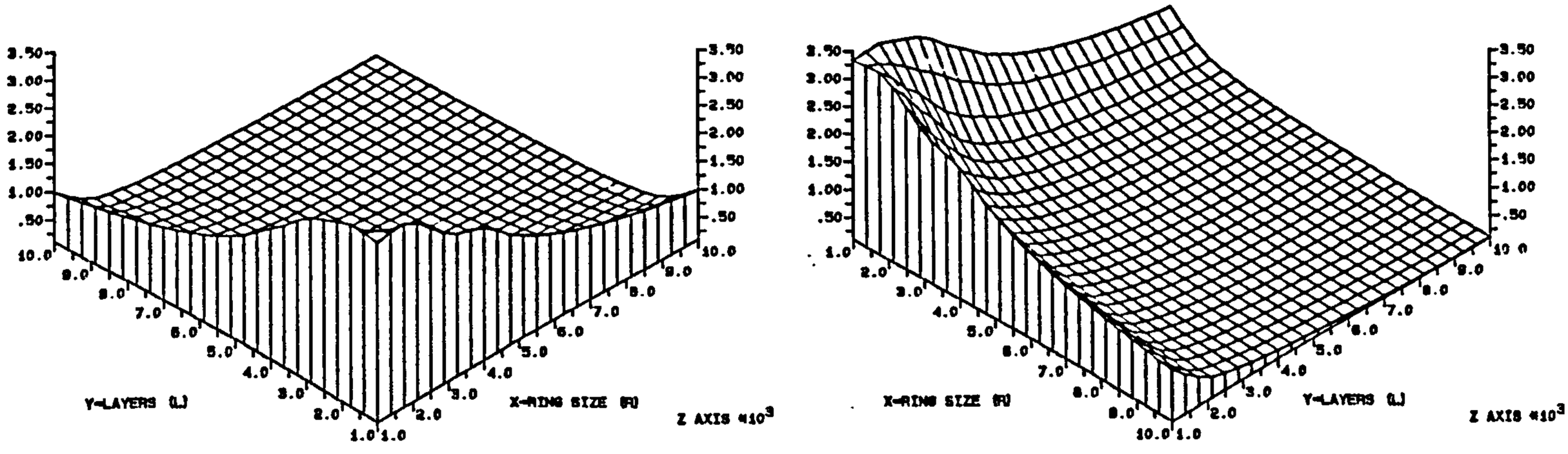
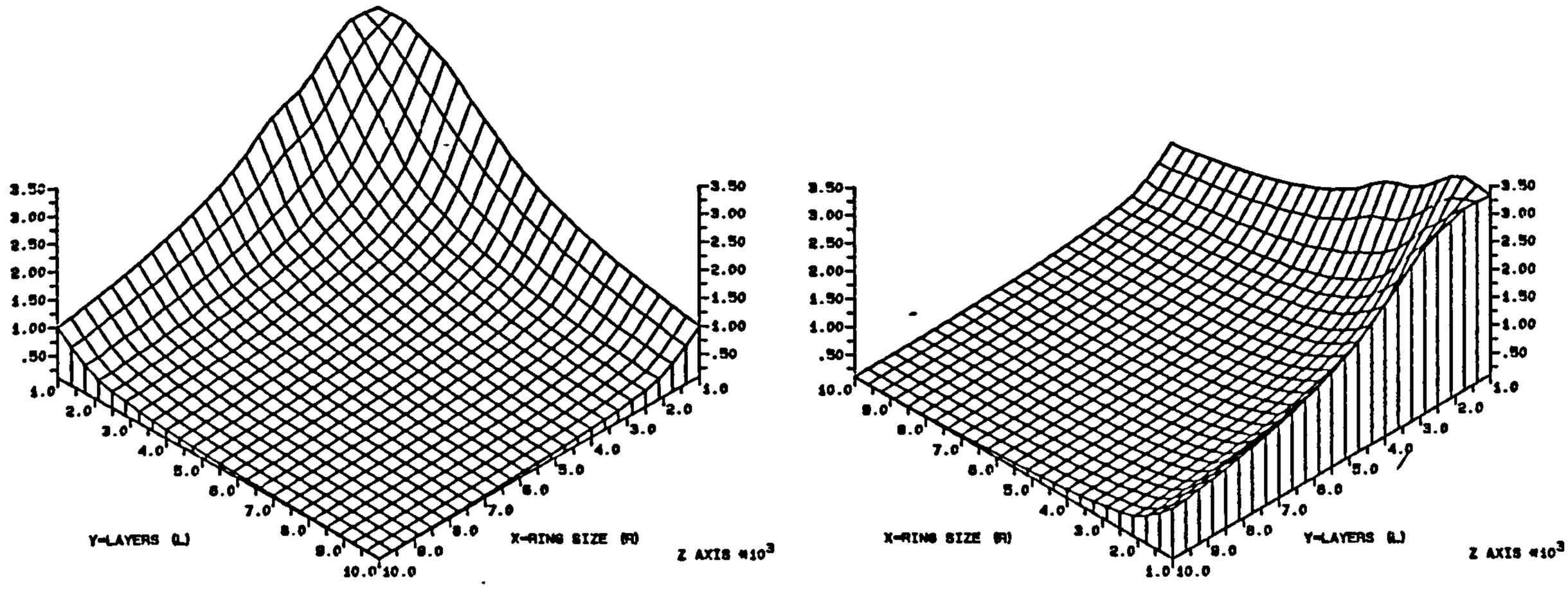


FIG. 4-26.1 HOMOGENEOUS COMMS1 FED AT ONE POINT WITH DATA OF TYPE 10. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



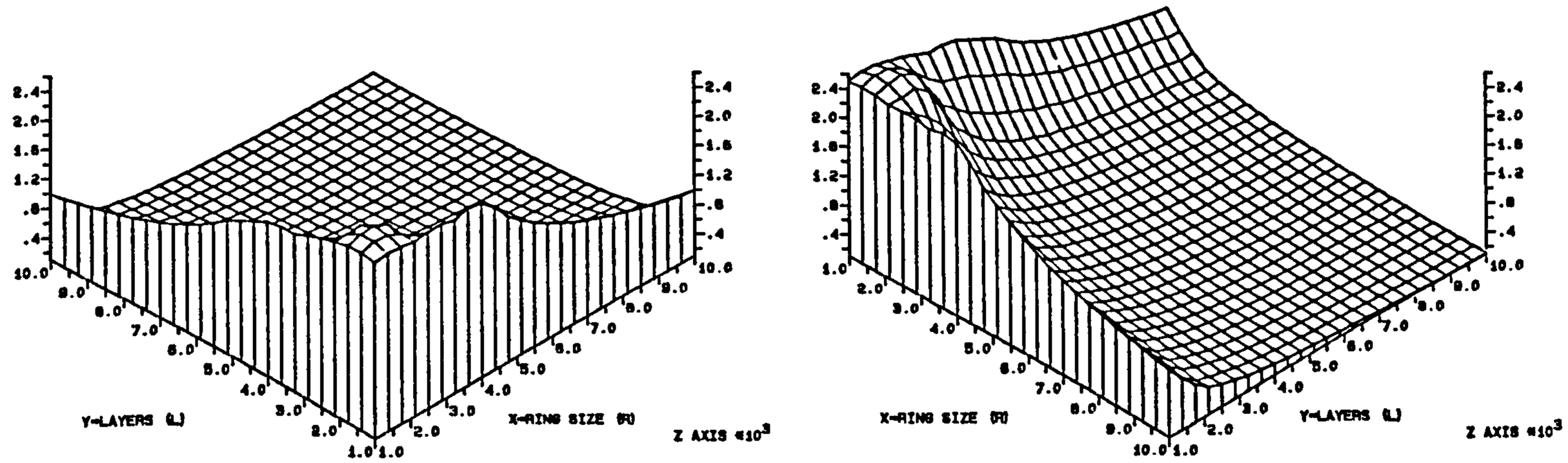
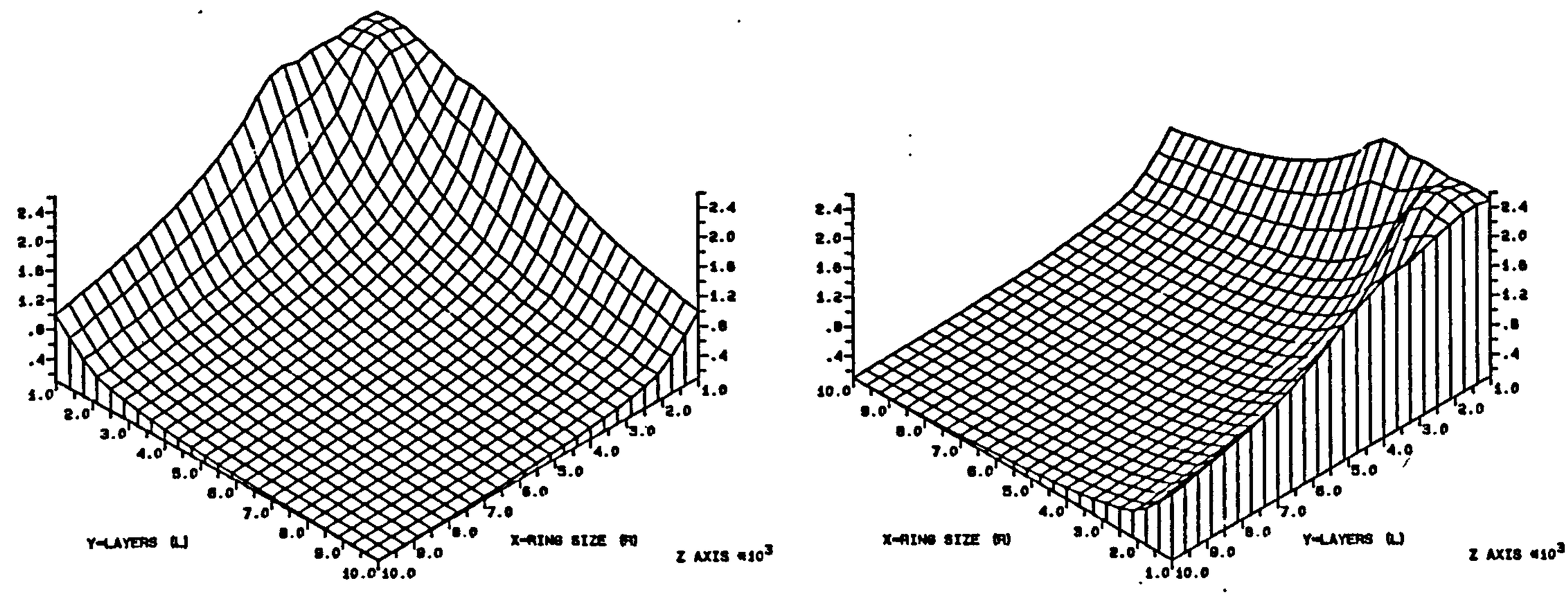
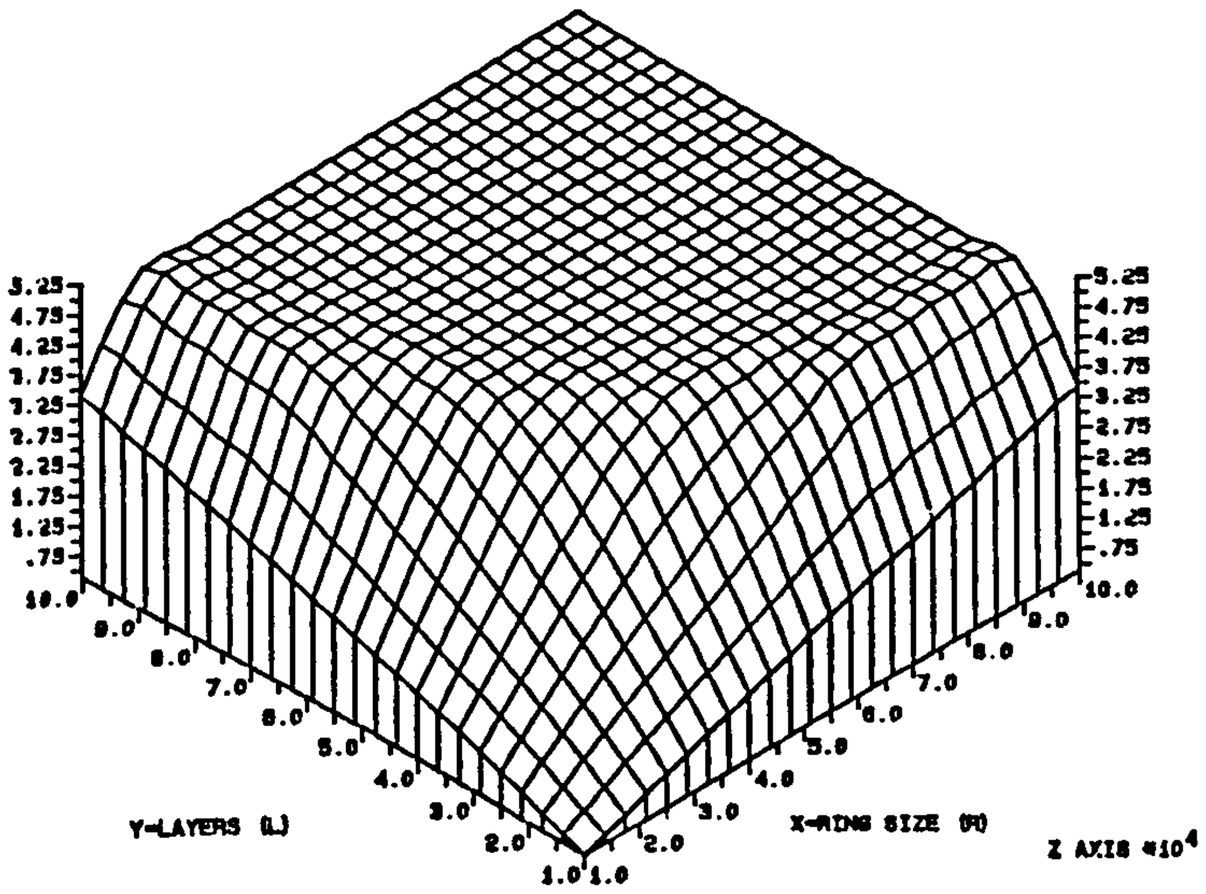


FIG. 4-26.2 HOMOGENEOUS COMMS2 FED AT ONE POINT WITH DATA OF TYPE 10. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





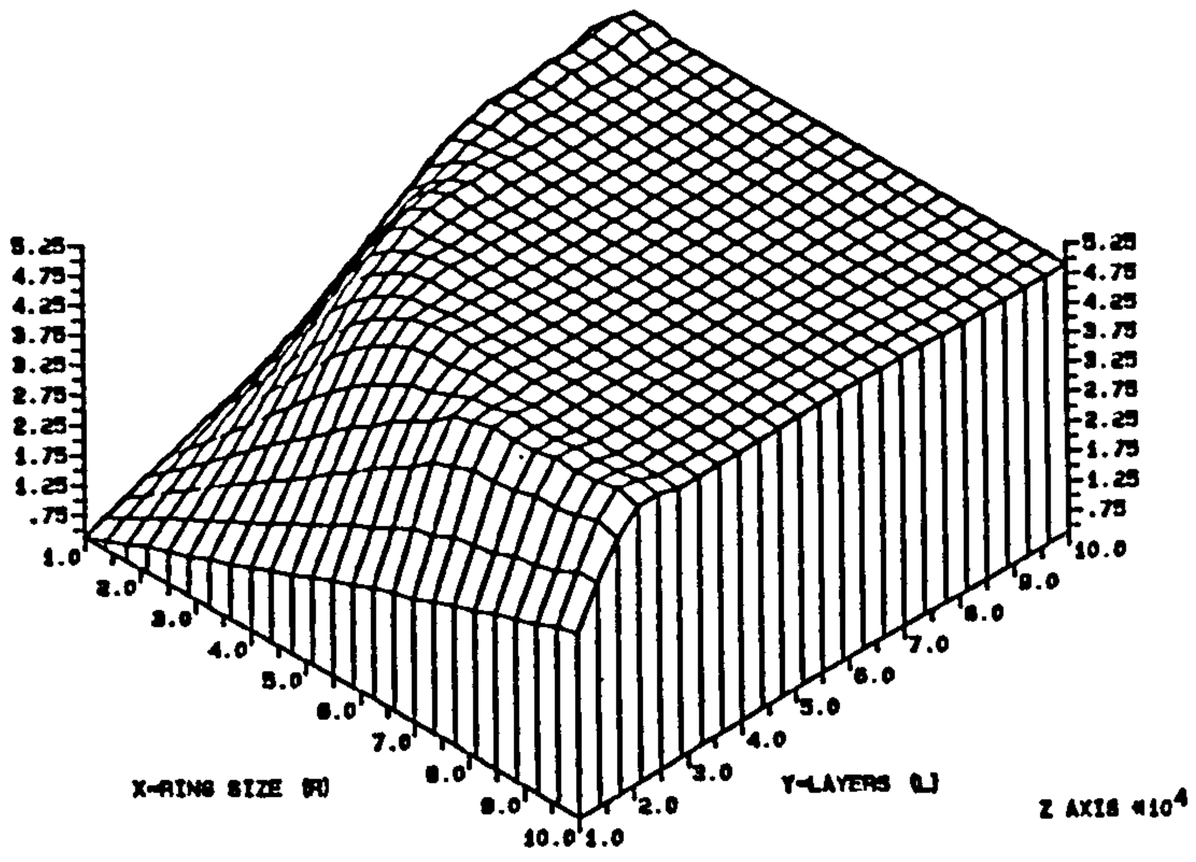
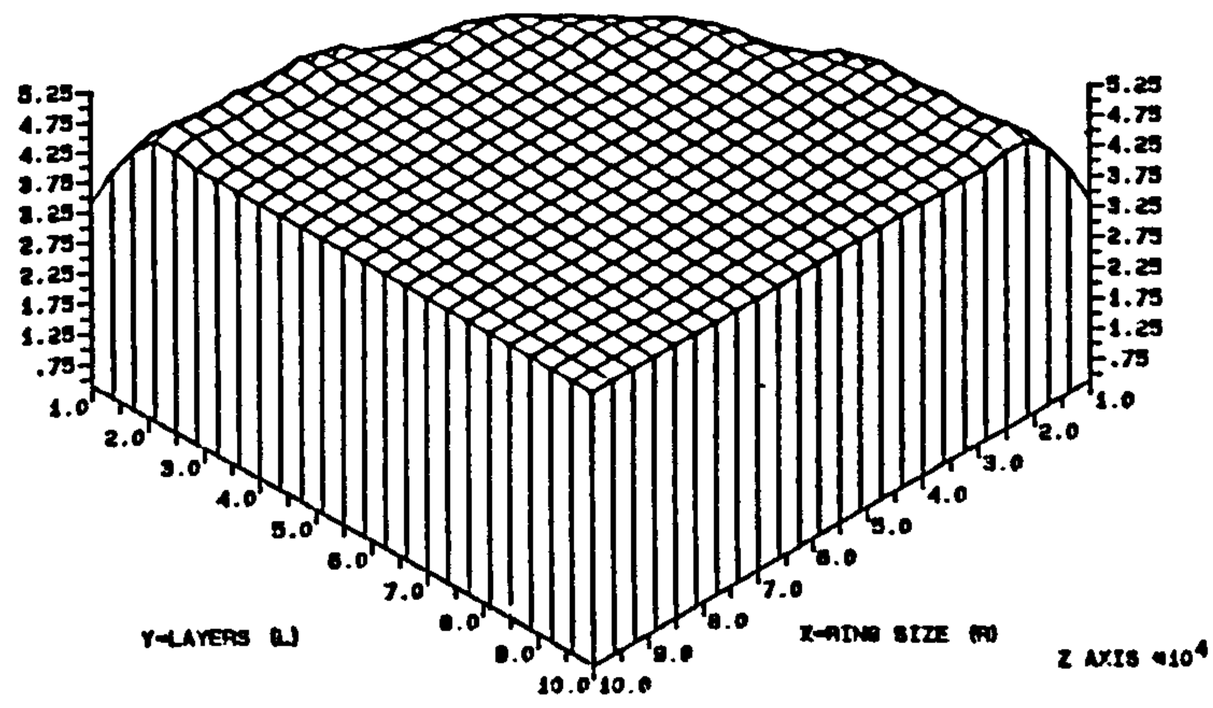
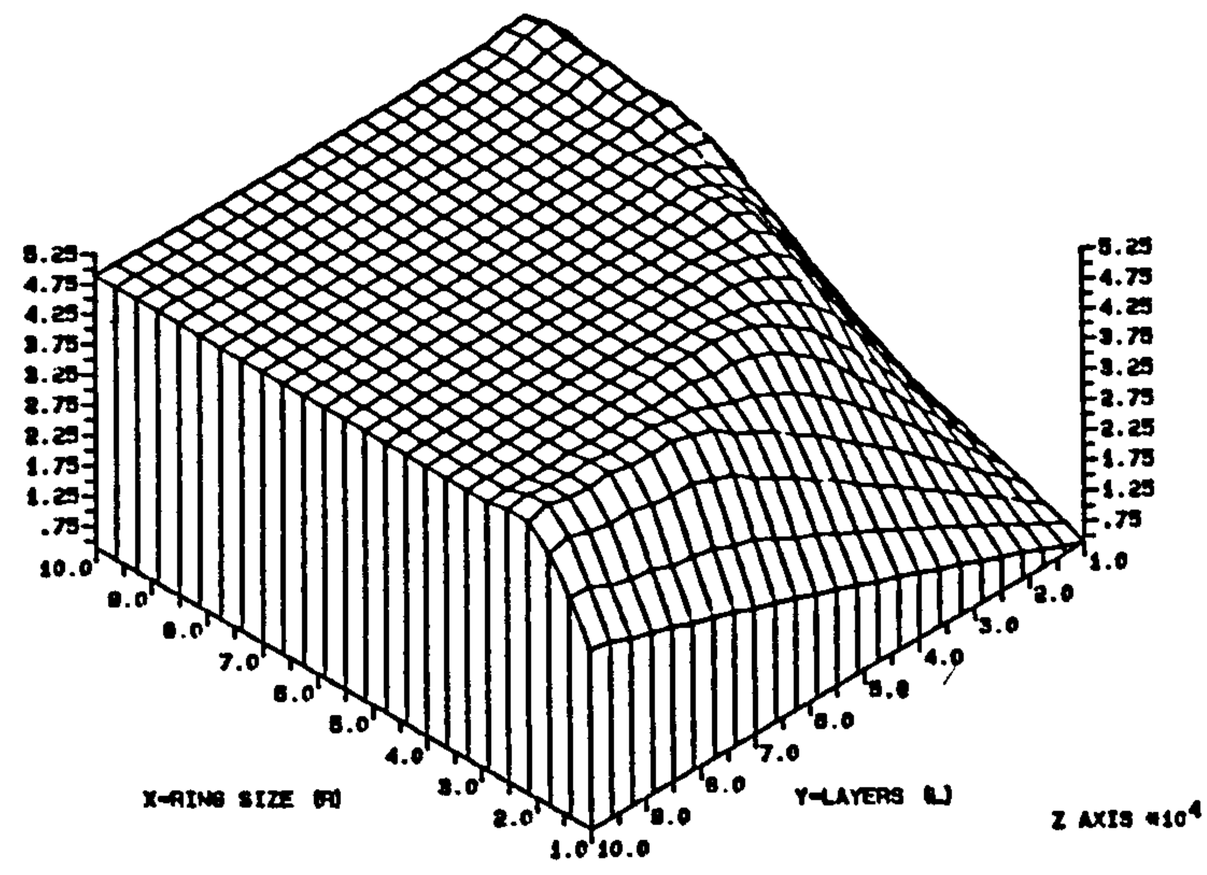
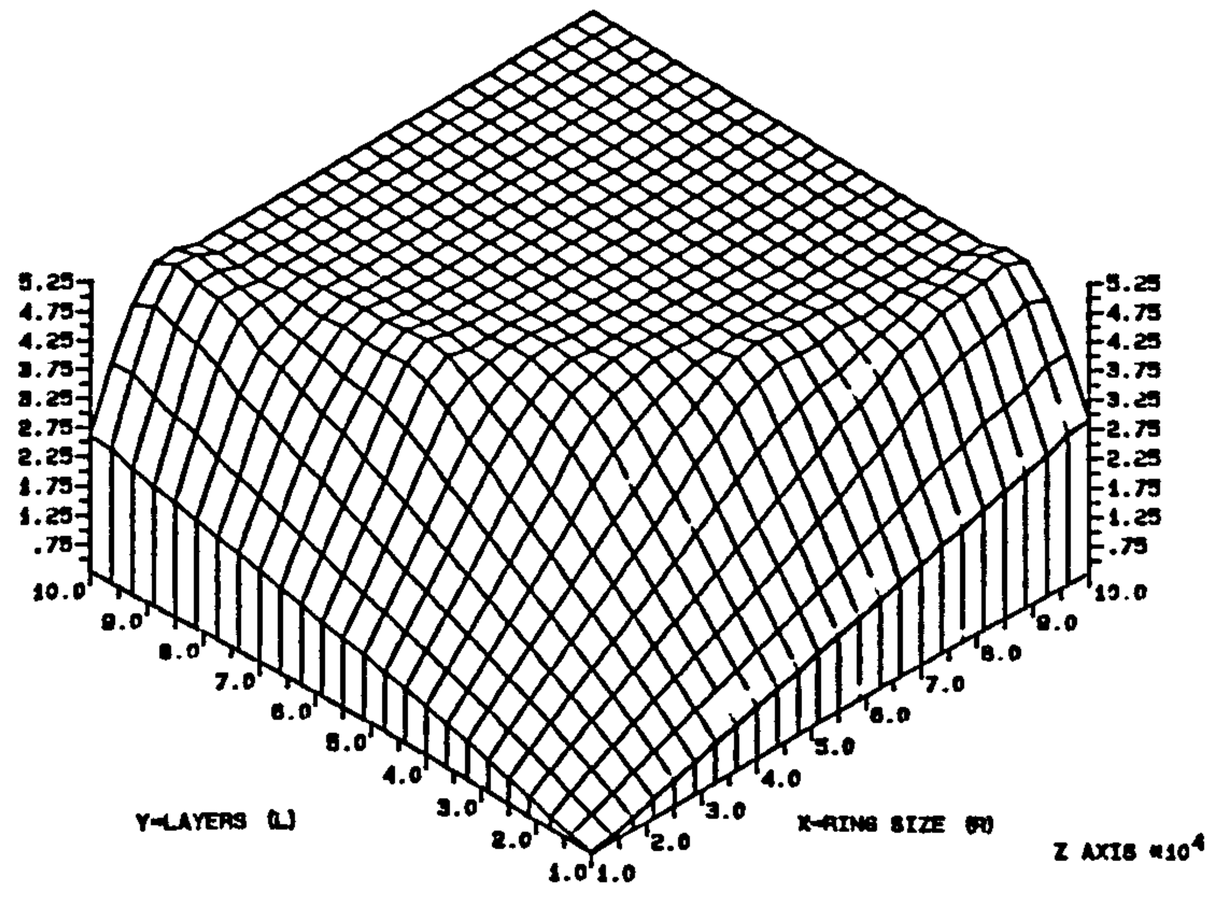


FIG. 4-27.1 HOMOGENEOUS COMMS1 FED AT ONE POINT WITH DATA OF TYPE 50. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.







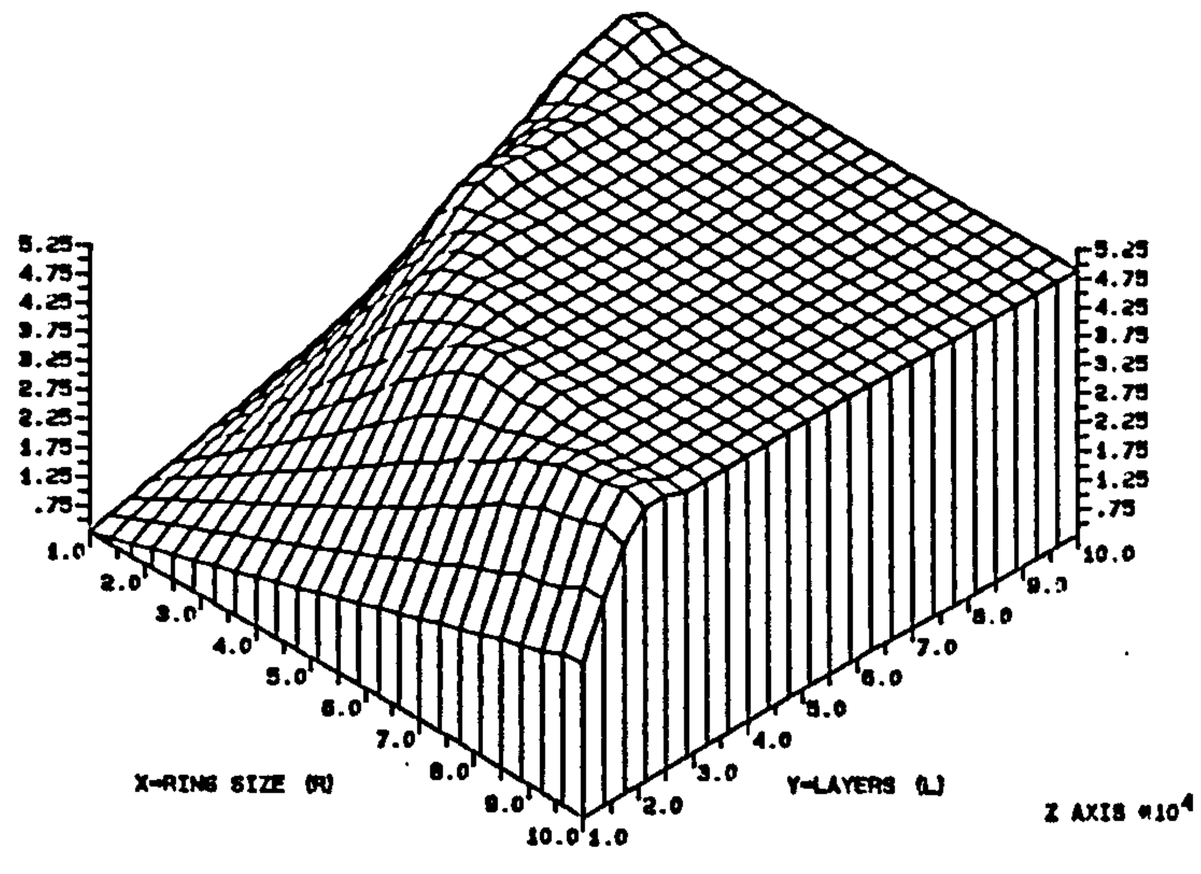
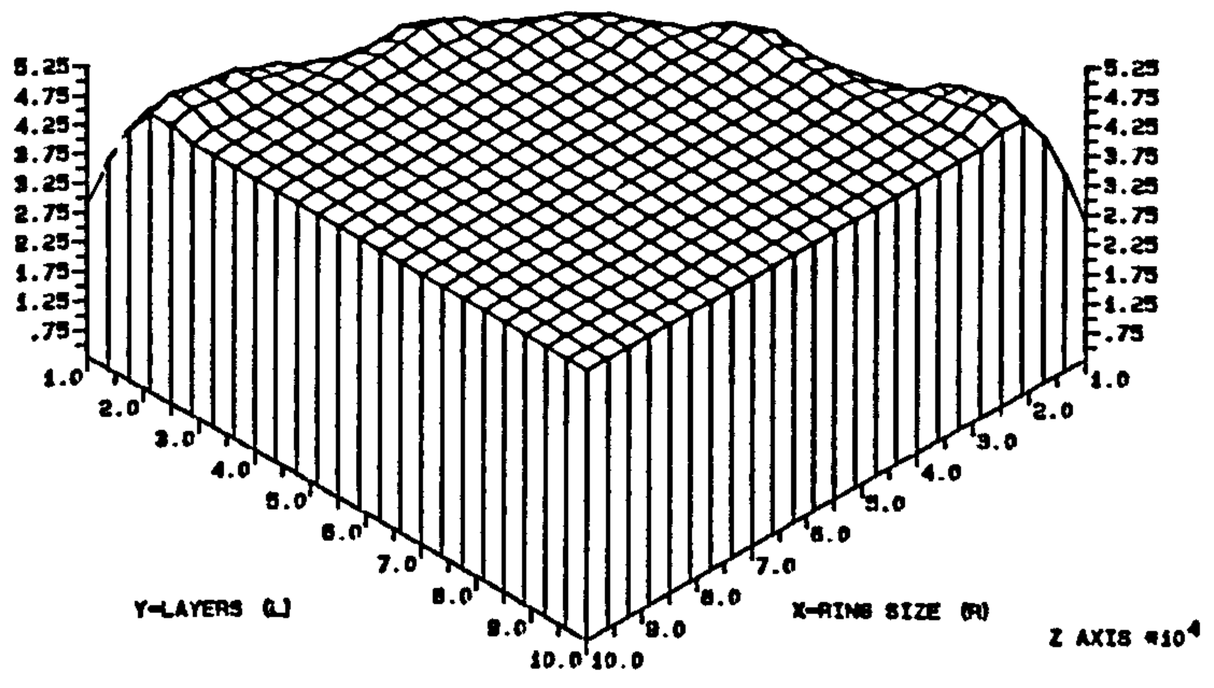
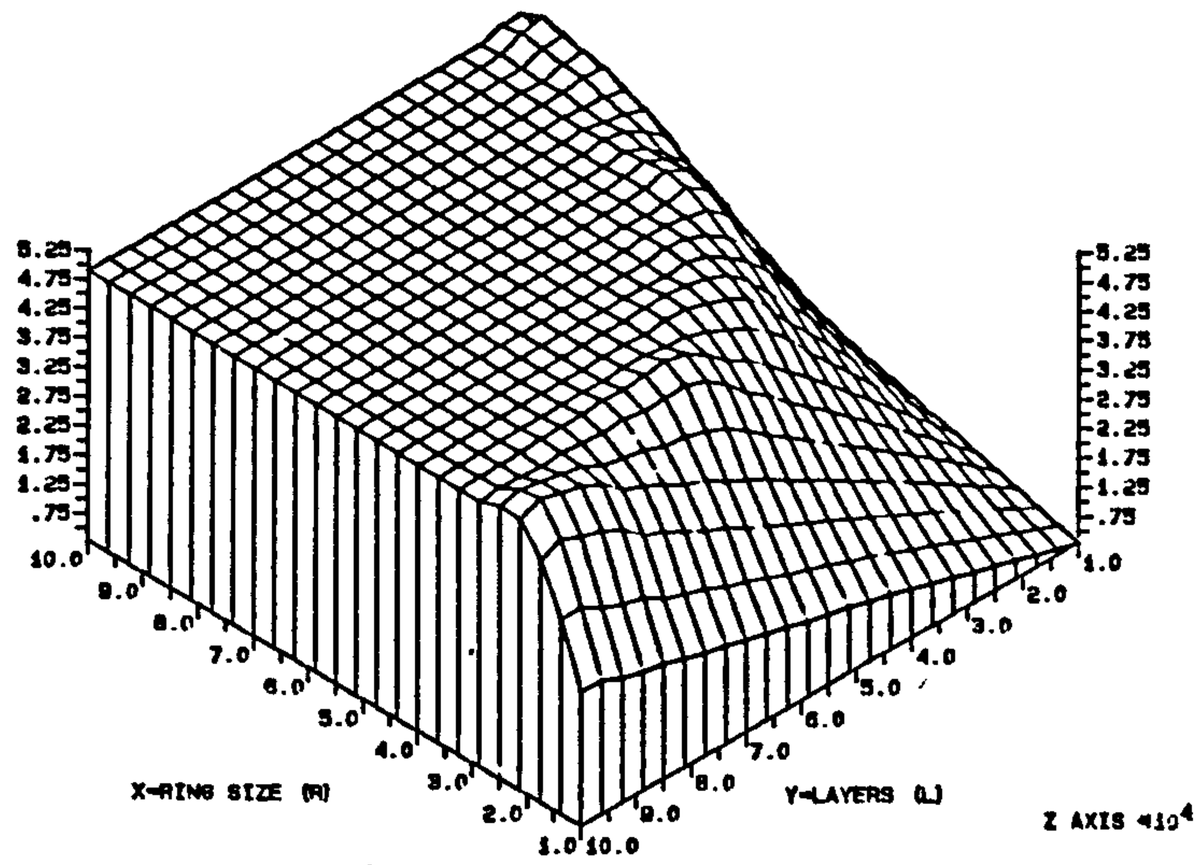


FIG. 4-27.2 HOMOGENEOUS COMMS2 FED AT ONE POINT WITH DATA OF TYPE 50. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





4.12.2.2.2 Fault Tolerance

One of the expected advantages of the homogeneous cylinder, as with the ring but with a greater path redundancy, was that of fault tolerance. It should be noted that in contrast to fault tolerance achieved through the system patching itself up following a fault[4,15,53,14] the system carries on regardless with a reduced throughput. A discussion of the possible failures and the preventative mechanisms and methods required are discussed in chapter 3, section 3.7. An abstract simulation of failure of processors within the system was carried out using the simulation model. A facility was provided of specifying at the commencement of a simulation run whether any processors had failed and if so, which ones. Failed processors were simply simulated as 'doing nothing gracefully', it was assumed that either some suitable fault detection was incorporated within the processing nodes to cope with the source, sink and faulty processing scenarios described in section 3.7 or that such conditions were highly unlikely.

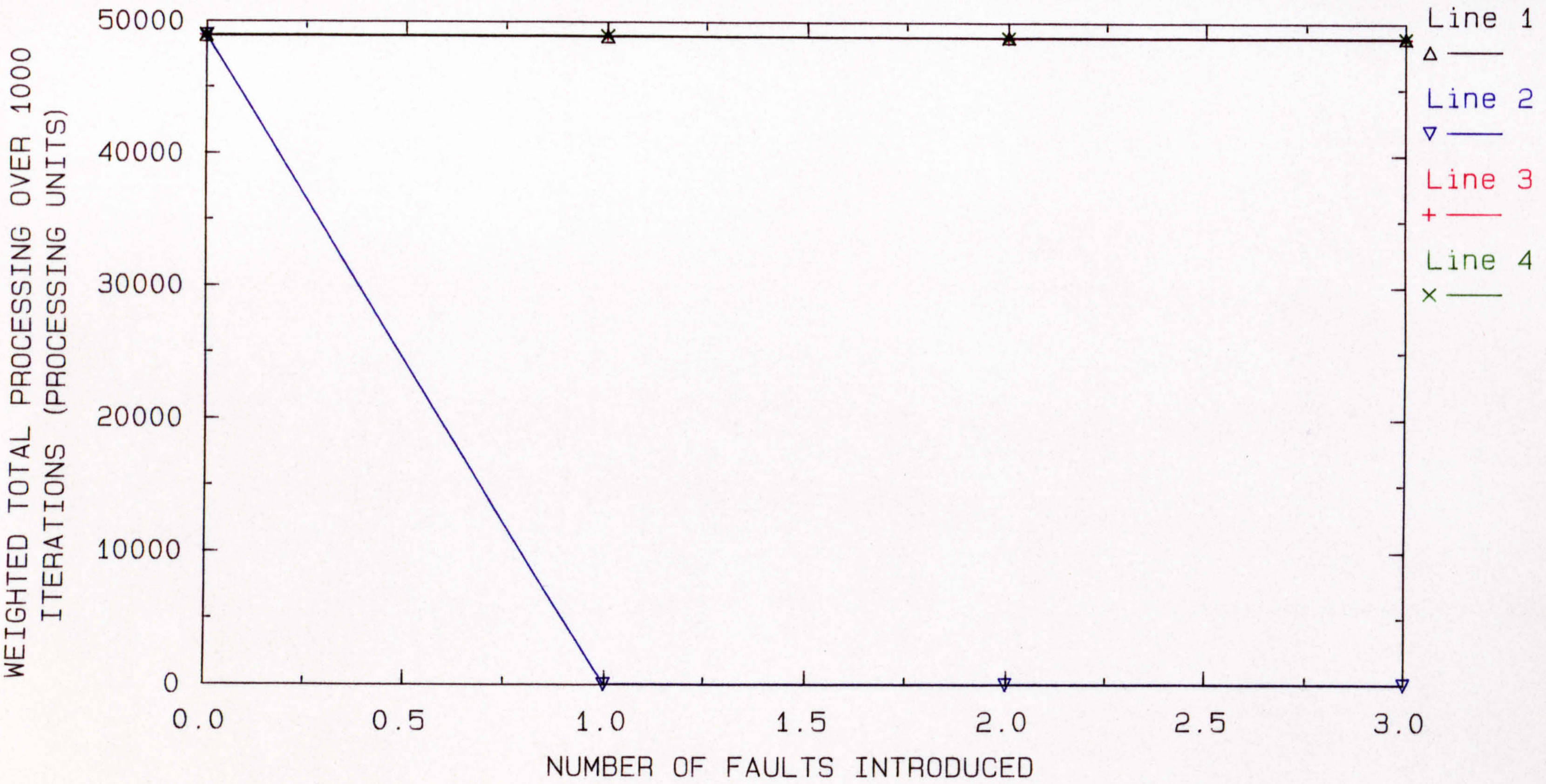
Two principle cases were studied using the simulation model, the first was a cylinder of processors with R=L=10 fed with data of type 50 at only one of the nodes of the top layer and the second case was a similar cylinder again with R=L=10 fed with data of type 50 but at all of the nodes of the top layer.

Figs 4.28–4.31 show the processing achieved by a cylinder of processors using the four communication algorithms when up to 3 randomly chosen faults were introduced. Four runs were carried out for each algorithm, each using a different seed for

the random number generator used to determine the faulty nodes.

The graphs show clearly the effect of the failure of the node to which the data is fed in line 2. The other lines however show a negligible change in processing despite the introduction of faults. Bearing in mind that only one node is fed with data and that this will restrict the number of nodes usefully applied, some of the faults introduced will cause an already idle processor to fail with no consequent effect on the total processing.

FIG. 4-28 HOMOGENEOUS COMMS1 WITH R=10, L=10 FED FROM ONE POINT WITH DATA OF TYPE 50.



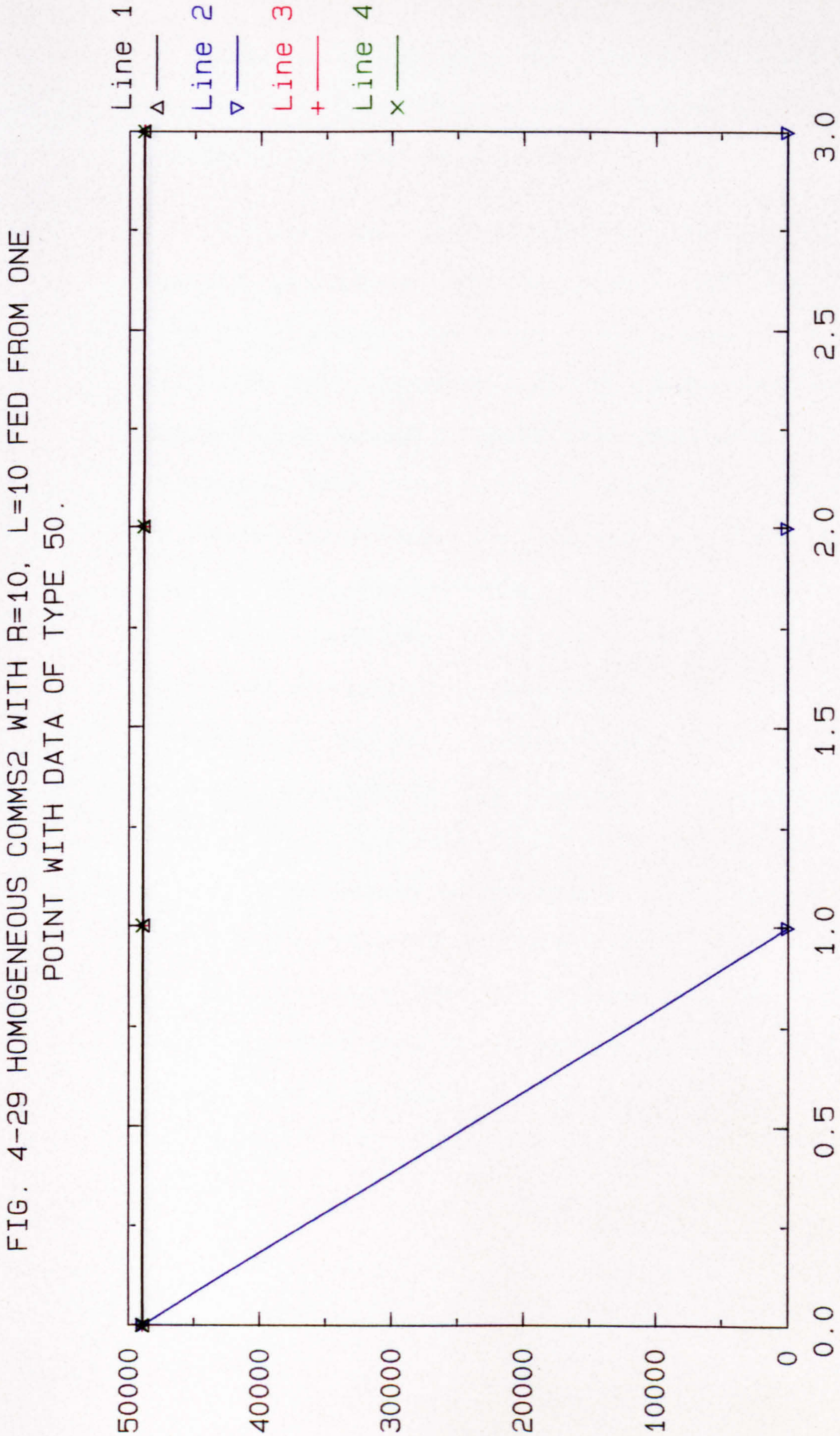

Line 1
Line 2
Line 3
Line 4
4-29 HOMOGENEOUS COMMS2 WITH R=10, L=10 FED FROM ONE POINT WITH DATA OF TYPE 50.
NUMBER OF FAULTS INTRODUCED
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
10000
20000
30000
40000
50000
WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS (PROCESSING UNITS)

FIG. 4

FIG. 4-30 HOMOGENEOUS COMMS3 WITH R=10, L=10 FED FROM ONE POINT WITH DATA OF TYPE 50.

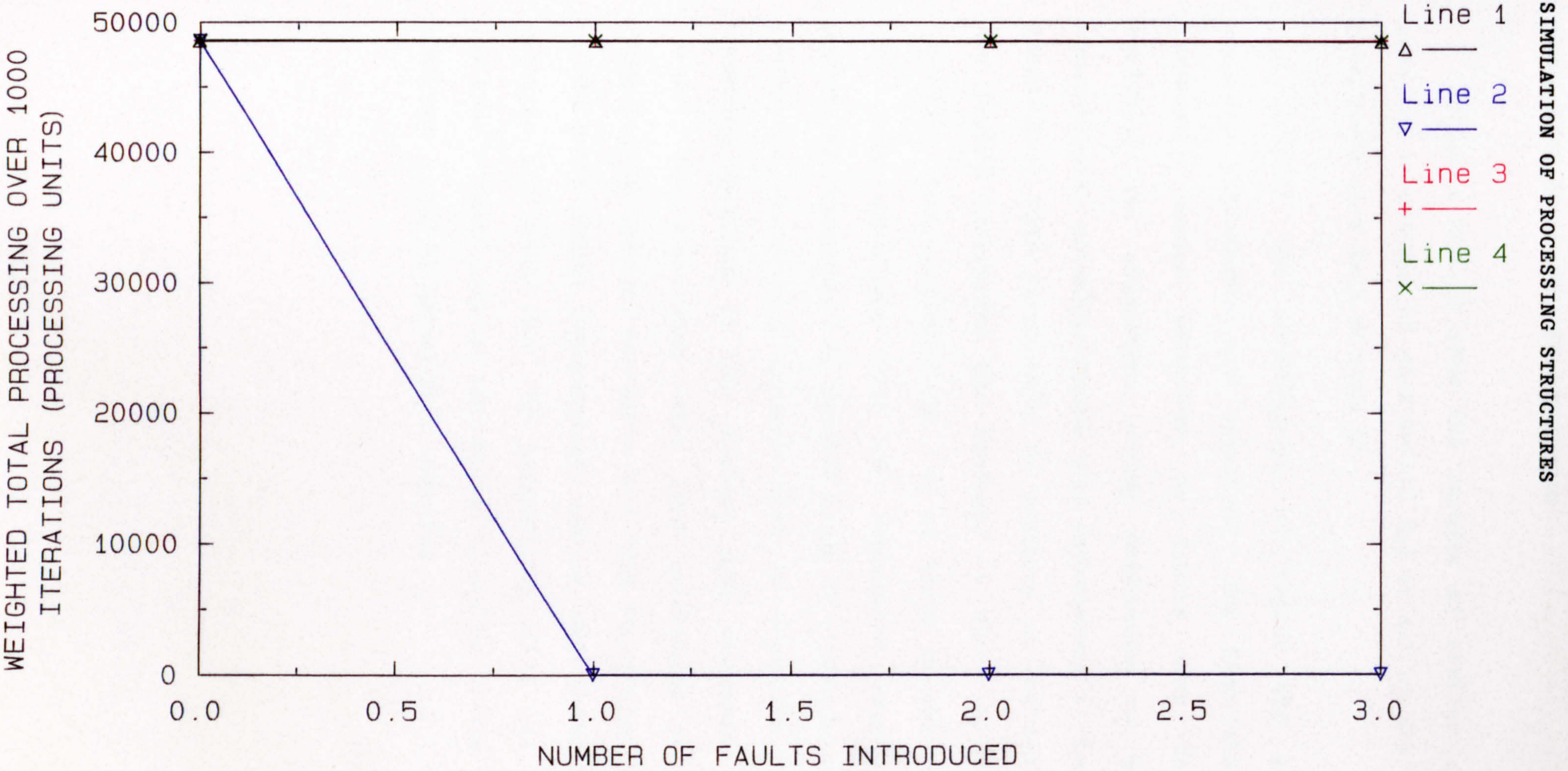
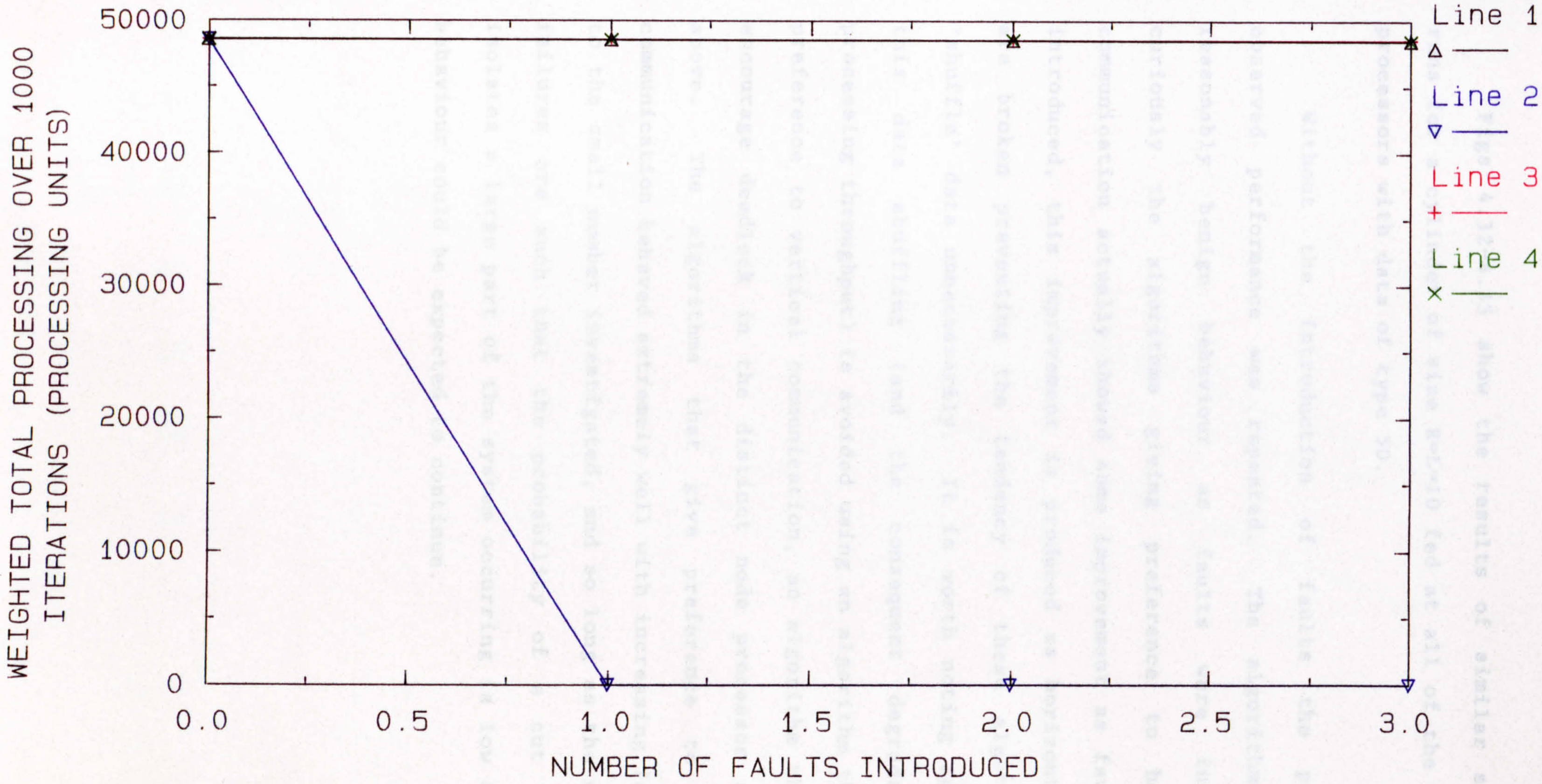


FIG. 4-31 HOMOGENEOUS COMMS4 WITH R=10, L=10 FED FROM ONE POINT WITH DATA OF TYPE 50.



Figs 4.32-4.35 show the results of similar simulation runs for a cylinder of size R=L=10 fed at all of the uppermost processors with data of type 50.

Without the introduction of faults the previously observed performance was repeated. The algorithms showed reasonably benign behaviour as faults were introduced, curiously the algorithms giving preference to horizontal communication actually showed some improvement as faults were introduced, this improvement is produced as horizontal rings are broken preventing the tendency of these algorithms to 'shuffle' data unnecessarily. It is worth noting again that this data shuffling (and the consequent degradation of processing throughput) is avoided using an algorithm that gives preference to vertical communication, an algorithm that would encourage deadlock in the distinct node processor simulated above. The algorithms that give preference to vertical communication behaved extremely well with increasing faults, up to the small number investigated, and so long as the number of failures are such that the probability of a cut set that isolates a large part of the system occurring is low then this behaviour could be expected to continue.

FIG. 4-32 HOMOGENEOUS COMMS1 WITH R=10, L=10 FED FROM ALL POINTS WITH DATA OF TYPE 50.

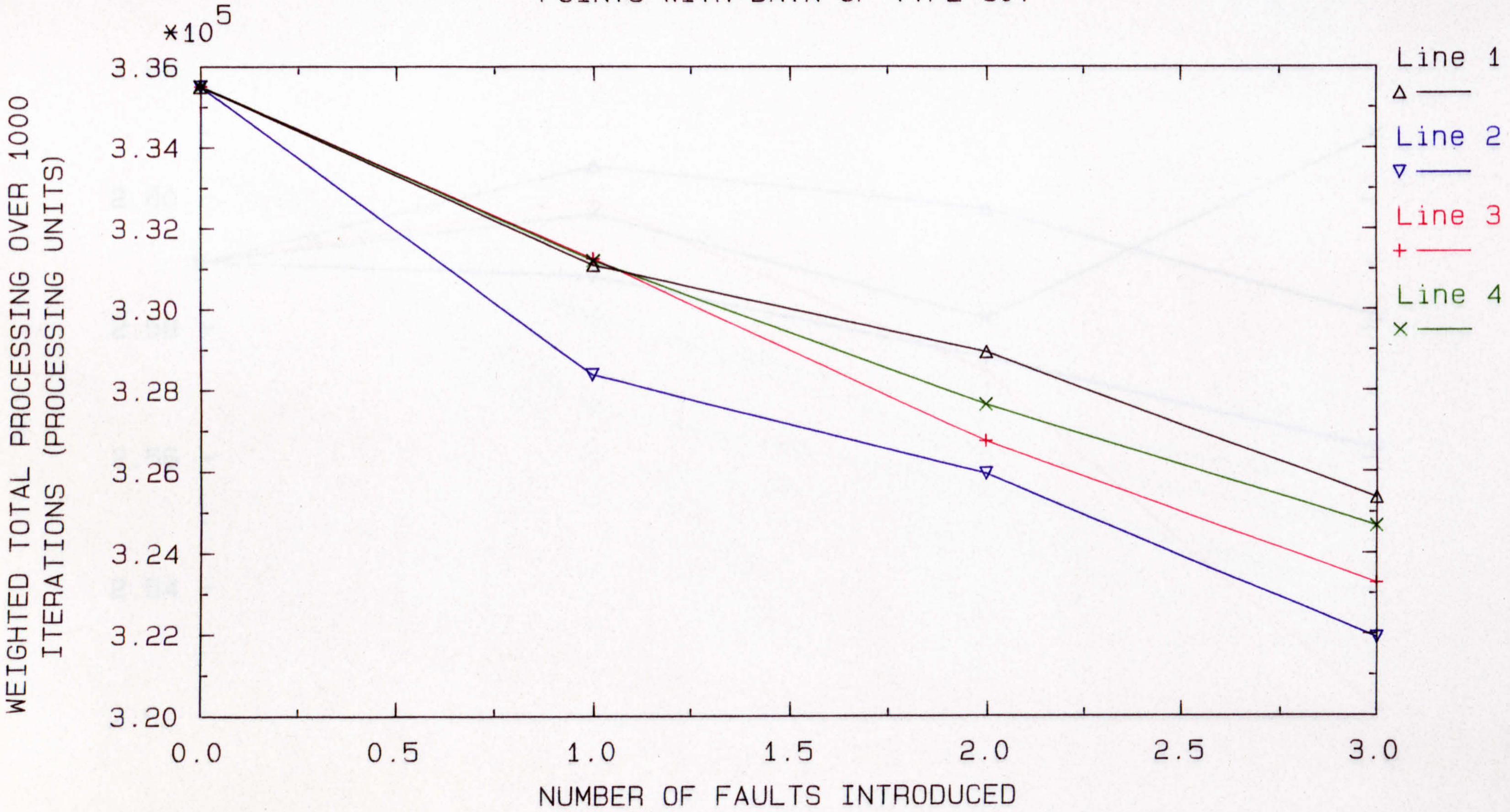


FIG. 4-33 HOMOGENEOUS COMMS2 WITH R=10, L=10 FED FROM ALL POINTS WITH DATA OF TYPE 50.

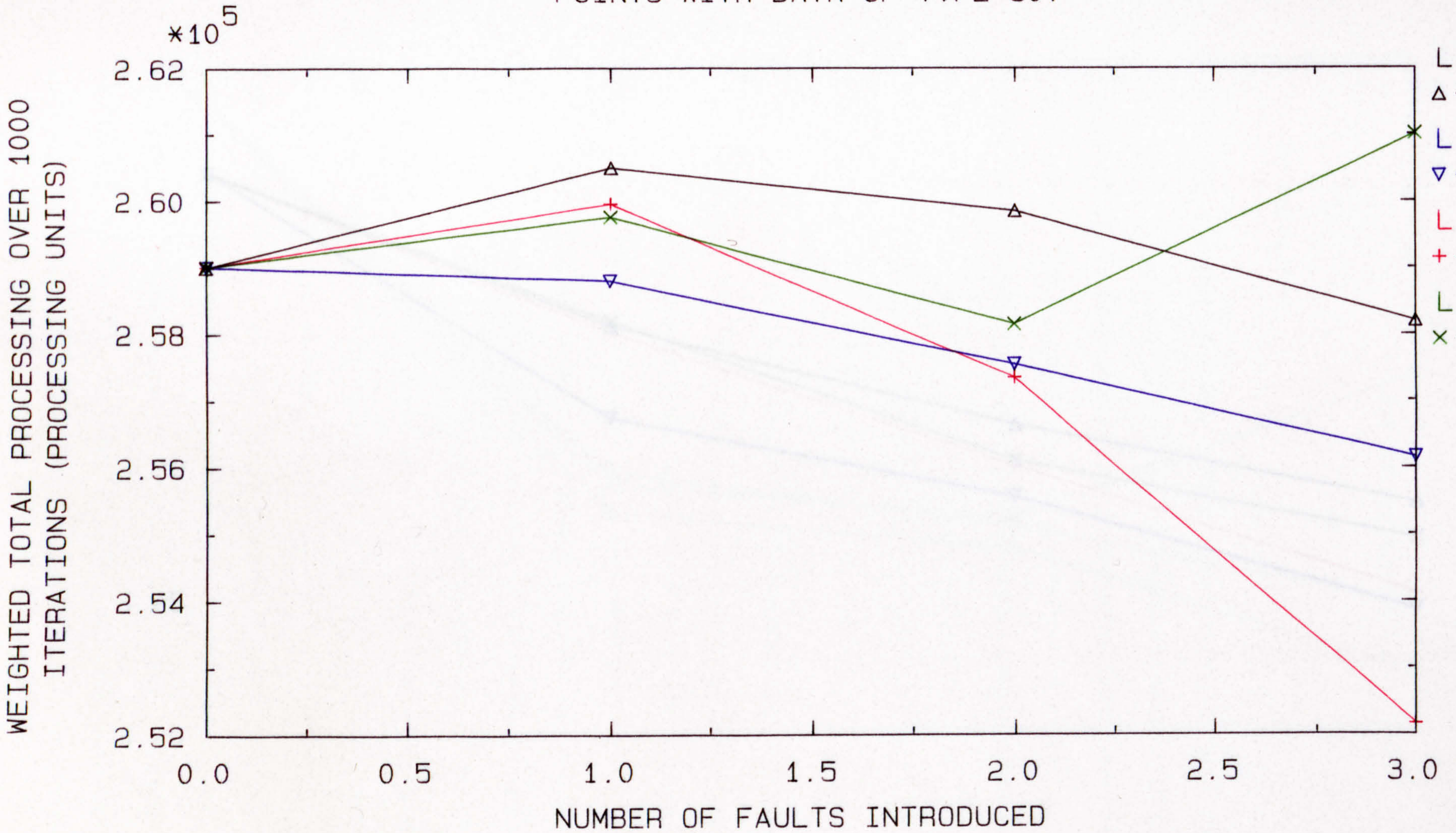


FIG. 4-34 HOMOGENEOUS COMMS3 WITH R=10, L=10 FED FROM ALL POINTS WITH DATA OF TYPE 50.

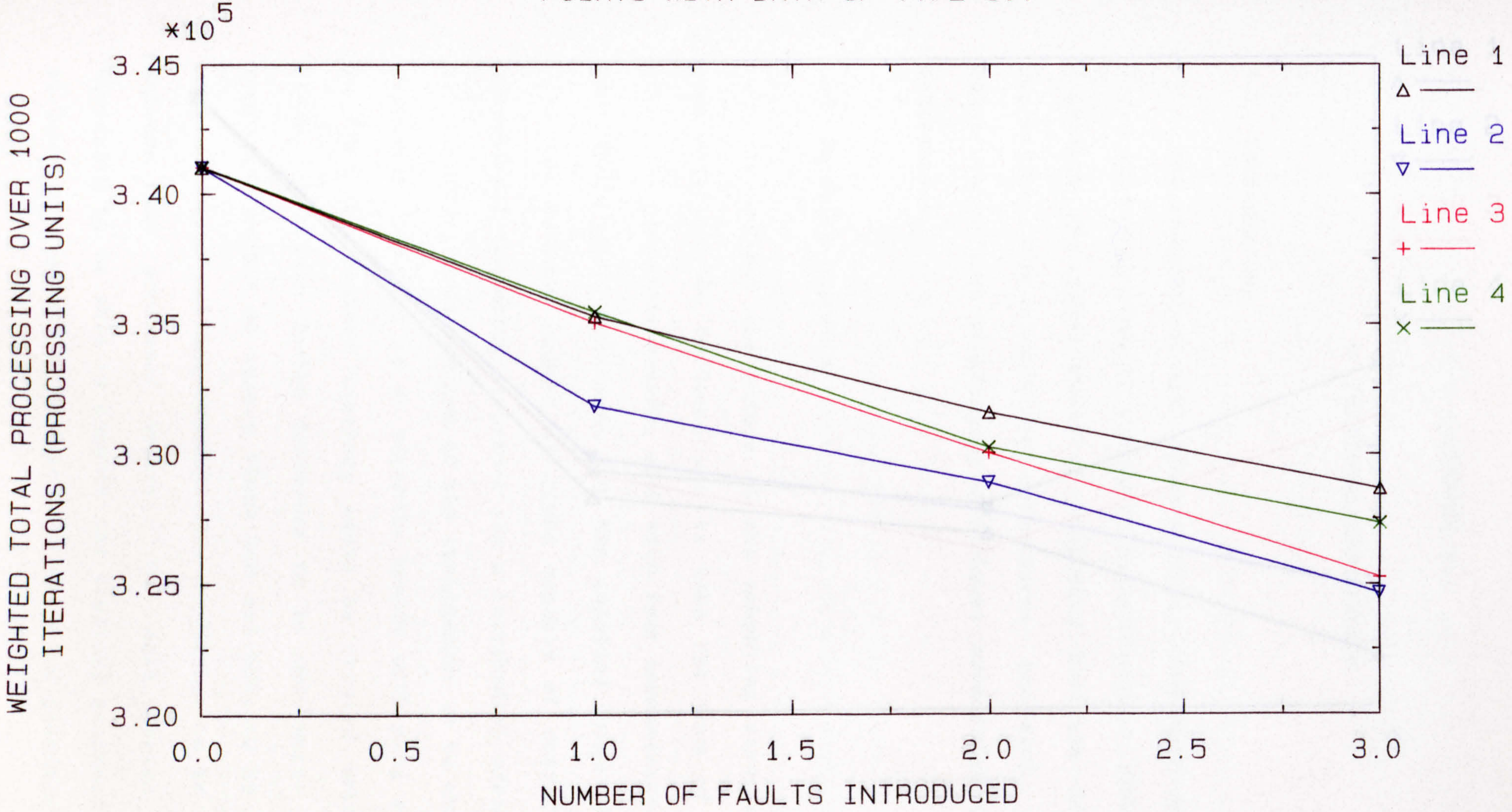
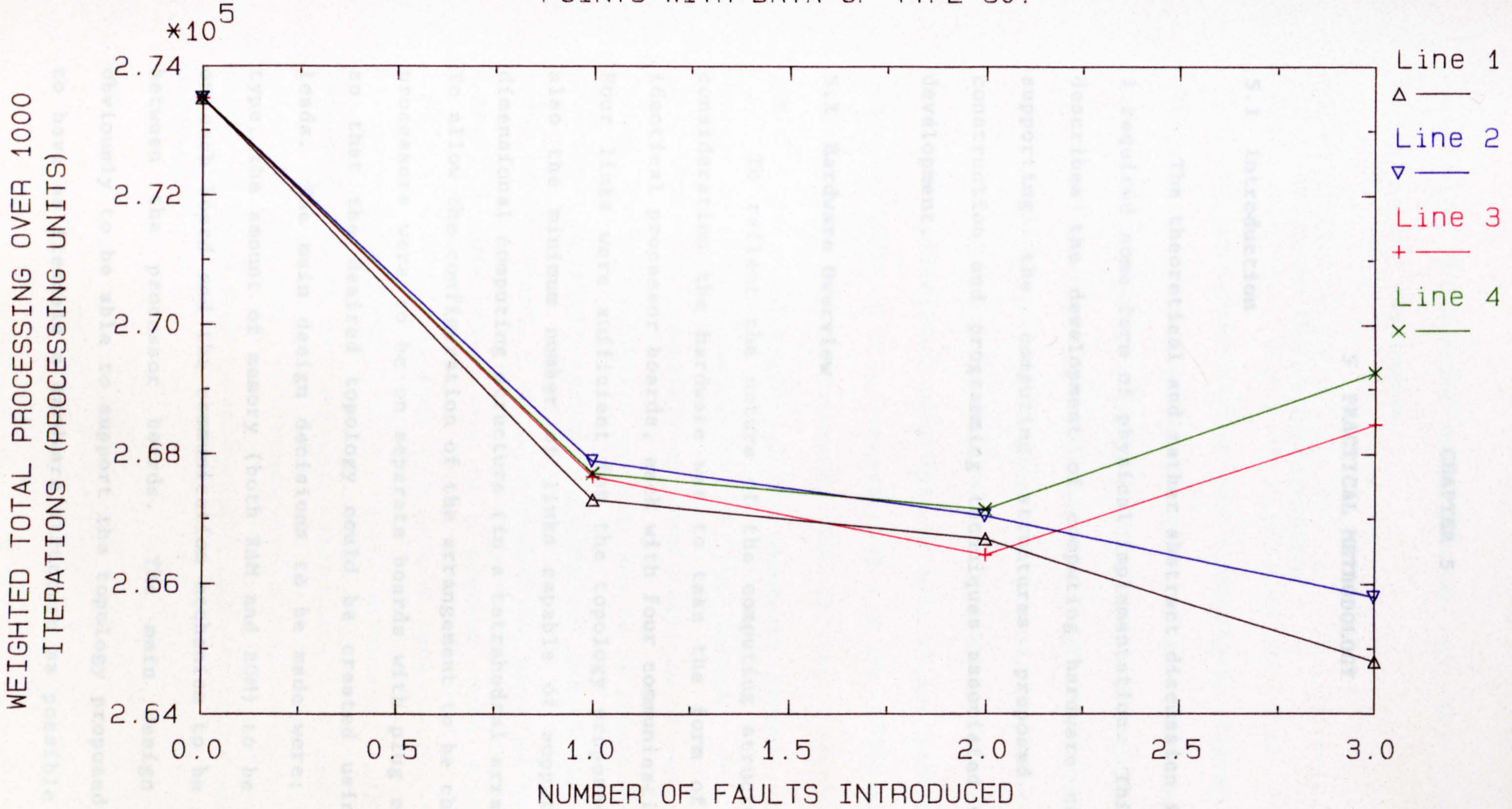


FIG. 4-35 HOMOGENEOUS COMMS4 WITH R=10, L=10 FED FROM ALL POINTS WITH DATA OF TYPE 50.



CHAPTER 5

5 PRACTICAL METHODOLOGY

5.1 Introduction

The theoretical and rather abstract discussion in chapter 3 required some form of physical implementation. This chapter describes the development of computing hardware capable of supporting the computing structures proposed and the construction and programming techniques associated with this development.

5.2 Hardware Overview

To reflect the nature of the computing structure under consideration the hardware was to take the form of separate identical processor boards, each with four communication links. Four links were sufficient for the topology proposed and are also the minimum number of links capable of supporting a 3 dimensional computing structure (in a tetrahedral arrangement). To allow the configuration of the arrangement to be changed the processors were to be on separate boards with plug connectors so that the desired topology could be created using flying leads. The main design decisions to be made were; processor type, the amount of memory (both RAM and ROM) to be contained on each board and the communication mechanism to be available between the processor boards. The main design aim was obviously to be able to support the topology proposed but also to have as flexible a hardware structure as possible to allow

the boards to be used as a general tool for investigating multiprocessor topologies.

5.2.1 Processor Memory Configuration

The processor chosen was a Z-80[163] since these are cheap, plentiful and quite versatile. A great deal of software and hardware support already exists for these devices. A 32k × 8 bit RAM space was decided upon since this was available in a single package and use of this device would keep the component count low whilst providing sufficient memory space for most envisaged applications also leaving half of the Z-80 addressing space for ROM. A U.V. erasable 8k × 8 bit EPROM was decided upon for the Read Only Memory. Use of single devices for the RAM and ROM requirements of the processor greatly reduced the memory decoding requirement of the system. The type of inter-processor communication could take one of many different forms. The communication protocol had to be independant of the processor to allow the possibility of connecting different types of processor, a serial protocol was selected to keep the number of interconnections to a minimum since with a large number of processors, each with several connections, the number of interconnections, with their associated problems of crosstalk, noise and physical placement, can easily become excessive if each interconnection itself requires a large number of wires such as in the case of a parallel communication protocol.

5.2.2 Communication Protocol Selection

The protocol developed by INMOS for use with the TRANSPUTER[78,102,156,148,11,155] was adopted since it met all of the original design considerations and also offered the possibility of direction connection with the TRANSPUTER which has been designed specifically for construction of network computers. The TRANSPUTER protocol also has the advantage that communication and handshaking is carried out using only two wires, reducing the number of connections required by a factor of 2 as compared to one of the more conventional serial protocols. Buffering in the form of 16 bytes of FIFO memory both in the transmit and receive data paths of each link were provided (giving a total of 32 bytes of buffering in each direction). This scheme is similar in many respects to that in the hypercube of Tuazon *et al*[147], using a fifo memory between processors.

5.2.3 Additional Input/Output Facilities

Though not used in this particular application the processor cards also included a parallel input/output device (PIO). Since the emphasis in the design of the hardware was on flexibility the possibility of passing data in and out of each node individually and the ability to simulate a bus interconnection was considered a worthwhile addition to the design.

5.2.4 Power Supply and Physical Support

The multiprocessor hardware was built into a card frame with an STE backplane. The STE backplane was used to provide the power requirements of the boards only, since the communication between the cards was to be by flying leads to allow a wide range of connection topologies to be realised.

5.2.5 External Communication Interface

To allow the computer to receive data and programming information from, and return data to, the outside world interfaces from the TRANSPUTER protocol to a more conventional and widely available protocol were required. Boards to convert from the TRANSPUTER protocol to a standard 8 bit serial protocol at RS-232C voltage levels at 9600 baud were constructed. These boards were used to prototype the TRANSPUTER compatible link circuit before construction of the processor boards was undertaken.

5.3 Interface Board Design

Since the interface boards were constructed before the processor boards in order to prototype the serial link circuit the design of the former will be used to illustrate the design of the link circuit.

5.3.1 Serial Communication Protocol

The serial protocol selected, for reasons outlined above, was that selected for use by INMOS with their TRANSPUTER. The

protocol involves a two-wire physical connection between the links as shown in fig 5.1(a). There are two types of 'packet' that can be sent, an acknowledge 'packet' and a data 'packet' as shown in fig 5.1(b), these packets are sent at a rate of 10 Mbits/sec. Handshaking is achieved through the use of the acknowledge packet, the transmitter sends a data packet and cannot then send another data packet until it has received an acknowledge packet. The TRANSPUTER, and the links designed here, allow an acknowledge packet to be send **during** the reception of data so that continuous data transmission is possible. During bi-directional communication acknowledge packets and data packets are mixed.

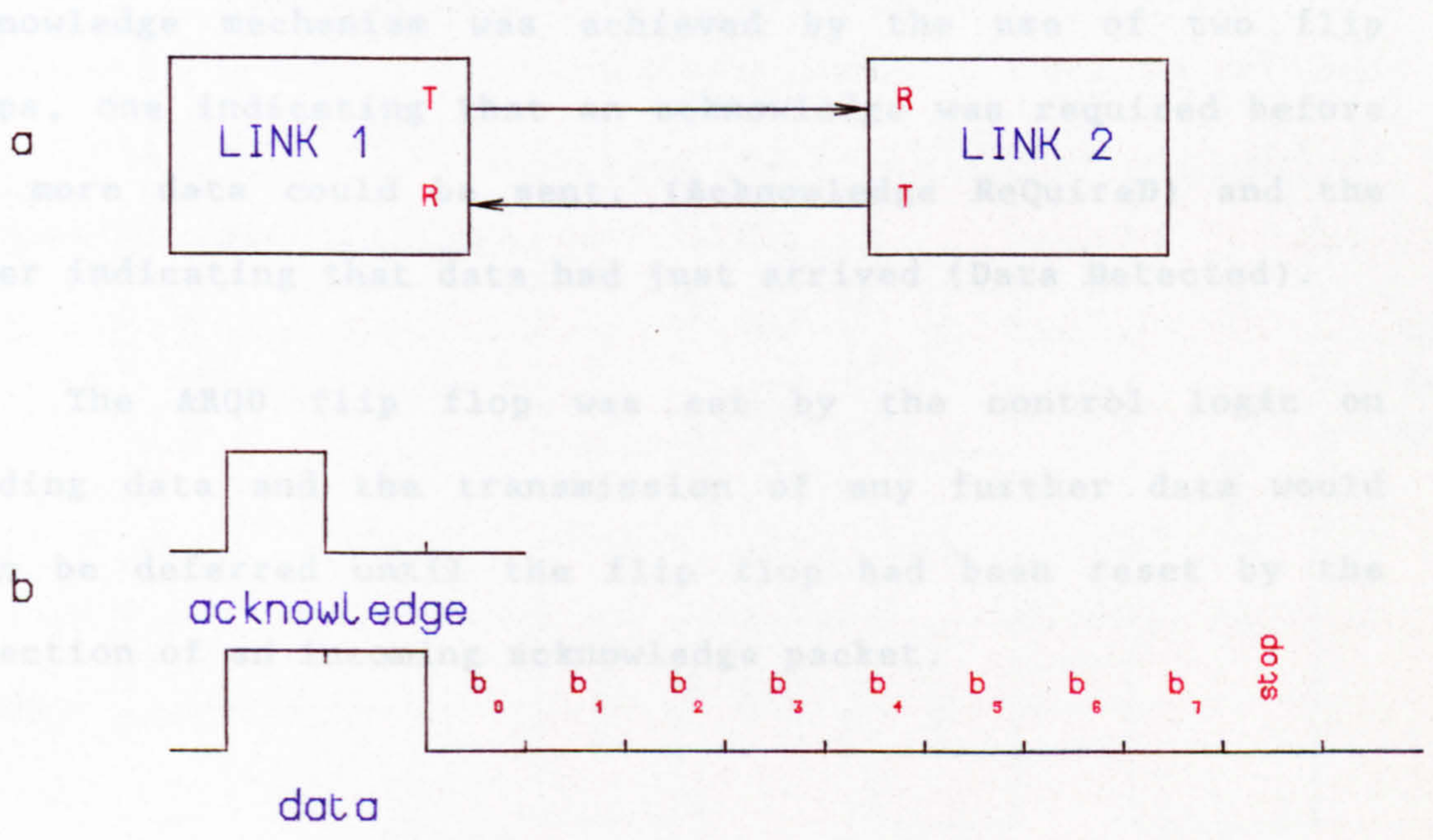


FIG. 5-1 TRANSPUTER COMMUNICATION SCHEME.

5.3.2 Availability of Suitable Devices

The aim of the link circuit was to produce a serial link capable of connecting to virtually any parallel bus, though INMOS had a device to perform this function under development, this was not available initially and when released was prohibitively expensive. This coupled with the considerable number required made the development of a cheap alternative viable.

5.3.3 Transmit Receive Synchronisation

The circuit developed is shown in fig 5.2. The circuit has three main elements, the receive shift register, the transmit register and the control logic. The handling of the acknowledge mechanism was achieved by the use of two flip flops, one indicating that an acknowledge was required before any more data could be sent, (Acknowledge ReQuireD) and the other indicating that data had just arrived (Data Detected).

The ARQD flip flop was set by the control logic on sending data and the transmission of any further data would then be deferred until the flip flop had been reset by the detection of an incoming acknowledge packet.

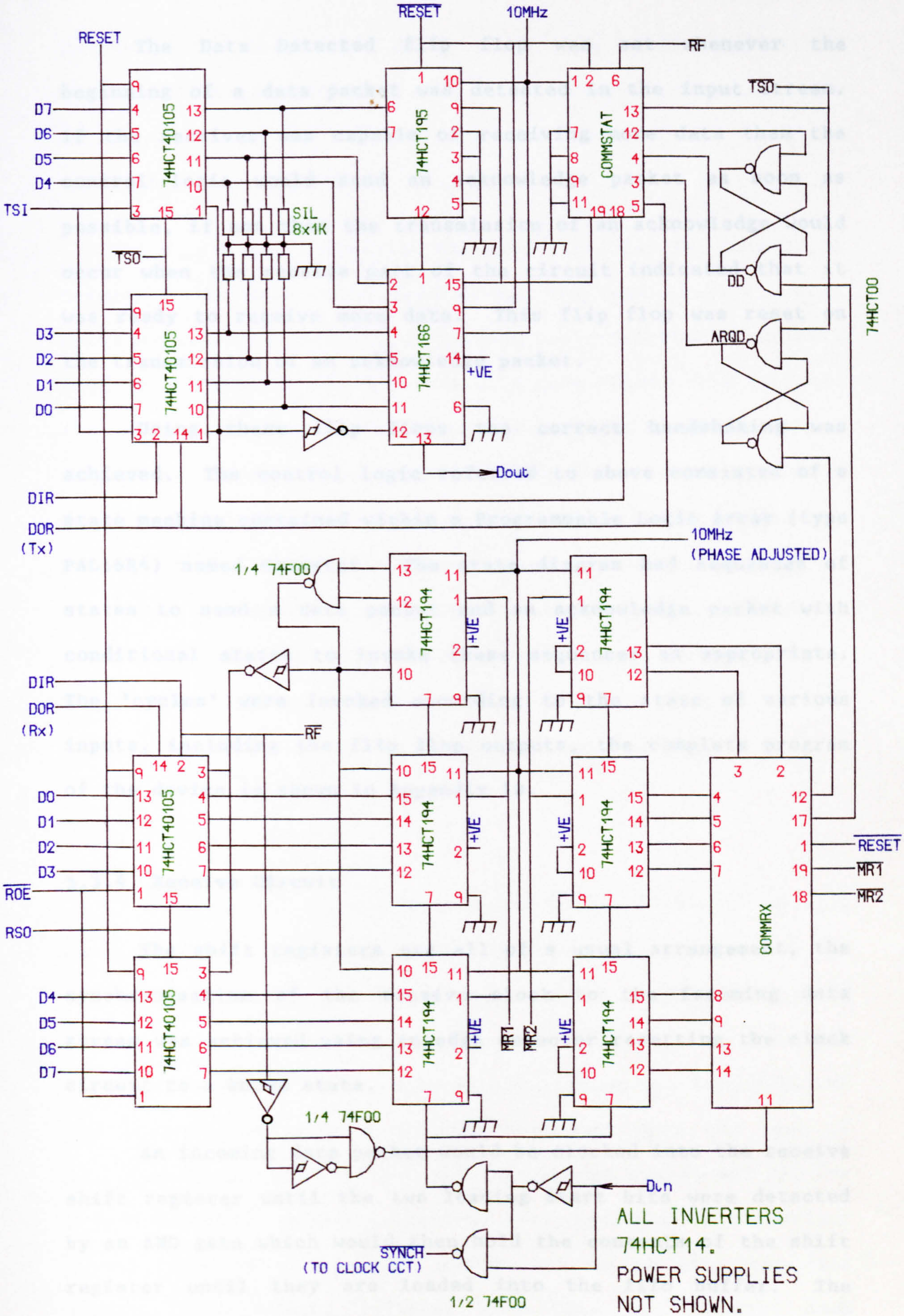


FIG. 5-2 LINK CIRCUIT.

The Data Detected flip flop was set whenever the beginning of a data packet was detected in the input stream. If the receiver was capable of receiving more data then the control logic would send an acknowledge packet as soon as possible, if not then the transmission of an acknowledge would occur when the receive part of the circuit indicated that it was ready to receive more data. This flip flop was reset on the transmission of an acknowledge packet.

Using these flip flops the correct handshaking was achieved. The control logic referred to above consisted of a state machine contained within a Programmable Logic Array (type PAL16R4) named Commstat. The state diagram had sequences of states to send a data packet and an acknowledge packet with conditional states to invoke these sequences as appropriate. The 'cycles' were invoked according to the state of various inputs, including the flip flop outputs, the complete program of the device is shown in Appendix 10.

5.3.4 Receive Circuit

The shift registers are all of a usual arrangement, the synchronisation of the receive clock to the incoming data stream was achieved using an edge detector resetting the clock circuit to a known state.

An incoming data packet would be clocked into the receive shift register until the two leading start bits were detected by an AND gate which would then hold the contents of the shift register until they are loaded into the fifo buffer. The 'hold' signal was used to load the data into the buffer. The

change of the Data Input Ready (DIR) flag of the buffer was used to clear the shift register. The transfer of data occurs within the stop bit of the incoming data packet to allow continuous data transfer.

A second shift register was maintained for the purpose of detecting the headers of acknowledge and data packets. This shift register is required to differentiate between the genuine headers and similar bit patterns occurring within data packets, the necessary logic functions being contained within a Programmable Logic Array (type PAL16L8) given the name commrx. To ensure that the two bits used to detect the header bit pattern arrive after the other outputs of the shift register the last two bits of the shift register were subjected to feedback within the PAL. This technique has, perhaps rather surprisingly, proved reliable effective and reproducable. The program for this device can be found in appendix 10.

5.3.5 Transmit Circuit

The transmit shift register had to be capable of sending data packets, with a suitable pair of start bits, or an acknowledge packet. This was achieved on loading the shift register by ensuring that a '1' was always loaded into the leading bit of the shift register and loading the second bit of the shift register with a value dependant upon the type of packet to be sent. ('0' for and acknowledge packet and '1' for a data packet). To avoid spurious data appearing in the output stream when an acknowledge packet was to be sent the data inputs to the shift register were held low by pull down

resistors and the output from the buffer disabled, the signal to disable the buffer also being used to generate the appropriate value for the second bit of the header.

5.3.6 Interface to RS-232 C

The TRANSPUTER link circuit described above is easily connected to virtually any parallel bus, decoding for connection to a Z-80 bus was achieved using a single PAL (type 16L8) named decode1 the design of which can be found in appendix 10. The arrangement is shown in fig 5.3. Since only three control lines are required by the link circuit two such links can be controlled by a single decoding PAL.

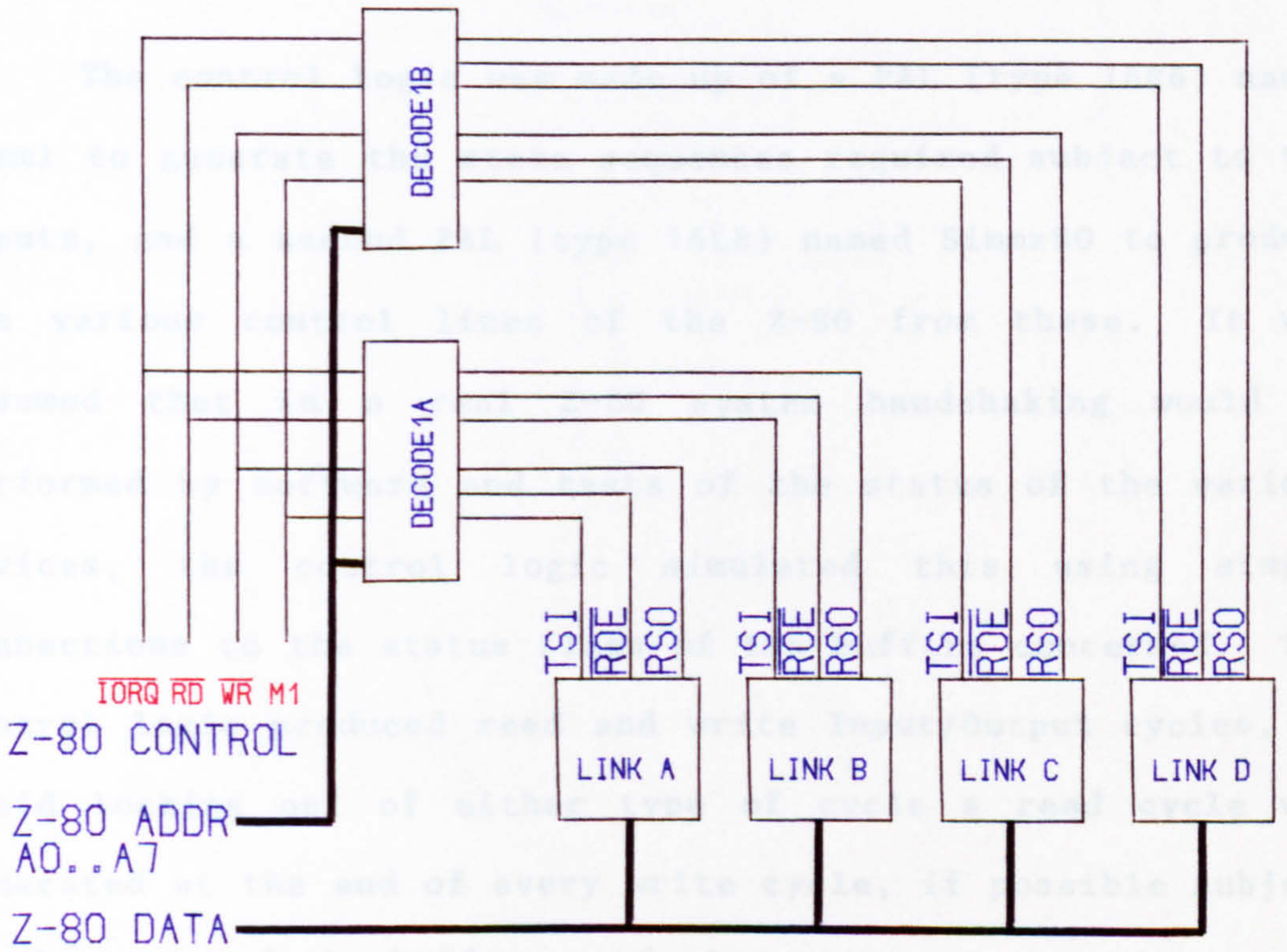


FIG. 5-3 DECODING OF THE Z-80 BUS TO CONNECT TO THE LINK CIRCUIT OF FIG. 5-2.

To prototype the link circuit and decoding and to produce an RS-232C to TRANSPUTER interface a means of producing a pseudo Z-80 bus from a serial connection was required. The circuit to fulfill this function is shown in fig 5.4.

5.3.6.1 Parallel Bus simulation

Conversion to and from serial data was performed by an industry standard Universal Asynchronous Receiver Transmitter (type 6402), this was controlled by a state machine to produce simulated Read and Write Input/Output cycles of a Z-80. The parallel inputs and outputs of the UART were linked together so that by the use of suitable control signals the operation of a parallel bus could be reproduced.

5.3.6.2 Control Logic

The control logic was made up of a PAL (type 16R6) named Simm1 to generate the state sequences required subject to the inputs, and a second PAL (type 16L8) named Simmz80 to produce the various control lines of the Z-80 from these. It was assumed that in a real Z-80 system handshaking would be performed by software and tests of the status of the various devices, the control logic simulated this using simple connections to the status flags of the buffers concerned. The control logic produced read and write Input/Output cycles, to avoid locking out of either type of cycle a read cycle was generated at the end of every write cycle, if possible subject to the state of the buffers, and vice versa.

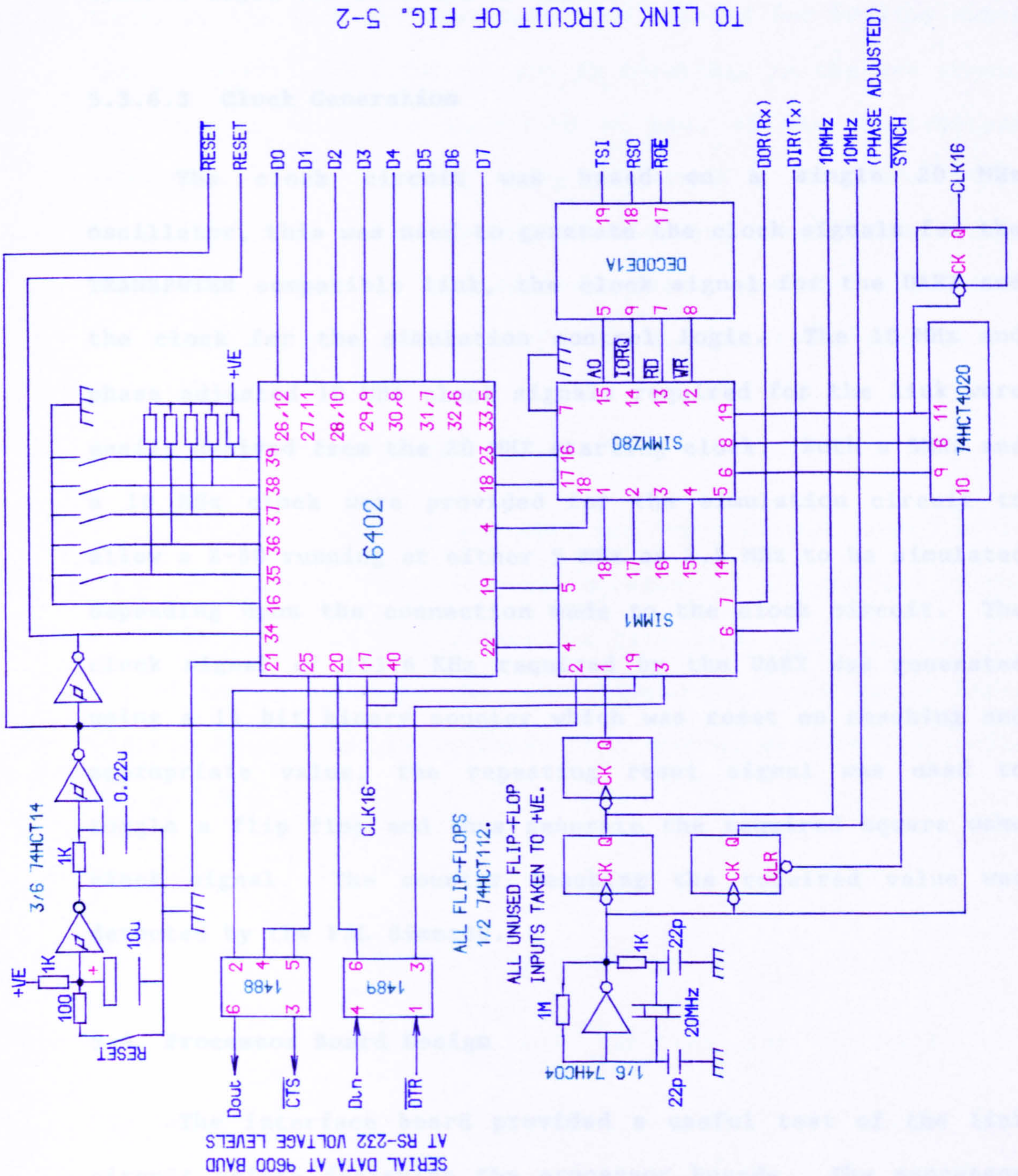


FIG. 5-4 CIRCUIT TO INTERFACE SERIAL DATA TO LINK CIRCUIT OF FIG. 5-2.

Though the Z-80 control signals could not be generated in precise time relationship the overall form of the cycles could be produced easily. The programs for the PALs used for the control logic can be found in appendix 10.

5.3.6.3 Clock Generation

The clock circuit was based on a single 20 MHz oscillator, this was used to generate the clock signals for the TRANSPUTER compatible link, the clock signal for the UART and the clock for the simulation control logic. The 10 MHz and phase adjusted 10 MHz clock signals required for the link were easily derived from the 20 MHZ starting clock. Both a 5MHz and a 10 MHz clock were provided for the simulation circuit to allow a Z-80 running at either 5 MHz or 2.5 MHz to be simulated depending upon the connection made to the clock circuit. The clock signal of 153.6 KHz required by the UART was generated using a 14 bit binary counter which was reset on reaching and appropriate value, the repeating reset signal was used to toggle a flip flop and thus generate the required square wave clock signal. The counter reaching the required value was detected by the PAL Simmz80.

5.4 Processor Board Design

The interface board provided a useful test of the link circuit before designing the processor boards. The processor board was based around a Z-80 microprocessor, 32 Kilobytes of Read/Write memory in the form of a single 32k x 8 bit static RAM and 8 kilobytes of Read Only Memory in the form of a single

8k x 8 bit Erasable Programmable ROM. A Z-80 parallel Input Output (PIO) device was also included on each board.

The circuit of the processor board is shown in fig 5.5. Three of the link circuits have been omitted for brevity since their connection to the circuit is identical to the one shown. The link circuits are addressed as part of the Input/Output memory map, the decoding being performed by two PALs (type 16L8), Decode1a and Decode1b, the complete programs for these devices are shown in appendix 10. These decoders also decode the signals for input latches for the buffer status signals.

5.4.1 Link Status Latches

Two input latches provided two Input/Output mapped ports to allow the programmer to read the state of the buffers of the link circuits, one port giving the Data Input Ready signals for all of the buffers and the other giving the Data Output Ready signals for all of the buffers. The latches have the following bit patterns :

DOR Latch

Bit	0	1	2	3	4	5	6	7
DOR	RxA	RxB	RxC	RxD	TxA	TxB	TxC	TxD

DIR Latch

Bit	0	1	2	3	4	5	6	7
DIR	TxA	TxB	TxC	TxD	RxA	RxB	RxC	RxD

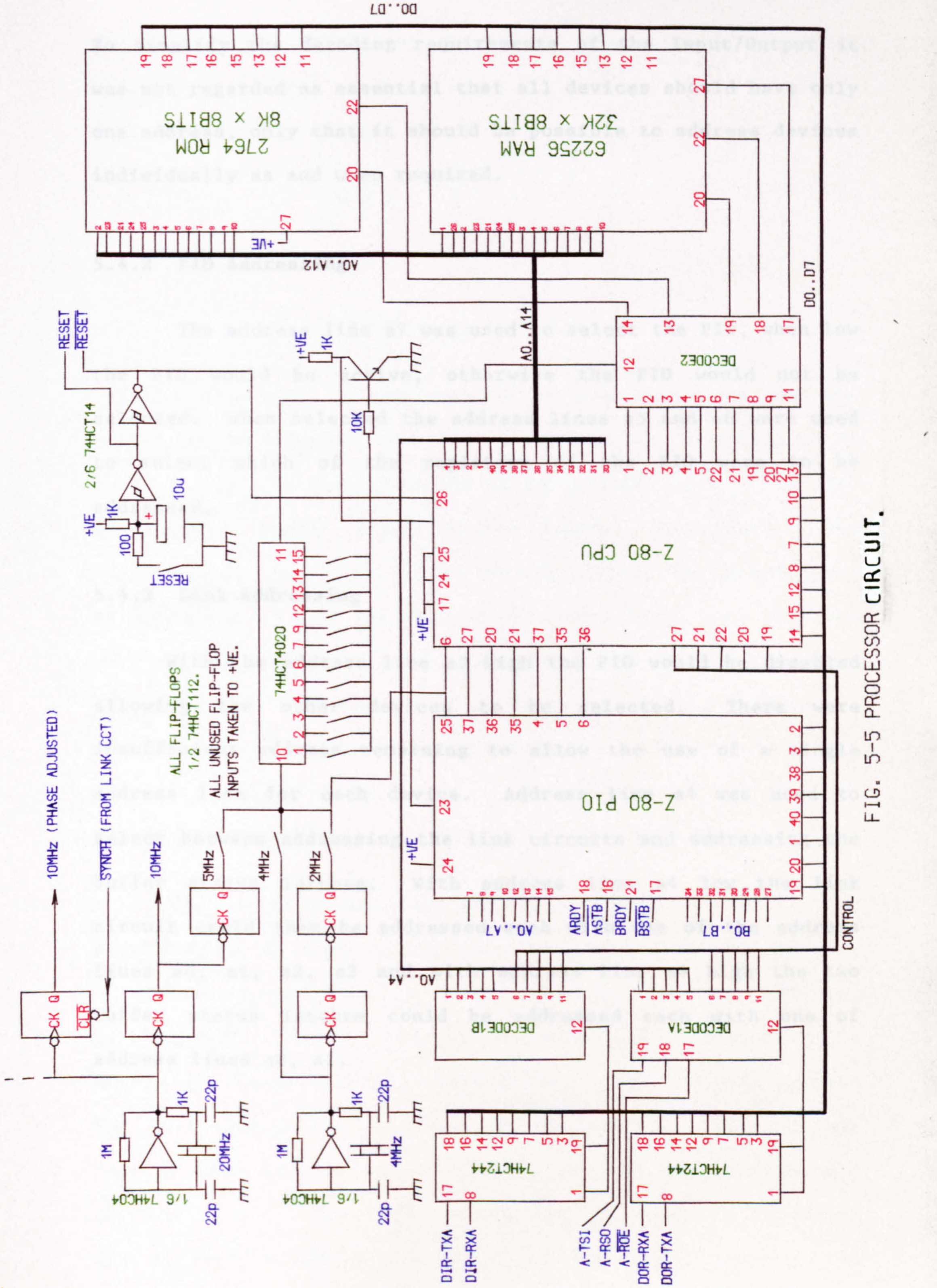


FIG. 5-5 PROCESSOR CIRCUIT.

To simplify the decoding requirements of the Input/Output it was not regarded as essential that all devices should have only one address, only that it should be possible to address devices individually as and when required.

5.4.2 PIO Addressing

The address line a7 was used to select the PIO, when low the PIO would be active, otherwise the PIO would not be selected. When selected the address lines a5 and a6 were used to select which of the registers of the PIO were to be addressed.

5.4.3 Link Addressing

With the address line a7 high the PIO would be disabled allowing the other devices to be selected. There were insufficient address remaining to allow the use of a single address line for each device. Address line a4 was used to select between addressing the link circuits and addressing the buffer status latches. With address line a4 low the link circuit could then be addressed each with one of the address lines a0, a1, a2, a3 and with address line a4 high the two buffer status latches could be addressed each with one of address lines a0, a1.

The mapping of the links to the address lines is:

Link A a0

Link B a1

Link C a2

Link D a3

which corresponds to the mapping of the bits used for the buffer status latches, this allows the address of a link to be used to mask the data from the buffer status latch to extract the buffer status using simple logical operations. The Input/Output addressing scheme chosen allowed the decoding to be performed by only two devices and also allows more than one device to be addressed simultaneously should this be required, though this is only meaningful on write cycles (e.g. output to more than one link circuit simultaneously).

5.4.4 Memory Decoding

The memory decoding was similarly designed to that of the Input/Output in that reflections of regions of memory were considered acceptable in order to minimise the decoding requirements. Memory address decoding was performed by a single PAL (type PAL16L8) named Decode2, the complete program for this device can be found in appendix 10. The RAM was enabled with address line a15 high, the ROM was enabled with address line a15 low. The scheme was very easy to implement and gives the memory map shown in fig 5.6. Decode2 also provided the interrupt acknowledge signal required by the

interrupt generator circuit which is described below in connection with the clock circuit.

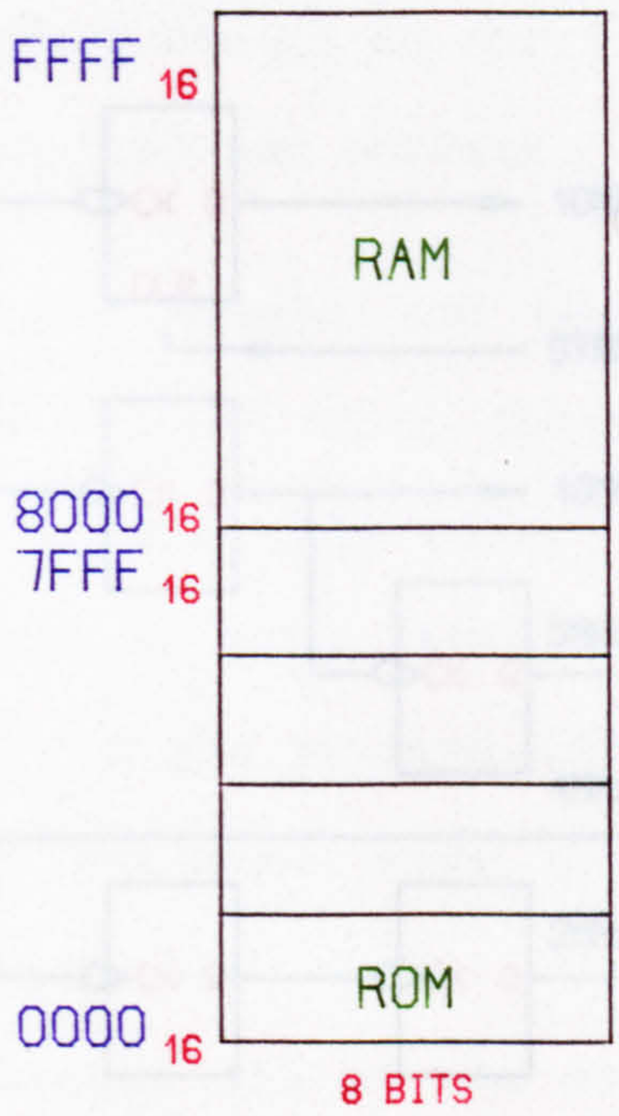


FIG. 5-6 MEMORY MAP OF THE Z-80 SYSTEM.

5.4.5 Clock Generation

The clock circuit of the processor board had several functions to perform, it had to provide; the clock signal for the microprocessor, the 10 MHz clock signal for the link circuits transmit side, the phase adjusted 10MHz clock signal for the link circuits receive side and an interrupt signal for the microprocessor. The clocks were derived from one of a 20MHz or a 4MHz starting clock frequency. The clock circuit is shown in fig 5.7, this also provides 2, 4 and 5 MHz clock signals for the microprocessor, selectable by DIL switches on the processor board. The interrupt signals were generated from a counter, the period between interrupts being selectable by DIL switches from frequencies of the microprocessor clock

frequency divided by one of 2, 2^4, 2^5, 2^6, 2^7, 2^8, 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}.

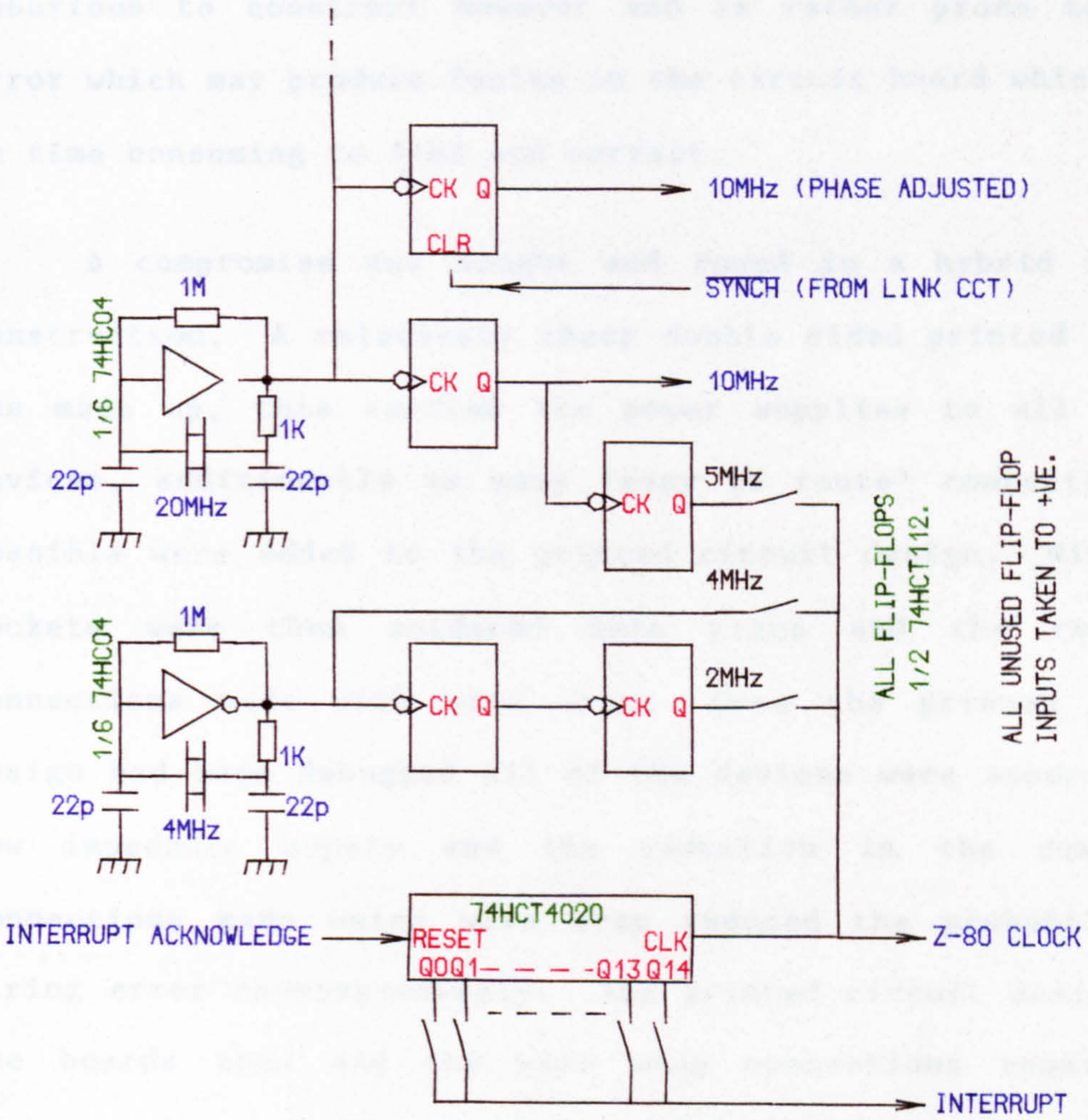


FIG. 5-7 CLOCK GENERATION CIRCUIT FOR THE PROCESSOR BOARD.

5.5 Hardware Construction

The design outlined above presented the problem of requiring a large number of devices and a consequently large number of interconnections, a fairly high component density had to be achieved to allow the processor boards to be kept to a reasonable size. Printed circuit techniques were considered but were rejected since an extremely expensive multi-layer board would be required. Wire wrap construction was considered

and had the advantage that the components could be mounted adjacent to each other; however, wire wrap is extremely laborious to construct however and is rather prone to human error which may produce faults in the circuit board which would be time consuming to find and correct.

A compromise was sought and found in a hybrid form of construction. A relatively cheap double sided printed circuit was made up, this carried the power supplies to all of the devices, additionally as many 'easy to route' connections as possible were added to the printed circuit design. Wire wrap sockets were then soldered into place and the remaining connections made with wire wrap. Once the printed circuit design had been debugged all of the devices were assured of a low impedance supply and the reduction in the number of connections made using wire wrap reduced the probability of wiring error correspondingly. The printed circuit designs for the boards used and the wire wrap connections required to complete the circuit are to be found in appendix 6.

5.6 Software Overview

5.6.1 Programming Languages

The microprocessor system was programmed using a combination of Z-80 assembly code[163] and the C programming language[83,117,47]. The C compiler used produced Z-80 assembly language as an intermediate stage of compilation allowing easy interception and modification of the compiled assembly code if desired. The stages involved in producing a binary file suitable for down line loading are shown in the

form of t (translation) diagrams in fig 5.8.

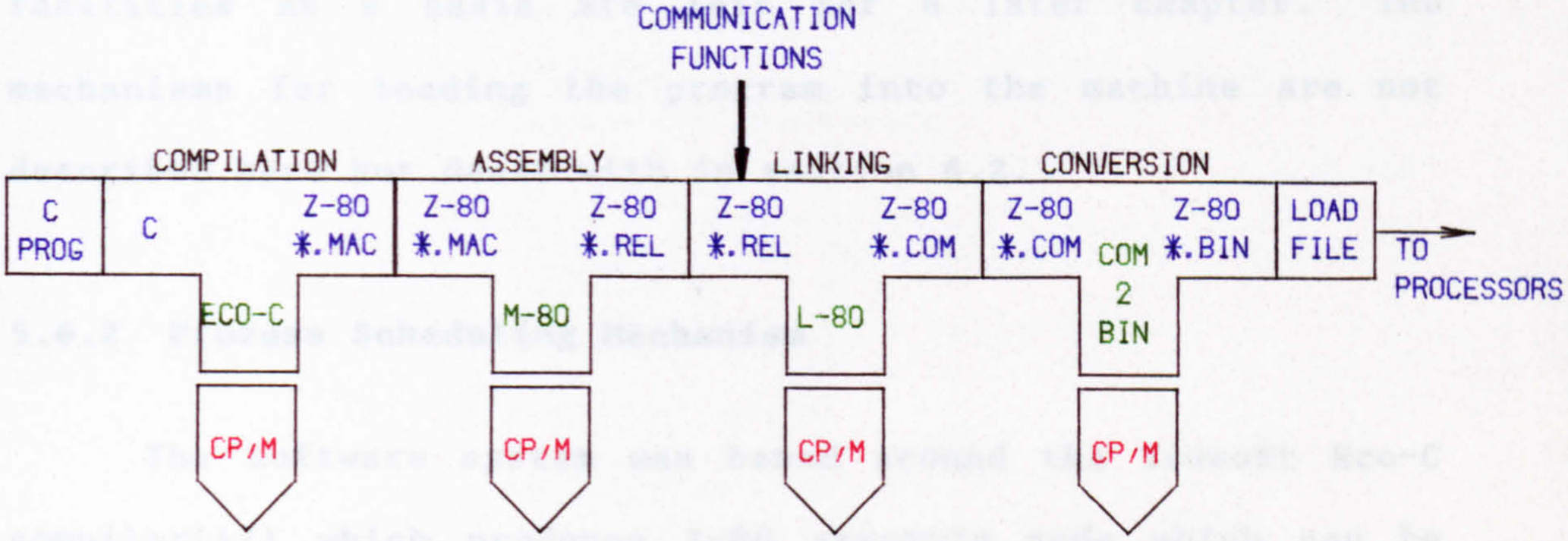


FIG. 5-8 TRANSLATIONS REQUIRED TO PRODUCE A FILE SUITABLE FOR DOWN LOADING.

A scheme of concurrent execution of programs within each processor was created, loosely based on the Communicating Sequential Processes (CSP) model of Hoare[67] which has been developed by INMOS to give the programming language OCCAM[79,157] used to program concurrent systems using the TRANSPUTER[78,102,156,148,11,155]. In the CSP model communication between processes takes place only when both processes are ready to communicate, the processes waiting if required. In the model created here a process sending a value may send its value and proceed independant of the state of the receiving process, the CSP style interaction may be simulated; using acknowledgement sent in the reverse direction it is possible to ensure that the sending process is forced to wait if this is required.

The mechanisms used to provide the fundamental functions required for the concurrent execution of programs are described here, the development of various algorithms using these facilities as a basis are left for a later chapter. The mechanisms for loading the program into the machine are not described here but dealt with in section 6.2.

5.6.2 Process Scheduling Mechanism

The software system was based around the Ecosoft Eco-C compiler[47] which produces Z-80 assembly code which can be intercepted and if necessary modified before use. The time slice mechanism and the functions to interact with the Input/Output devices were the only additions to the standard compiler output, the programs used to make up the multiprocessing system are to be found in appendix 11. The start address of the functions to be executed concurrently as processes were easily accessed at the stage of linking the program, allowing them to be incorporated into the concurrent execution scheme. It proved possible to make all of the necessary additions at the compilation (by inclusion of suitable data definitions) and linking (by linking with hand written code) stages avoiding the need to edit the assembly code produced by the compiler from the program source code. Program and data reside in separate segments of memory, the layout created within memory is shown in fig 5.9. The process scheduling was simple preemptive scheduling on a Round Robin (RR) basis[39,144]. The data structure created for use with the time slice generation mechanism is shown in fig 5.10, the first two bytes indicate which process is currently running and

the number of the highest numbered process respectively. Pairs of bytes are then used to store 16 bit addresses, these being the value contained in the stack pointer for each process.

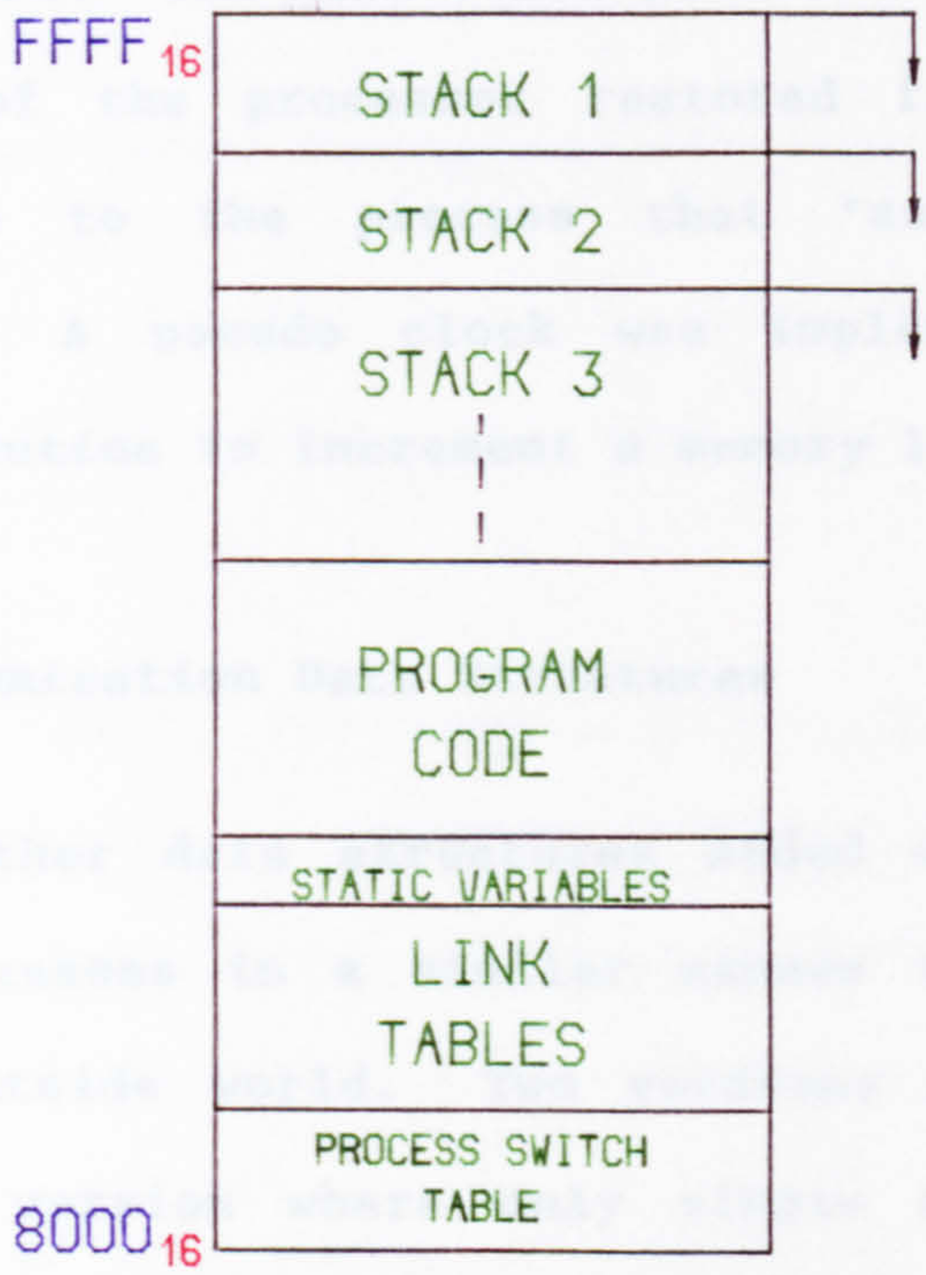


FIG. 5-9 MEMORY USAGE BY THE MULTIPROCESSING SYSTEM.

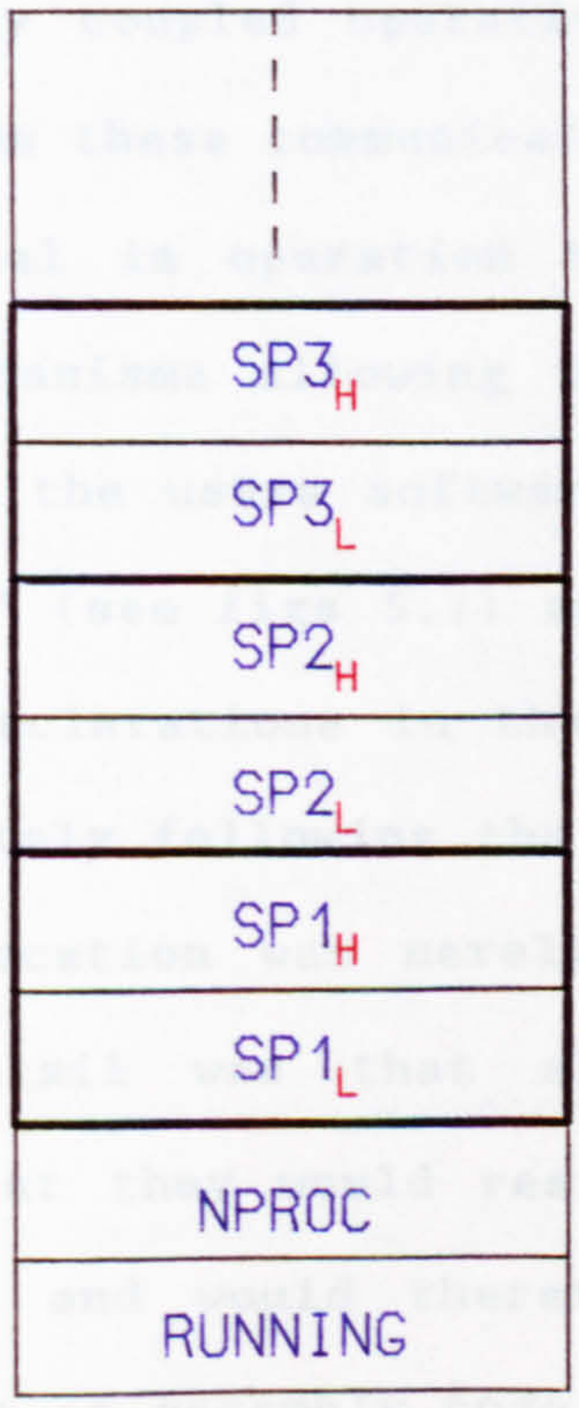


FIG. 5-10 PROCESS SWITCH TABLE.

When an interrupt occurred the state of the processor was saved on the stack associated with that process, the stack pointer was then saved in the appropriate place in the list of stack pointers. The next stack pointer would then be restored, the state of the processor restored from this stack and a return made to the process that 'suffered' an interrupt previously. A pseudo clock was implemented by using this interrupt routine to increment a memory location.

5.6.3 Communication Data Structures

The other data structures added were for communication between processes in a similar manner to their communication with the outside world. Two versions of this software were created, a version where only single bytes could be passed between processes with no buffering (see appendix 12) and a second version including a fifo buffer between processes to allow more loosely coupled operation (see appendix 13). The functions to access these communication mechanisms were written to appear identical in operation to the programmer for both communication mechanisms allowing the use of either mechanism with no change in the users software. For each process to be run a 'link table' (see figs 5.11 and 5.12) was created, using static variable declarations in the C code, this was arranged to reside immediately following the list of stack pointers but choice of this location was merely a matter of convenience. The important detail was that all such link tables were contiguous and that they would reside in a location known to the linker used, and would therefore be accessible to the functions, written in assembly code, for the transfer of data.

Since the 'link tables' were created using static variable declarations within the C program they would also be available to the C programmer if desired.

5.6.4 Communication Functions

For both the fifo buffered and single byte communication mechanisms the functions provided to access these link tables were :

```
void   swbyte(proc,link,value);
value  rwbyte(proc,link);
report stbyte(proc,link,value);
report rtbyte(proc,link);
```

where proc is the number of the process to be accessed (in reality this refers to which link table is accessed and the use of other than one link table for a process will change the mapping of proc to the process to be communicated with) and link is the particular link in the process's link table that is to be accessed; value refers to the returned value and report refers to a success or failure indication depending upon the state of the data stream being written to or read from. These functions perform similar operations to the ? and ! operators of CSP[67] and OCCAM[79]. The functions send wait byte and receive wait byte do not return until their service has been performed. The functions send test byte and receive test byte both perform their service immediately if possible and if not return with an indication of failure (in practice a report value greater than 255).

5.6.5 Inter-Process Communication Addressing

In keeping with the style of the flags from the external links the flags indicated DIR, that is when the bit was set there was no valid byte in the table and a byte could be written. The addressing of the links within a table is also in keeping with the style of the external links in that the links were addressed by particular bits of the value of the link, the nth link being addressed by a value of 2^n ($0 \leq n \leq 7$). With the internal links however, unlike the external links, simultaneous access to more than one link is not possible with the functions provided.

The functions used for internal transfers were also used for transfers of data to and from external links with a value for proc of zero being used to indicate external rather than internal links.

5.6.6 Link Table Structure

The structure of a link table for un-buffered communication is shown in fig 5.11. The first byte of the table was a set of flags used to indicate the presence of valid data in the link buffers. Each bit of the flag byte was used to indicate the presence of a valid byte in one of the link buffers, up to a maximum of 8. Though this scheme may appear to limit the total number of links to a process the absence of protection mechanisms in the simple environment constructed here permit more than one link table for each process and even access to tables used by other processes.

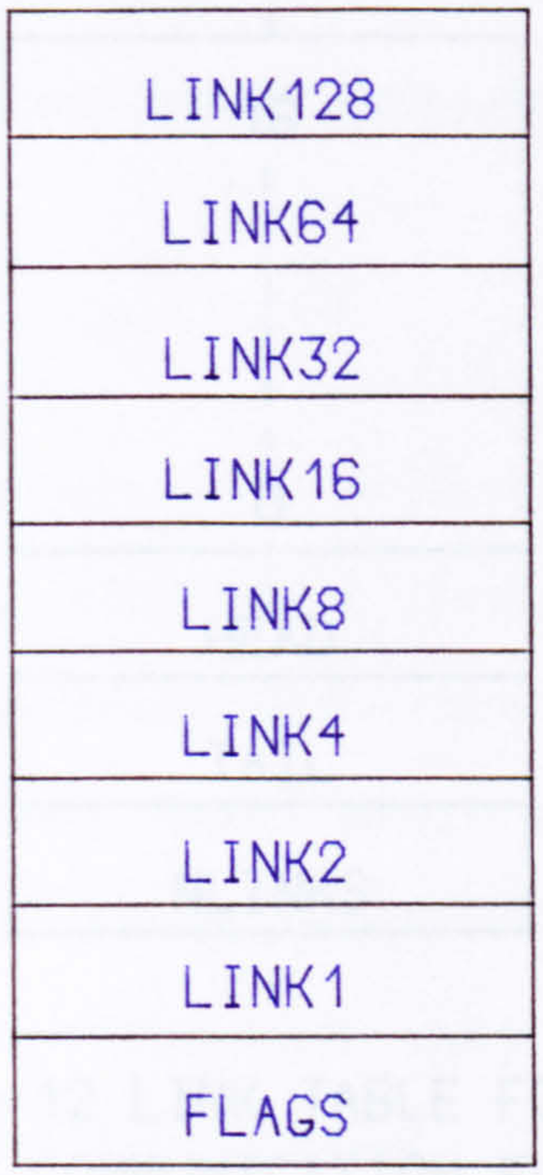


FIG. 5-11 LINK TABLE FOR BYTE BUFFERED COMMUNICATION FUNCTIONS.

Fifo-buffered communications while offering a reduced computing overhead by the avoidance of un-necessary process switching requires a greater amount of memory in its implementation of the link table. The structure created is shown in fig 5.12, the first byte of the structure indicates the number of such buffers associated with the process, for each of these buffers there is then a pair of bytes holding the head and tail values for the queue and then follows one area of memory set aside for the queue. This whole structure is repeated for each process.

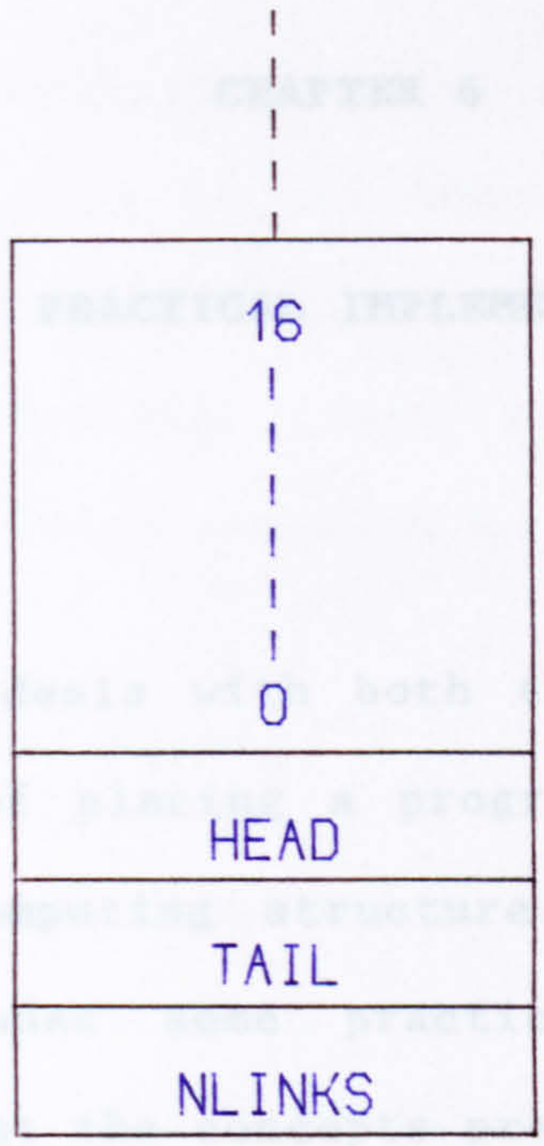


FIG. 5-12 LINK TABLE FOR FIFO
BUFFERED COMMUNICATION FUNCTIONS.

The hardware and software outlined in this chapter describe the basis upon which the concurrent processing scheme was mounted, as stated above the mechanisms for programming, in the sense of getting compiled programs into the memory of the individual processors, are described later in chapter 6 section 6.2. This chapter has only given the philosophy behind the concurrent operating system, the details of implementation in the form of listings of the component parts are to be found in appendices 11 through to 14.

CHAPTER 6

6 PRACTICAL IMPLEMENTATION

6.1 Introduction

This chapter deals with both the abstract and the more tangible elements of placing a program within the processing elements of the computing structure under discussion. This chapter also includes some practical details of software required to implement the concepts presented previously and, in addition, the implementation of various algorithms for use in such a multiprocessing environment.

6.2 Programming Methods

6.2.1 Circuit Switching

The initial method of passing the program to the machine that was considered was to use steerable data streams, similar to the steerable packets used to program the Cosmic Cube[134] but using a circuit switching technique rather than packet switching. This method would require that the structure and the connection point to the structure be known to the loading program. There are two well established methods of traversing graphs, *depth first search* and *breadth first search*[132,41], and either of these could be used as the basis of an algorithm for programming the computing structure proposed.

For each of these methods the program required could be sent down a single link to the computing structure, the node currently being addressed changing either according to a depth first search or breadth first search pattern with the programming circuit being switched to suit.

6.2.1.1 Storage Requirements

This method of programming would allow all of the processing nodes to be programmed individually though this would require copies of all of the programs of the individual nodes to be retained by the programming source. This would involve having a very large amount of storage capacity at the programming source to hold all of the programs. If large programs were required to be loaded into processing nodes then this could possibly lead to a restriction on the number of nodes in the system, however for a 64 Kilobyte 'program' to be down loaded to 1000 processing nodes this still only requires 64 Megabytes of storage which, though large, is not an un-realistic figure, especially since not all of the data has to reside in memory simultaneously. Also of some significance is the time required to complete the programming of the nodes.

6.2.1.2 Programming Time

In the case of circuit switching, and also packet switched programming techniques, where a distinct progam is required to be loaded into each processing node only one node may be subject to programming at any instant. Considering again the case of 64 Kilobyte programs to be loaded into 1000

processing nodes down links of 10 Megabit/second capacity (this figure is marginally better then the transmission speed of the serial links used with the transputer[78]). The time taken to program all of the nodes would be approximately

$$1000 \times 8 \times \frac{64 \times 10^3}{10 \times 10^6} \text{ seconds}$$

that is 51.2 seconds. This is quite a long period if programming has to be performed frequently and it should also be borne in mind that this is an absolute minimum figure and would almost certainly be increased through the use of any routing information, error correcting redundancy etc.

6.2.2 Identical Mutual Programming

Though for many applications processing nodes are required to have distinct programs running in separate nodes the homogeneous processing scheme under investigation required the same processing to be applied to all incoming data. This allowed all of the processing nodes to run the same program, which suggested the possibility of having processing nodes pass the program between each other.

Using this method only one node would require programming initially and this node could then go on to program its immediate neighbours which would then program their immediate neighbours and so on until all of the processing nodes had been programmed.

6.2.2.1 Storage Requirements

This method has advantages in the amount of storage required for the programming source in that only one copy of the program would be required by the programming source with a consequent reduction in the amount of storage space required.

6.2.2.2 Programming Time

The time required to program a network computer by mutual programming will depend upon the connection graph of the computer. The worst case would be where processors were arranged in a linear arrangement and the program supplied to one end as in fig 6.1a. In this case only one processor would be undergoing programming at any one time and the time taken to program the network would be almost the same as that taken in the case of circuit or packet switching of distinct programs. An improvement in the programming time could be achieved, without changing the structure, simply by supplying the program to one of the processors other than the end one, as in fig 6.1b. The 'program front' will then proceed in both directions away from the originating node simultaneously.

The time to program the system obviously depends upon the graph of the interconnection pattern of the nodes as seen by the program source. If the processing nodes encountered by the program have a high degree then the program will be distributed more rapidly throughout the system than if nodes of a low degree are encountered. However, if the graph contains one or more circuits then in some cases the nodes to which the program is offered will already be programmed. In such a case the

processor offering the

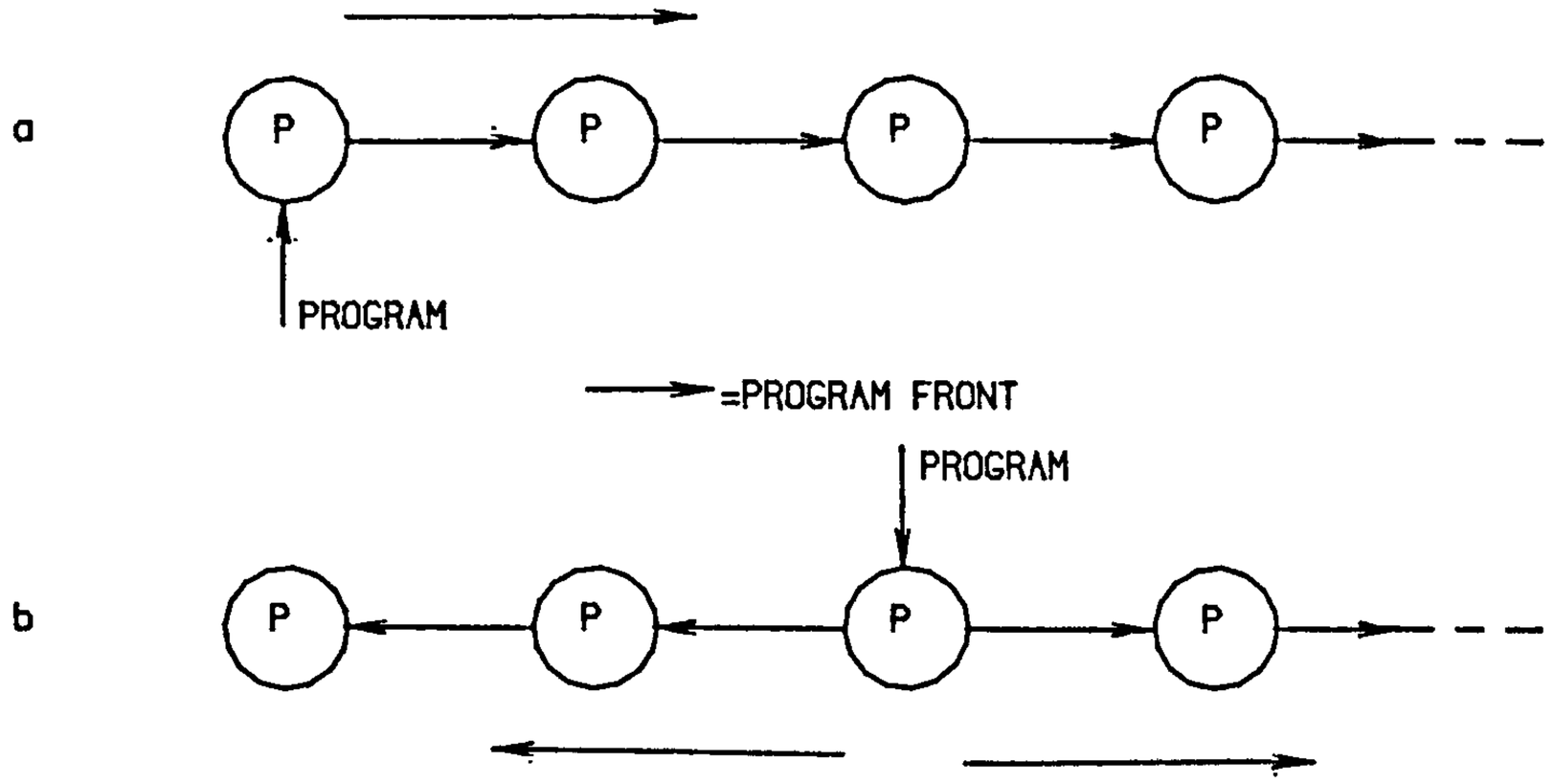


FIG. 6-1 PROPAGATION OF PROGRAM FRONTS
FOR TWO PROGRAM SUPPLY POINTS.

program would appear, as far as the propagation of the programming front is concerned, as though it had a lower degree than it actually has within the graph. As a result of this behaviour the program is never passed around a circuit so the program is distributed along one of the spanning trees of the graph, the particular spanning tree being dependant upon the time taken for processors to program each other and the consequent speed of propagation of the program down the different sub-trees.

6.2.2.3 Simultaneous Programming

Using the hardware described in chapter 5 it would be possible to send data down one or more links truly simultaneously. This capability opens up the possibility of sending a copy of the program to all of the nearest neighbour

un-programmed nodes simultaneously with a consequently more rapid dispersion of the program throughout the system. This mechanism was not implemented in practice because the simultaneous write facility is not generally available and also because during such a programming action the insertion of a verification of correct program transmission based upon some form of acknowledgement mechanism would be difficult to implement.

6.2.2.4 Error Correction Mechanisms

Since the program has to pass through D (where D is the diameter of the graph of the interconnection scheme of the processing nodes), and possibly more, serial links to reach some of the processing nodes the possibility of corruption of the program becomes significant. The program may also be corrupted while residing within the memory of a processing node, however this is considerably less likely than the corruption during transmission and may be dealt with using a suitable memory arrangement, e.g. a ninth parity bit on RAM devices.

Some form of error detection and correction of the transmission of program material down the serial links would be desirable. There are two main possibilities, error detection and re-transmission or error correction using redundant information or possibly a combination of both techniques, depending upon the degree of security required and the overhead that is acceptable.

One point to note it that error correcting codes, e.g. parity words, could easily be applied in the simultaneous program transmission scheme mentioned above, the same information being sent to all nodes being programmed, however because the interaction between the nodes is not on a one to one basis the nodes being programmed cannot easily request re-transmission of sections of the program without interfering with the programming of the other nodes.

6.2.2.5 Programming Algorithm Adopted

Bearing in mind the preceding points a programming mechanism was implemented on the Z-80 based system described in chapter 5. The programming system consisted of two main parts, the program loader residing in each processing node and the software required to send a program to the processor network in the correct form, the latter being merely a special case of the program loader, there being only output to deal with.

Since the system to be programmed was initially to be only small error detection and correction mechanisms were not implemented, however they could easily be inserted into the algorithm when the system reached such proportions that they were required. As mentioned above, the ability to send program material simultaneously to more than one link was not used as this is not a general feature of such processor schemes and the increased speed of programming was not required in the small scale system initially developed.

The algorithm used within the processing nodes written in 'pseudo code' was :

```
wait until the offer of a program copy is received;
send an acknowledgement;
receive the program;
for all links other than the source link do
  begin
  send an offer of a program copy;
  awaittimeout;
  if acknowledgement received then
    send the program;
  end;

enter the program;
```

Clearly some information other than just program material had to be sent to processors and this had to be distinguishable from the program material itself. A system of 'escape' codes were used, in this application the value FF_{16} was used as an escape value, however, the particular value chosen is unimportant. When a value of FF_{16} occurred in the program material this would be sent as FF_{16} FF_{16} which would allow distinction between this and one of the escape codes and would be correctly interpreted by the receiving processor, all single occurrences of FF_{16} indicating an escape sequence.

The codes used were:

FF_{16} FF_{16} : Indicates a value of FF_{16}

FF_{16} 00_{16} $Addr_L$ $Addr_H$: Sets the address to load data into and sets redirection to the memory

FF_{16} 01_{16} (SP-2) (SP-1): Pushes a value onto the stack

FF_{16} 02_{16} SP_L SP_H : Loads the Stack Pointer with a value

FF_{16} 03_{16} $LINK_{16}$: Sets the link to send data to and sets redirection to that link

FF_{16} 04_{16} : Performs a RETurn instruction

FF_{16} 05_{16} : Exits from the programming loop (Marks end of program)

FF_{16} 06_{16} $Addr_L$ $Addr_H$: Set the highest address required to forward to the next processor

FF_{16} 07_{16} $Addr_L$ $Addr_H$: Sets the lowest address required to forward to the next processor

The loading program was designed to have two modes of operation, one in which incoming bytes were stored in contiguous memory with the address to which the bytes were sent being incremented automatically and an alternative where all incoming data was sent to one of the serial links. Setting of the address or link to which data should be sent was performed using the sequences FF_{16} 00_{16} and FF_{16} 03_{16} respectively followed by the required data. The sequence FF_{16} 01_{16} followed by a 16 bit value pushed a value onto the stack, this in combination with the sequence FF_{16} 04_{16} could be used to jump to a particular location, also, when the program loop had been left and any neighbour processors programmed a RETurn would automatically be made to the address on the top of the stack. The sequence FF_{16} 02_{16} followed by a 16 bit value would position the stack pointer where required. The sequences FF_{16} 06_{16} and FF_{16} 07_{16} followed by a 16 bit address indicated the top and bottom respectively of the region of memory that must be sent on to any processors requiring a copy of the program.

6.2.2.6 Differing Processor Functions

The technique of mutual programming is considerably more convenient than programming each node individually and it would be desirable to apply this same technique to the case where different functions are to be carried out by different processing nodes. To some extent it is possible to do this. The programs executed in the nodes may be made to be dependant upon the position of the processing node within the computing structure. This can be achieved by sending the same program to all of the processing nodes but upon commencement of execution

of the program having the program perform interactive 'enquiries' of its neighbours (if any) to determine the type of situation in which it finds itself, possibly determining if any special devices are connected (e.g. disc units) in the process. Depending upon the results of this then going on to initialise and execute the appropriate section of code within the program. For example, in the cylindrical arrangement of stacked rings proposed in this thesis the processing nodes could perform a simple test to determine whether the node lies at the bottom of the stack (in which case no data should be sent further down) or at the top of the stack and if so whether any input of output devices are connected or not (and consequently enable or disable the sections of code to input or output data as appropriate).

A very simple example of such a program could be :

```
enquire_lower_node;
enquire_upper_node;
if not lower_node and not upper_node then
  single_ring_code;
if     lower_node and     upper_node then
  inside_layer_code;
if     lower_node and not upper_node then
  top_layer_code;
if not lower_node and     upper_node then
  bottom_layer_code;
```

6.3 Data Processing

The four algorithms for the use of the cylindrical stack of processors for processing of event data in a homogeneous fashion, the study of which by simulation has been described in chapter 4, were implemented on the hardware described in chapter 5. Four separate event processing programs were used, these were supplied with data by driver programs serving the

function of the supply of data by an event manager, the programs involved are to be found in appendix 15. Only three processors were available and it was only possible to supply data at one point. Though this did limit the extent to which the behaviour of the system for differing shapes could be explored it did allow the correct operation of such a system to be verified or denied. Since the simulations carried out used a globally synchronised communication scheme it would be possible for a deadlock condition or unreasonable behaviour of a completely asynchronous system to have been overlooked.

6.3.1 Data Format

Data from an 'event' was encoded in a simple packet structure utilising a header value, followed by the length of the data, followed by the data itself, a simple example being shown in fig 6.2. The same structure was also used to carry results in the reverse direction.

FIG. 6-2 EVENT PACKET STRUCTURE.

This 'packaging' offers no fault detection or recovery

but for the purposes of verifying the operation of the homogeneous computing structure has proved satisfactory. It may in a more complete implementation be advantageous to distinguish between incoming data packets and outgoing results packets but due to the policy of sending results in the reverse direction to data the type of packet may be determined from its direction of travel within the structure.

6.3.2 Data Supply

Data was supplied to the system by a driver program serving the functions of both the event manager and the mainframe used for the accumulation of data. A simple scheme was used, the characters 'a' to 'g' being sent as raw events and the uppercase characters 'A' to 'G' returned as processed events. The selection of printable characters for the data made for easier debugging during the initial testing stages. Events were supplied to the processor as fast as the processor would accept them, not only to determine the maximum processing rate but to illustrate the behaviour of the system under heavy loading, particularly with regard to deadlock or under utilisation of any of the processors.

6.3.3 Data Display

During debugging a very simple data display consisting of the characters returned by the processor was presented on the screen of the machine accumulating the results. The continuous transfer of characters in this fashion was a potential bottleneck so a system of storing counts of the results

received and displaying the cumulative values at infrequent intervals was also implemented. This is distinctly different from the accumulation of results within the processing nodes before returning cumulative values.

6.3.4 Processing Behaviour

The results indicate the behaviour of a homogeneous processing scheme and allowed the **relative** behaviour of the four communication algorithms tested to be evaluated.

Though no formal proof of the liveness of the processors was undertaken no deadlock was observed in the implementation described here and the behaviour of the four algorithms tested was found to be virtually identical. This behavioural equivalence was due to the continuous cycling of the program masking any apparent priority.

6.3.4.1 Distribution Preservation

One of the programs tested gave a significant weighting of processing speed to one type of event as compared with other events (see appendix 15), the events sent to the processor were evenly distributed and for the processed data to be meaningful in a real implementation the distribution of supplied events must be preserved regardless of differing processing times of events. The distribution of processed events was indeed observed to be uniform despite the bias toward some types of event in all of the topologies examined verifying the validity of this approach to processing event data.

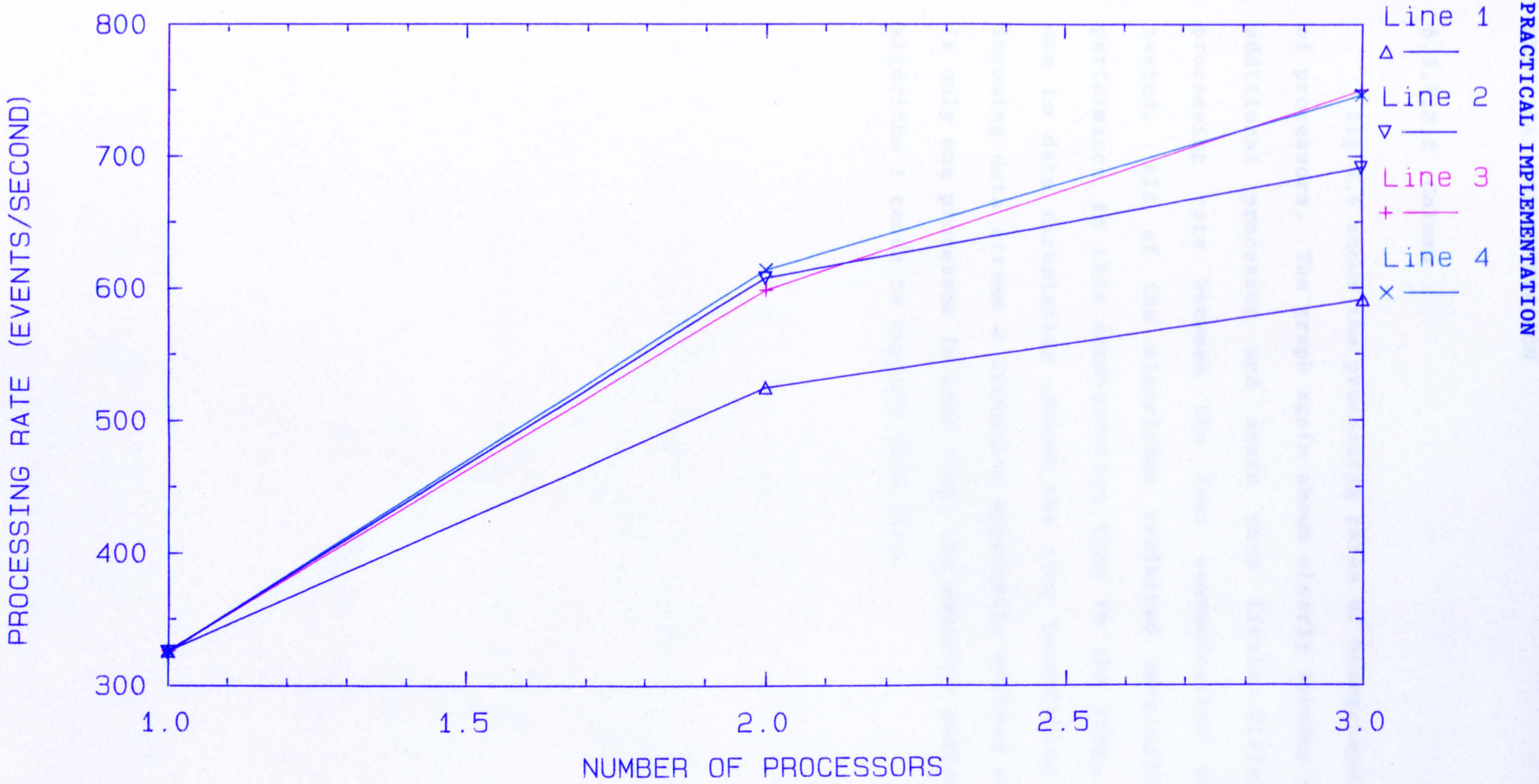
6.3.4.2 Processing Speedup

The rate of processing in events per second was measured over 5000 events for a variety of structures made up of the three processors available. It should be noted that events with a fairly long processing time were represented and that the processing rates do not relate to the target figure of 100 000 events/second aimed for in a final implementation but indicate that speedup is achieved and that processors will distribute the workload amongst themselves. Algorithm 1 produced a comparatively poor throughput, this probably results from the statement order giving some weighting to horizontal communication producing the 'data shuffling' effect where data circulating around the ring interferes with the intake of data, wasting computing cycles, as was observed in the simulations.

6.3.4.2.1 Ring

Fig 6.3 shows the processing rates of homogeneous rings of processors. The graph shows clearly speedup with each additional processor and very little difference in processing rate between the four communication algorithms tested.

FIG. 6-3 PROCESSING RATE FOR A RING OF PROCESSORS RUNNING THE FOUR COMMUNICATION ALGORITHMS.



6.3.4.2.2 Column

Fig 6.4 shows the processing rates of homogeneous columns of processors. The graph again shows clearly speedup with each additional processor and again very little difference in processing rate between the four communication algorithms tested. All of the algorithms exhibited marginally better performance in this configuration than in the ring, possibly due to data circulating around the ring interfering with the incoming data stream a situation apparently avoided when there is only one processor in each ring, the superior performance of algorithm 1 tends to support this view.

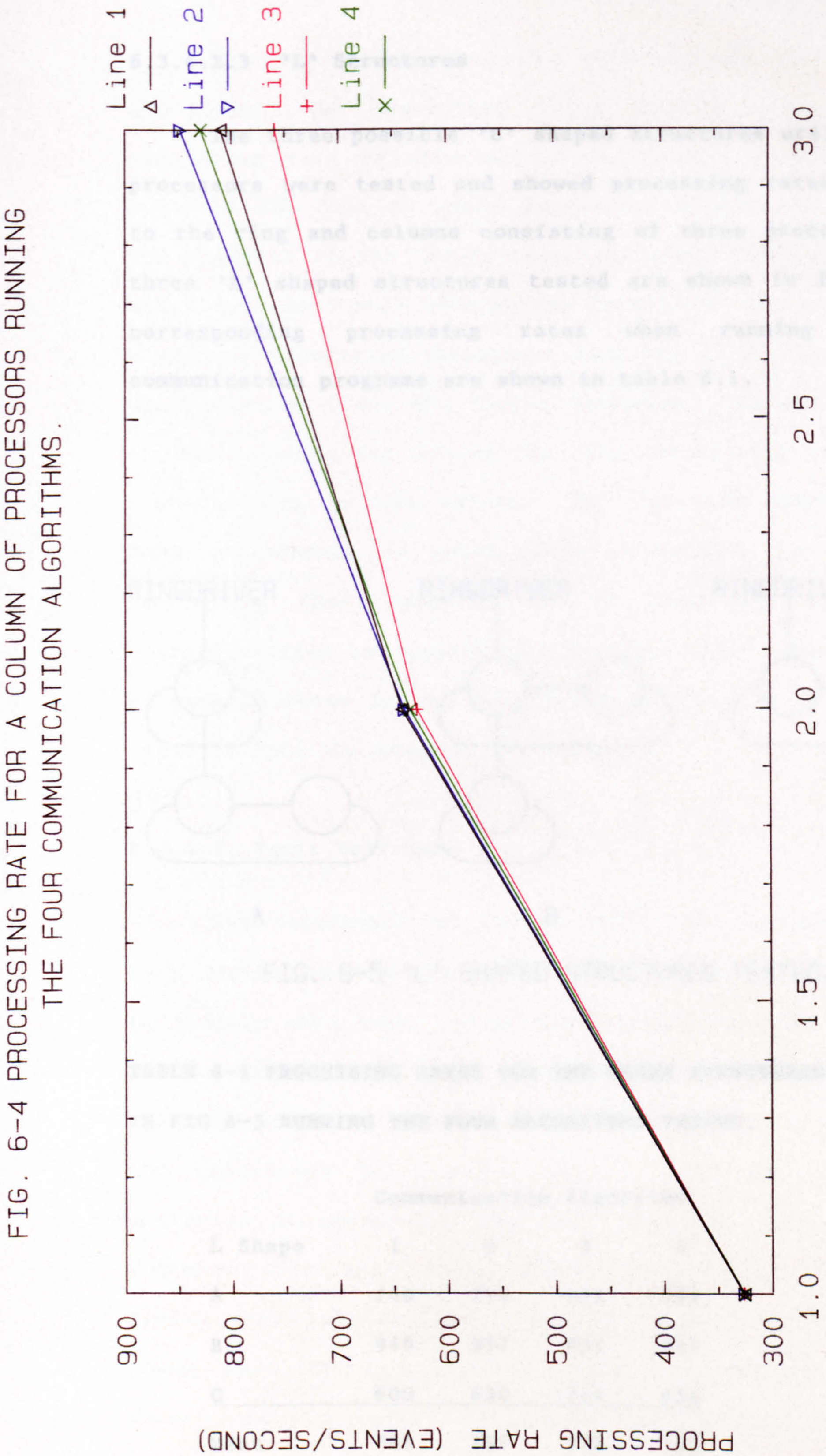


FIG. 6-4 PROCESSING RATE FOR A COLUMN OF PROCESSORS RUNNING THE FOUR COMMUNICATION ALGORITHMS.

6.3.4.2.3 'L' Structures

The three possible 'L' shaped structures utilising three processors were tested and showed processing rates comparable to the ring and columns consisting of three processors. The three 'L' shaped structures tested are shown in fig 6.5, the corresponding processing rates when running the four communication programs are shown in table 6.1.

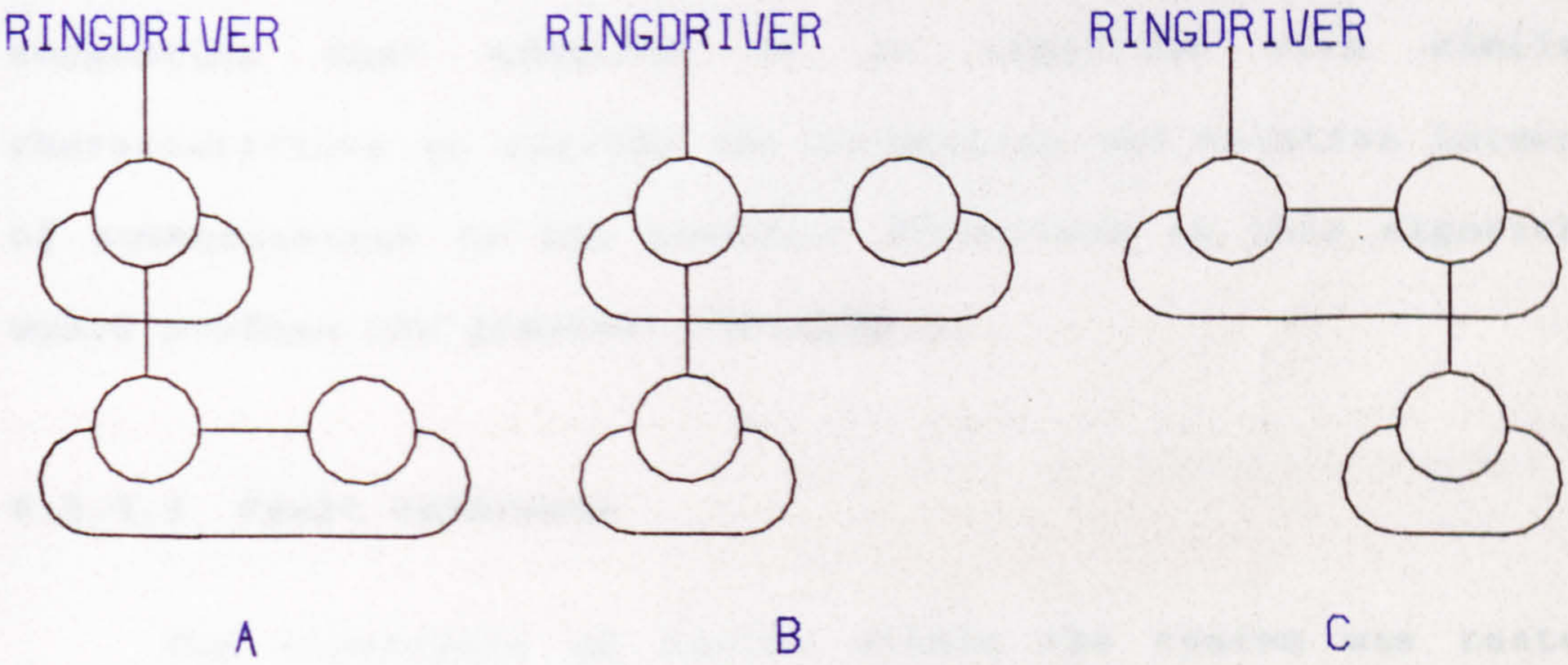


FIG. 6-5 'L' SHAPED STRUCTURES TESTED.

TABLE 6-1 PROCESSING RATES FOR THE THREE STRUCTURES ILLUSTRATED IN FIG 6-5 RUNNING THE FOUR ALGORITHMS TESTED.

	Communication Algorithm			
L Shape	1	2	3	4
A	740	794	671	721
B	949	937	931	934
C	600	630	714	636
Mean	763	787	772	763

The processing rates are very similar for all of the algorithms, the algorithms being largely equivalent. The processing rate is affected by the number of data transfers required for data to reach particular processors (related to the diameter of the interconnection graph) configuration B in which data need only undergo one forwarding operation provides a greater processing throughput than the other configurations. Configuration C has the lowest throughput, believed to be due to data circulating around the ring interfering with vertical communication to some extent. The algorithm showing the best mean performance in these three structures is algorithm 2 suggesting that adoption of an algorithm with similar characteristics as regards the priorities and relative latency of communication in the possible directions as this algorithm would produce the greatest throughput.

6.3.4.3 Fault Tolerance

The occurrence of faults within the system was tested very simply by removing links from the system while running. No timings were taken though processing rates similar to those of one of the structures described above would be expected. The possibility of communication in the reverse direction upon the occurrence of a fault was not explored and as such isolation of processors from the supply of data was readily achieved with only three processors. In the case of the column, the ring and the three 'L' shaped structures it was found that those processors remaining connected so as to receive data as links were removed would contribute processing power to the system.

6.3.4.4 Lost Events

On initiation of the data supply some events were lost, these were events sent to un-connected lines of a processor or results of processed events sent to un-connected lines of a processor. After the initial loss, which can be avoided by incorporating a test for un-connected lines before processing commences in the communication scheme, no further lost events were observed; the events were believed to be lost on the removal of a link in the demonstration of faults tolerance but due to the difficulty of making observations in the environment of ongoing processing no observations of these were made.

6.4 Distributed Depth First Search Algorithm

6.4.1 Initial Implementation

The depth first search algorithm developed in chapter 3 was expressed in 'pseudo code' as :

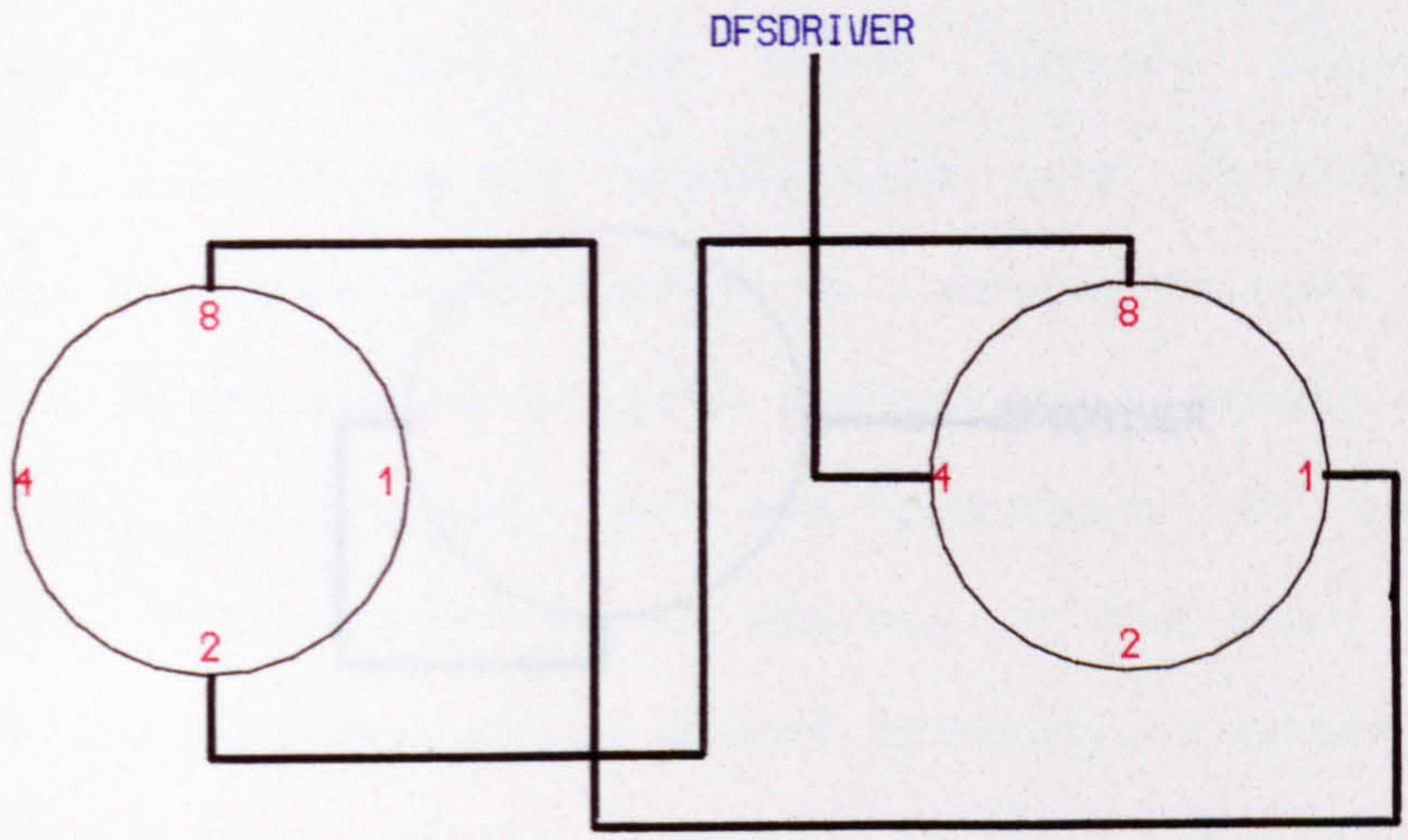
```
procedure ddfs;
  var now, val:integer;

  procedure visit_call(t:link);
  begin
  send now to visited node;
  while waiting
    if val_requested  then
      send val to requesting link;
  receive new now from visited node;
  end;

  procedure visit answer(t:link);
  begin
  receive value of now;
  for t:=1 to 4 do
    begin
    if valreq(t) = 0 then
      visit call(t);
    end;
  return new value of now;
  end;

begin
repeat
  begin
  wait for something;
  if visited then
    visit answer;
  if val_requested then
    send val to requesting link;
  end
until forever;
end;
```

This algorithm had to be coded in a form suitable for down loading into the processors available, in the modified C programming language developed. Since bytewise communication was used in the programming system this limited the number of processing nodes that the implementation of the algorithm could correctly scan, this maximum was further reduced by the need to allocate some of the 256 values as special codes. Though the actual number of processing nodes possible was limited this does not invalidate the test of the depth first search algorithm as longer messages carrying larger values could be simply incorporated into this framework.

For the algorithm to produce output more amenable to debugging and testing printable characters were used as the special codes and values. The 'values' of the visited nodes were represented by lower case characters 'a' (unvisited) through to 'z' (maximum) giving a maximum number of nodes of 25. The values of the links were sent as the characters '1', '2', '4' and '8' these representing the actual addresses of the links within the processing node.

A driver program to send the required codes to initiate the depth first search and to collect the resulting data and print this as an adjacency matrix was written in C. This program and the depth first search programs are to be found in appendix 16.

Though only three processing nodes were available for testing the algorithm this was sufficient to test the propagation of the algorithm through the processing system and the correct functioning of the algorithm with various connection patterns. Some connection patterns tested and their resulting labels and adjacency matrix are shown below in fig 6.6, the latter of these is obviously spurious data due to the incorrect scanning of a self-loop.

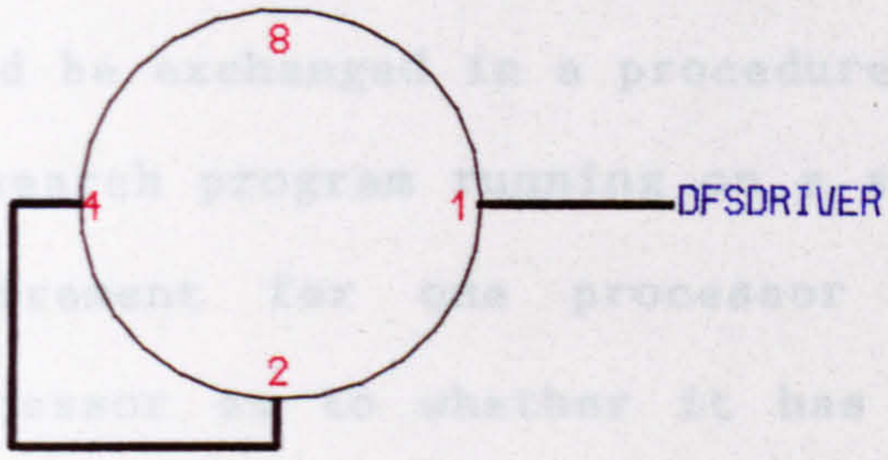


Total Number of Connected Nodes = 2

	FROM								
	0	1	2	3	4	5	6	7	8	9
TO										
0	0	4	0	0	0	0	0	0	0	0
1	0	0	82	0	0	0	0	0	0	0
2	0	18	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0

Note: Vertex 0 represents the connection point to the graph
The numbers indicate which link forms the connection(s)

FIG. 6-6a PROCESSOR INTERCONNECTION PATTERN AND THE RESULTING ADJACENCY MATRIX USING THE DDFS ALGORITHM.



Total Number of Connected Nodes = 1

	FROM									
	0	1	2	3	4	5	6	7	8	9
TO										
0	0	1	0	0	0	0	0	0	0	0
1	4	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0

Note: Vertex 0 represents the connection point to the graph
The numbers indicate which link forms the connection(s)

FIG. 6-6b PROCESSOR INTERCONNECTION PATTERN AND THE

RESULTING ADJACENCY MATRIX USING THE DDFS ALGORITHM.

6.4.2 Multiprocessing Implementation

The depth first search algorithm itself is inherently simple, being extended to a distributed form merely by separating visit calls and visit answers into separate processors and initiating appropriate data exchanges between processors as would be exchanged in a procedure call and return of a depth first search program running on a single processor. However the requirement for one processor to be able to interrogate a processor as to whether it has been visited or not, and its consequently assigned identity, regardless of its state complicates matters somewhat in that provision must be made to respond to such requests at all times. This ability to respond is essential to allow both circuits and self-loops within graphs to be dealt with correctly.

To retain the simplicity of the depth first search and avoid the complication of responding to the requests for the processor identity it would be possible, using the message passing multiprocessing system developed to separate these two functions. This could be achieved by intercepting the requests for identity and responding to them without involving the depth first search program. To achieve this a process would be required to 'filter' all incoming data streams to intercept the Value Request Codes and a process would be required to combine the responses to Value Request Codes and the data streams from the depth first search program. These processes are shown in fig 6.7.

The C multiprocessing program represented in this diagram is to be found in appendix 16. The program was tested in the same fashion as the previously discussed depth first search program and was found to function in the same way, however it produced the correct output on encountering self-loops, as shown in fig 6.8.

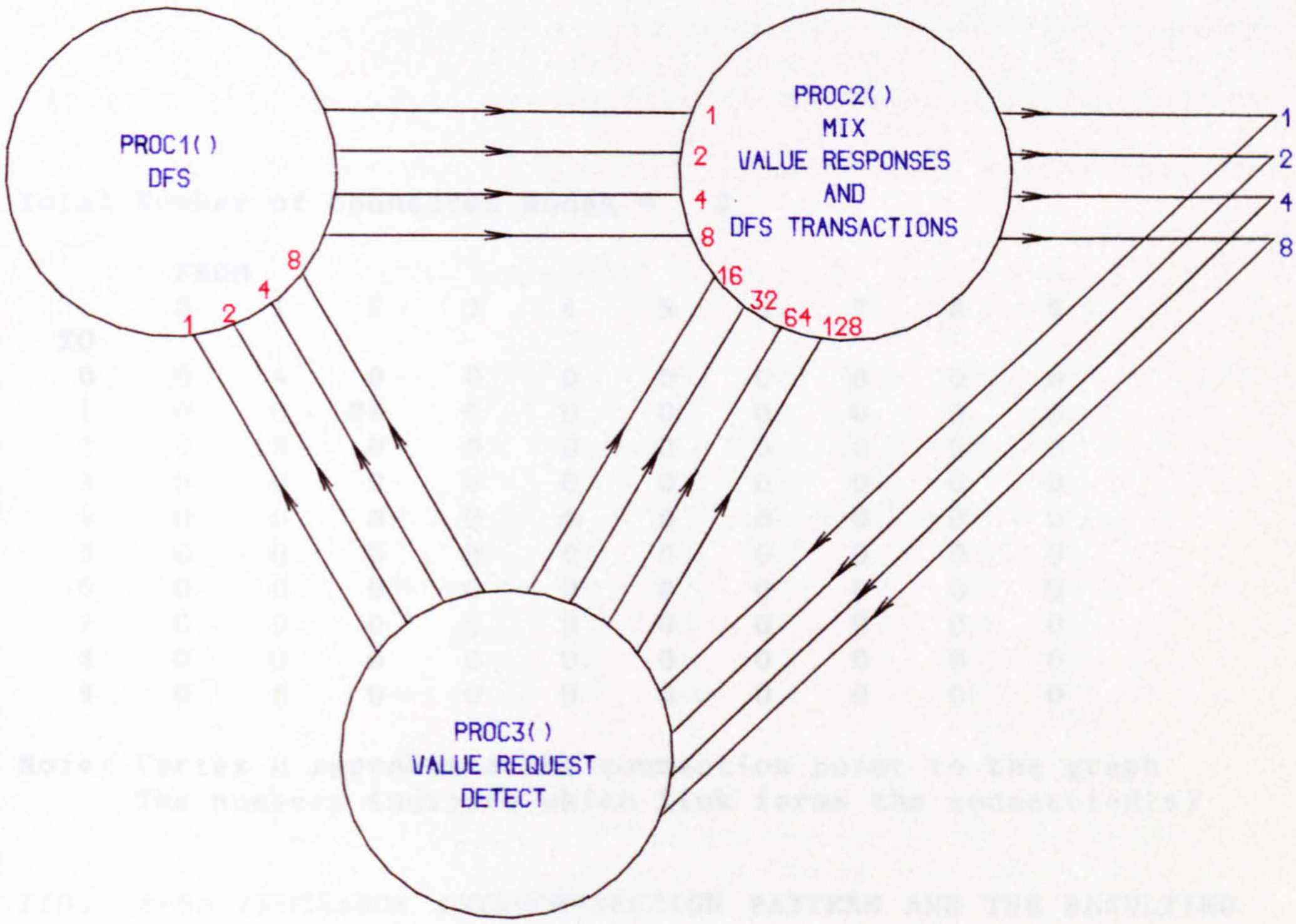
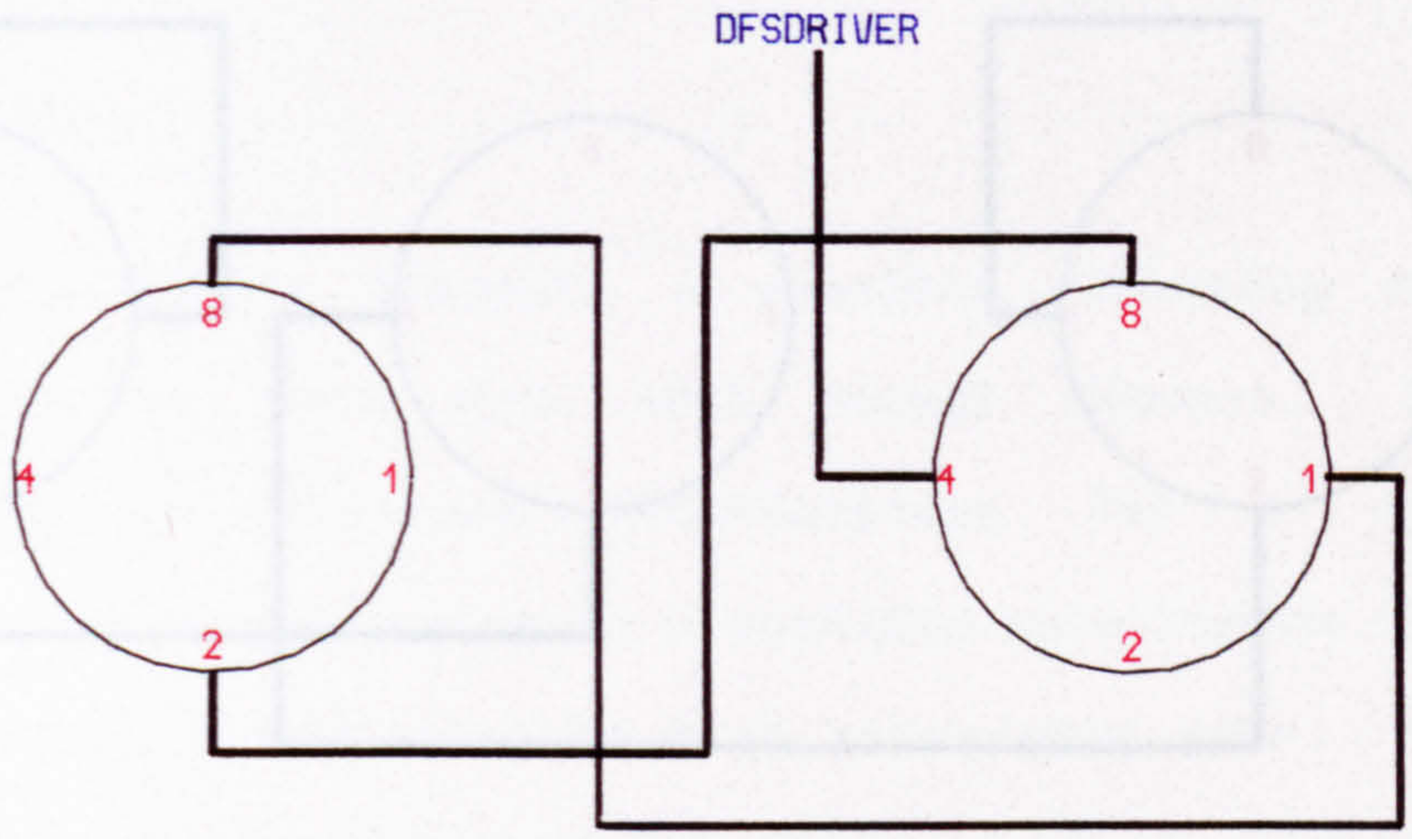


FIG. 6-7 PROCESS MODEL OF THE MULTIPROCESSING DEPTH FIRST SEARCH.

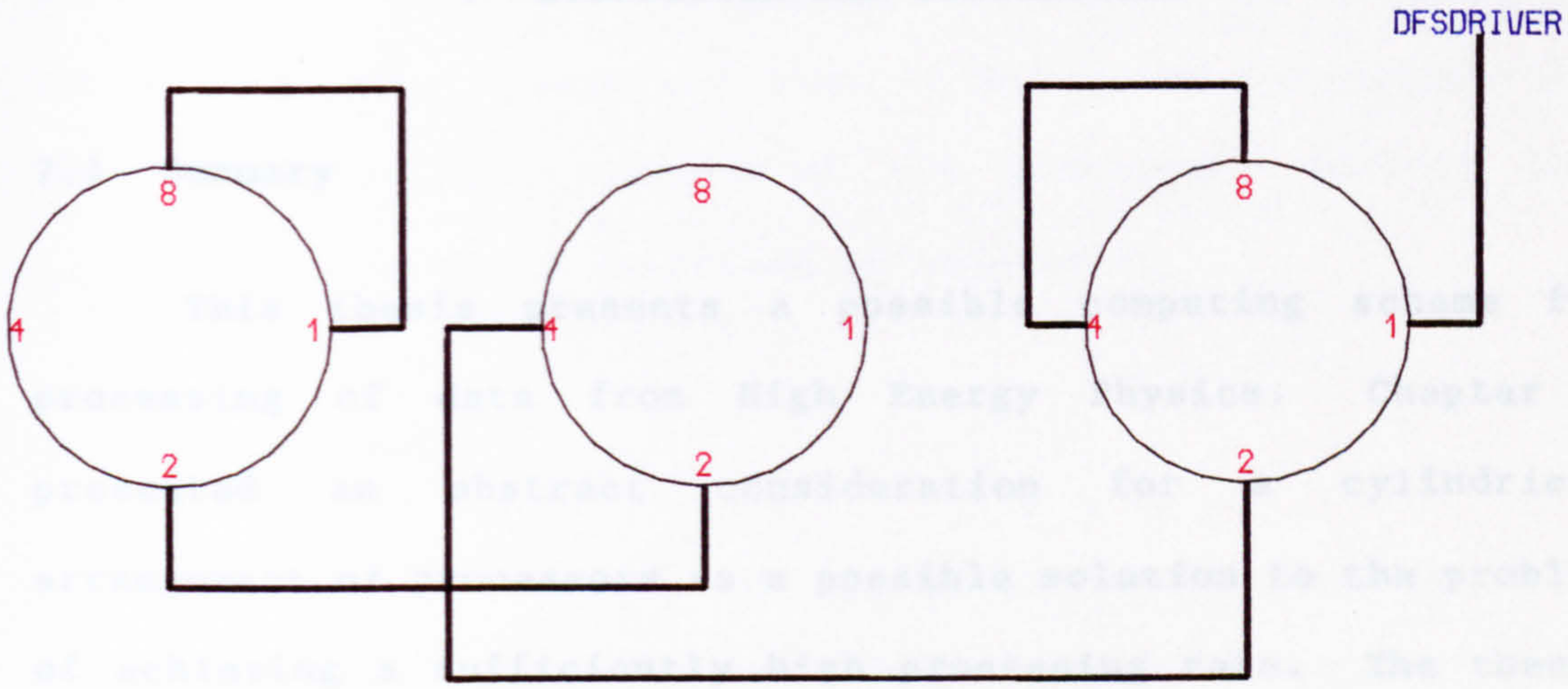


Total Number of Connected Nodes = 2

	FROM									
	0	1	2	3	4	5	6	7	8	9
TO										
0	0	4	0	0	0	0	0	0	0	0
1	0	0	82	0	0	0	0	0	0	0
2	0	18	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0

Note: Vertex 0 represents the connection point to the graph
The numbers indicate which link forms the connection(s)

FIG. 6-8a PROCESSOR INTERCONNECTION PATTERN AND THE RESULTING ADJACENCY MATRIX USING THE DDFS (MULTIPROCESSING) ALGORITHM.



Total Number of Connected Nodes = 3

TO \ FROM	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	0	0	0	0
1	0	48	4	0	0	0	0	0	0	0
2	0	2	0	2	0	0	0	0	0	0
3	0	0	2	18	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0

Note: Vertex 0 represents the connection point to the graph
The numbers indicate which link forms the connection(s)

FIG. 6-8b PROCESSOR INTERCONNECTION PATTERN AND THE RESULTING ADJACENCY MATRIX USING THE DDFS (MULTIPROCESSING) ALGORITHM.

CHAPTER 7

7 EVALUATION AND CONCLUSIONS

7.1 Summary

This thesis presents a possible computing scheme for processing of data from High Energy Physics. Chapter 3 presented an abstract consideration for a cylindrical arrangement of processors as a possible solution to the problem of achieving a sufficiently high processing rate. The theory developed as a predictor of behaviour is of an extremely simple nature and possible rather crude; however, as the nature of the processing to be performed was not rigidly defined, varying depending upon the experimentor's requirements, the development of a more exact model was not readily justified. The structure proposed was that of a cylindrical connection of processors communicating asynchronously in a data driven scheme. Simulations were carried out using a state machine model of the processors which showed generally good agreement with the behaviour predicted from the model.

Two schemes of use for this structure were proposed, a non-homogeneous scheme in which data would be processed at a particular node and a homogeneous scheme in which data would be processed at any node. The homogeneous scheme was demonstrated to avoid many of the mapping and routing problems of the non-homogeneous scheme and hence was taken as the subject of further development.

Processor hardware and software support sufficient to test the correct functioning of the homogeneous scheme developed is described in chapter 5 and its use to test the above processing scheme and also an algorithm for determining the interconnection pattern of the processors derived from depth first search is described in chapter 6.

7.2 Evaluation

The variable nature of the amount of processing to be performed meant that the final throughput in events/second would be dependant upon the processing required by the experimentor, so no final figure of throughput could be obtained. However, the general behaviour of the system has been demonstrated for various conditions of input data by simulation and it has also been shown that the throughput of the system can be predicted to a reasonable degree of accuracy from the model developed in chapter 3 provided that some basic parameters of the processors making up the system are known. The correct functioning of the system with respect to producing results that correctly represented the distribution of data events and the ability to continue processing at a reduced rate despite faults has been demonstrated both by simulation and, for a small number of processors, in a hardware implementation. In addition to the above a 'distributed depth first search' algorithm has been developed to allow the connection topology to be verified and possibly perform some testing of the processors. The algorithm has been stated in a very general style facilitating coding in a variety of programming languages.

7.3 Further Research

This thesis presents the concepts behind a computing structure and illustrates some verification that the structure is functionally correct. There remains a considerable amount of research and development work before a final design could be realised.

7.3.1 Processor Selection

The computing scheme may be realised utilising one or more of the large number of microprocessors available today and some investigation into the relative merits of the microprocessors for this application would be required before selection could be made. Zanella[162] suggests that the HEP community have informally adopted the following microprocessors; 6809 for control applications and the 68000 series of microprocessors for more compute intensive tasks. These processors could prove suitable and their adoption would be in keeping with the informally adopted standard. The structure is tetravalent and suitable for implementation using TRANSPUTERS, their high speed and multiprocessing system support makes them a strong candidate for the construction of the system. The high price (1988) must be borne in mind and a more cost effective solution may be found through the use of a greater number of slower but cheaper microprocessors; this approach may also offer improved fault tolerance through a greater division of the workload and greater path redundancy in the interconnection pattern. Development of methods of estimating or measuring the values of β_{cmax}, K_r and β_{phys} for

various node designs would be important to allow the model to be used to make predictions of performance and behaviour.

7.3.2 Node Structure

Closely related to the selection of a microprocessor from which to construct the system would be the design of the internal structure of each processing node. The dual functionality of a node, ie the routing and the processing of data suggests a node with two distinct communicating sections, perhaps utilising both the 6809 and 68000 series of microprocessors. Since a large part of the software is independant of the application, such as the loading and routing software this could be retained in EPROM leaving only the experimentor's processing algorithm to be loaded. An element of the consideration of node structure would possibly include the requirement and possibility of incorporating fault detection hardware into the processing nodes.

7.3.3 Data Supply

The feeding of data into the system has not been fully considered, if a single data source is to be fed into the system it would be possible to use a multiplexer to divide a high bandwidth data stream into several low bandwidth links. If one of the data links has a sufficiently high bandwidth to take all of the data the use of a multiplexer may be avoided, the communication links being used to distribute data through the system. The possible use and design of such a multiplexer may be the subject of further research.

7.3.4 Data Retrieval

One aspect not thoroughly explored here is the retrieval of processed data from the computing structure. Though an algorithm has been presented, this being the reverse of the algorithm for the un-processed data the possibility of grouping processed events into 'bundles' to reduce the results traffic has not been explored. The inter-relationship between the factors of node storage requirement, results traffic, the possibility of large numbers of lost events should a node fail before a 'bundle' contained therein is transmitted and possible (temporary) distortion of the displayed spectra if large numbers of similar events are awaiting transmission would require investigation.

7.3.5 Programming Error Correction

The possibility of mutual identical programming has been demonstrated, however no error correction was incorporated. As any transmission errors would propagate to any further processors programmed and since any errors would be compounded a strong error correction mechanism would be desirable in this context.

7.3.6 Algorithm Development

Four communication algorithms have been isolated and utilised in both simulation and a hardware implementation, other algorithms could be developed with a view to improved behaviour, especially as regards fault-tolerance. In relation to this work the occurrence of 'processing shadows' behind

faulty nodes could be usefully evaluated, a Petri-Net technique would be useful to prove the correctness of the processor interactions and the development of a recoverable interaction scheme and reprogramming mechanism would be required to allow on-line replacement of faulty nodes. The algorithms used in the processing of events would be determined largely by the requirements of the experimentor though some consideration of general strategies (e.g. look-up table evaluation) would be desirable.

7.4 Final Implementation

The implementation of the computing structure as a real system would require the development of a considerable amount of hardware and software, not just for the computing engine itself but for interfaces device drivers etc. The loading algorithms and node communication programs would have to be written in an appropriate language. A user interface to the system would be required though this could be largely derived from that already in use since the use of the system would be unlikely to change.

7.5 Conclusions

The feasibility of using a multimicroprocessor to process data from High Energy Physics has been demonstrated. There remains considerable further work to be done before a final implementation could be realised.

That is the theory that I have and which is mine and what it is too.

Anne Elk (Miss) (circa 1970).

REFERENCES

[1] ABDEL KADER A.A., 'Design of the Systolic Array OCSAMO', *VLSI and Computers*, Proceedings of First International Conference on Computer Technology, Systems and Applications, IEEE Comput Soc Press, 1987, ISBN 0-8186-0773-4.

[2] ALMASI G.S., 'Overview of Parallel Processing', *Parallel Computing*, Vol 2, 1985.

[3] AMDAHL G.M., 'Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities', AFIPS Conference Proceedings, Thomson, Washington, DC, USA, 1967.

[4] ANDERSON T. and LEE P.A., *Fault Tolerance: Principles and Practice*, Prentice-Hall International, 1981, ISBN 0-13-308254-7.

[5] ANNARATONE M. *et al.*, 'The Warp Computer: Architecture, Implementation and Performance', *IEEE Transactions on Computers*, Vol C-36, No 12, December 1987.

[6] ANNUNZIATA M. *et al.*, 'A Daisy Architecture for the Multiprocessor Real Time Data Acquisition System of the Thor Tokomak Experiment', *Microprocessing and Microprogramming*, Vol 17, 1986.

[7] ANNUNZIATA M. *et al.*, 'AMDAS - An Advanced Microprogrammed Data Acquisition System: A first evaluation prototype', *Microprocessing and Microprogramming*, Vol 18, 1986.

[8] ANSADE Y. *et al.*, 'Algorithms Dedicated to a Network of Asynchronous Cells', *Parallel Algorithms and Architectures*, 1986.

[9] ANZALONE A., GIUSTOLISI F. and SCOLLO G., 'Distributed Processing in a Nuclear Data Acquisition System', *Performance of Data Communication Systems and their Applications*, North-Holland, Amsterdam, 1981.

[10] ARAVENA J.L. and PORTER W.A., 'Nonplanar Array Processing', *VLSI and Computers*, Proceedings of First International Conference on Computer Technology, Systems and Applications, IEEE Comput Soc Press, 1987, ISBN 0-8186-0773-4.

[11] ASKEW C.R., 'Parallel Processing using Transputers or OCCAM (or both)', Proceedings of the 1986 CERN School of Computing, CERN, Geneva, April 1987.

[12] AUGIN M. and BOERI F., 'The OPSILA Computer', *Parallel Algorithms and Architectures*, Proceedings of the International Workshop on Parallel Algorithms and Architectures, North-Holland, Netherlands, 1986, ISBN 0-444-70104-4.

[13] AUGUIN M. *et al.*, 'Experience Using a SIMD/SPMD Multiprocessor Architecture', *Microprocessing and Microprogramming*, Vol 21, 1987.

[14] BADR H., GELERNTER D. and PODAR S., 'An Adaptive Communication Protocol for Network Computers', *Performance Evaluation Reviews*, Vol 13, No 2, August 1985.

[15] BALASUBRAMANIAN V. and BANERJEE P., 'A Fault Tolerant Massively Parallel Processing Architecture', *Journal of Parallel and Distributed Computing*, Vol 4, Part 4, Aug 1987.

[16] BARSI F., GRANDONI F. and MAESTRINI P., 'A Theory of Diagnosability of Digital Systems', *IEEE Transactions on Computers*, Vol C-25, No 6, June 1976.

[17] BEETEM J., DENNEAU M. and WEINGARTEN D., 'The GF11 Supercomputer', Twelth Annual International Symposium on Computer Architecture, IEEE Comput Soc Press, 1985, ISBN 0-8186-0634-7.

[18] BELL T.E., 'Optical Computing: A Field in Flux', *IEEE Spectrum*, August 1986.

[19] BERARDI B., BOMBI F. and FERMANI G., 'Development Status of the FTU Control System', *Fusion Technology 1986*, Volume 2, Proceedings of the fourteenth Symposium, Pergamon Press, 1986.

[20] BERKLING K.J., 'A Computing Model Based on Tree Structures', *IEEE Transactions on Computers*, Vol C-20, No 4, April 1971.

[21] BERNSTEIN A.J., 'Analysis of Programs for Parallel Processing', *IEEE Transactions on Electronic Computers*, Vol EC-15, No 5, October 1966.

[22] BHATT S.N. and LEISERSON C.E., 'How to Assemble Tree Machines', *Communications of the ACM*, 1982.

[23] BUEHRER R.W., 'Emulation of a Parallel Codeblock Dataflow Processor', *Microprocessing and Microprogramming*, Vol 21, 1987.

[24] CAMPBELL M.L., 'Static Allocation for a Data Flow Mulitprocessor', Proceedings of the 1984 International Conference on Parallel Processing, IEEE Comput Soc Press, Washington, DC, USA, 1985.

[25] CARLSON W.W. and HWANG K., 'Algorithmic performance of Dataflow Multiprocessors', *IEEE Computer*, December 1985.

[26] CHRIST N.H. and TERRANO A.E., 'A Very Fast Parallel Processor', *IEEE Transactions on Computers*, Vol C-33, No 4, April 1984.

[27] CIN M.D. and FLORIAN F.H., 'Analysis of a Fault-Tolerant Distributed Diagnosis Algorithm', *FTCS 15*, Proceedings of Fifteenth Annual International Symposium on Fault-Tolerant Computing, IEEE Comput Soc Press, 1985, ISBN 0-8186-0618-5.

[28] CITTOLIN S., 'The UA1 VME Data Acquisition System', Proceedings of the 1986 CERN School of Computing, CERN, Geneva, April 1987.

[29] CONRAD M., 'On Design Principles for a Molecular Computer', *Communications of the ACM*, Vol 28, No 5, May 1985.

[30] CORSINI P. and PRETE C.A., 'Architecture of the MuTeam System', *IEE Proceedings*, Vol 134, Pt E, No 5, September 1987.

[31] CREUTZ M., 'High-Energy Physics', *Physics Today*, May 1983.

[32] DAHBURA A.T., SABNANI K.K. and KING L.L., 'The Comparison Approach to Multiprocessor Fault Diagnosis', *IEEE Transactions on Computers*, Vol C-36, No 3, March 1987.

[33] DALLY W.J. *et al.*, 'Architecture of a Message-Driven Processor', Fourteenth Annual International Symposium on Computer Architecture, IEEE Comput Soc Press, Washington, 1987.

[34] DARESBURY LABORATORY, User *Guide to Data Acquisition: Hardware*, Daresbury Laboratory, Daresbury, Warrington, UK, May 1984.

[35] DARESBURY LABORATORY, *User Guide to Data Acquisition: Multiparameter Event Monitoring*, Daresbury Laboratory, Daresbury, Warrington, UK, October 1983.

[36] DARESBURY LABORATORY, *User Guide to Data Acquisition: Software*, Daresbury Laboratory, Daresbury, Warrington, UK, June 1984.

[37] DARESBURY LABORATORY, *User Guide to Data Acquisition: Data Analysis*, Daresbury Laboratory, Daresbury, Warrington, UK, September 1984.

[38] DARESBURY LABORATORY, *User Guide to Data Acquisition: Introduction*, Daresbury Laboratory, Daresbury, Warrington, UK, May 1984.

[39] DEITEL H.M., *An Introduction to Operating Systems*, Revised First Edition, Addison-Wesley, 1984, ISBN 0-201-14502-2.

[40] DENNIS J.B., 'Data Flow Computation', *Control Flow and Data Flow: Concepts of Distributed Programming*, Proceedings of NATO Advanced Institute International Summer School, Springer-Verlag, Berlin, Germany, 1985, ISBN 0-38-713919-2.

[41] DEO N., *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall, 1974, ISBN 0-13-363473-6.

[42] DESPAIN A.M. and PATTERSON D.A., 'X-Tree: A Tree Structured Multi-Processor Computer Architecture', Proceedings of the Fifth Annual Symposium on Computer Architecture, 1978.

[43] DIETZ H. and KLAPPHOLZ D., 'Refined C: A Sequential Language for Parallel Programming', Proceedings of the 1984 International Conference on Parallel Processing, IEEE Comput Soc Press, Washington, DC, USA, 1985.

[44] DIJKSTRA E.W., 'Cooperating Sequential Processes', *Programming Languages*, Academic Press, New York, 1968.

[45] DU D.Z., HSU D.F. and HWANG F.K., 'Doubly Linked Ring Networks', *IEEE Transactions on Computers*, Vol C-34, No 9, September 1985.

[46] DUBIELEWICZ A. *et al*, 'An Architecture of a Stand-Alone Multiprogramming Data Flow System', *Microcomputers, Usage and Design*, Elsevier Science Publishers, EuroMicro 1985.

[47] ECOSOFT, *Eco-C C Compiler*, Utility Software Package, Reference Manual, Ecosoft Inc, 1984.

[48] ENSLOW P.H., *Multi-Processors and Parallel Processing*, John wiley & Sons, 1974, ISBN 0-441-16735-5.

[49] ETTINGER J.E., 'Distributed Array Processors', *Parallel Processing, State of the Art Report*, Pergamon Infotech Limited, Maidenhead, UK, 1987, ISBN 0-08-034113-6.

[50] FENG T., 'A Survey of Interconnection Networks', *IEEE Computer*, December 1981.

[51] FLYNN M.J., 'Some Computer Organizations and Their Effectiveness', *IEEE Transactions on Computers*, Vol C-21, No 9, September 1972.

[52] FLYNN M.J., 'Very High-Speed Computing Systems', *Proceedings of the IEEE*, Vol 54, No 12, December 1966.

[53] FORTES J.A.B. and RAGHAVENDRA C.S., 'Gracefully Degradable Processor Arrays', *IEEE Transactions on Computers*, Vol C-34, No 11, November 1985.

[54] GAJSKI D.D. and PEIR J., 'Comparison of Five Multiprocessor Systems', *Parallel Computing*, Vol 2, 1985.

[55] GELENBE E. *et al.*, 'A Performance Model of Block Structure Parallel Programs', Parallel Algorithms & Architectures, Proceedings of the International Workshop on Parallel Algorithms & Architectures, North-Holland, Amsterdam, 1986.

[56] GINOSAR R. and HILL D.D., 'Design and Implementation of Switching Systems for Parallel Processors', Proceedings of the 1984 International Conference on Parallel Processing, IEEE Comput Soc Press, Washington, DC, USA, 1985.

[57] GLENDINNING I. and HEY A., 'Transputer Arrays as Fortran Farms for Particle Physics', *Computer Physics Communications*, Vol 45, 1987.

[58] GOKHALE M.B., 'Macro vs Micro Dataflow: A Programming Example', Proceedings of the 1986 International Conference on Parallel Processing, IEEE Comput Soc Press, Washington, DC, USA, 1986.

[59] GOODMAN J.W. *et al.*, 'Optical Interconnections for VLSI Systems', *Proceedings of the IEEE*, Vol 72, No7, July 1984.

[60] GOULDING F.S and HARVEY B.G., 'Identification of Nuclear Particles', *Annual Review of Nuclear Science*, Vol 25, 1975.

[61] GURD J.R., KIRKHAM C.C. and WATSON I., 'The Manchester Prototype Dataflow Computer', *Communications of the ACM*, Vol 28, No 1, January 1985.

[62] HANDLER W., MAEHLE E. and WIRL K., 'DIRMU Multiprocessor Configurations', Proceedings of the 1984 International Conference on Parallel Processing, IEEE Comput Soc Press, Washington, DC, USA, 1985.

[63] HAWKES L.W., 'A Regular Fault-Tolerant Architecture for Interconnection Networks', *IEEE Transactions on Computers*, Vol C-34, No 7, July 1985.

[64] HAYES J.P., *Computer Architecture and Organization*, McGraw-Hill, 1978, ISBN 0-07-027363-4.

[65] HERTZBERGER L.O., 'New Architectures', *Computing in High Energy Physics*, Proceedings of the Conference on Computing in High Energy Physics, North-Holland, Amsterdam, Netherlands, 1986, ISBN 0-444-87973-0.

[66] HERTZBERGER L.O., 'Trends in Architectures', *Proceedings of the 1986 CERN School of Computing*, CERN, Geneva, April 1987.

[67] HOARE C.A.R., 'Communicating Sequential Processes', Communications of the ACM, Vol 21, No 8, August 1978.

[68] HOCKNEY R.W. and JESSHOPE C.R., *Parallel Computers, Architecture, Programming and Algorithms*, Adam Hilger Ltd, Bristol, 1981, ISBN 0-85274-422-6.

[69] HOROWITZ E. and ZORAT A., 'The Binary Tree as an Interconnection Network: Applications to Multiprocessor systems and VLSI', *IEEE Transactions on Computers*, Vol C-30, No 4, April 1981.

[70] HOUSHENG Z., 'Buffer_Insertion Ring: A Performance Study', *Microcomputers, Usage and Design*, Eleventh Euromicro Symposium on Microprocessing and Microprogramming, North-Hollland, Amsterdam, Netherlands, 1985.

[71] HUANG A., 'Architectural Considerations involved in the Design of an Optical Digital Computer', *Proceedings of the IEEE*, Vol 72, No 7, July 1984.

[72] HUFF B.R., 'The Cyberplus Parallel Processing System - A Supercomputer Alternative', *Computing in High Energy Physics*, Proceedings of the Conference on Computing in High Energy Physics, North-Holland, Amsterdam, Netherlands, 1986, ISBN 0-444-87973-0.

[73] HWANG K. and ZHIWEI X., 'Remps: A Reconfigurable Multiprocessor for Scientific Supercomputing', Proceedings of the 1984 International Conference on Parallel Processing, IEEE Comput Soc Press, Washington, DC, USA, 1985.

[74] HYVÄRINEN O., 'A High Performance Network Architecture for Digital Signal Processing Oriented VLSI Based Multiprocessor System', *VLSI in Computers and Communications*, 2nd Nordic Symposium on VLSI in Computers and Communications, 1986.

[75] IBBETT R.N., CAPON P.C. and TOPHAM N.P., 'MU6V: A Parallel Vector Processing System', Twelth Annual in Symposium on Computer Architecture, IEEE Comput Soc Press, 1985, ISBN 0-8186-0634-7.

[76] ICHIOKA Y. and TANIDA J., 'Optical Parallel Logic Gates Using a Shadow-Casting System for Optical digital Computing', *Proceedings of the IEEE*, Vol 72, No 7, July 1984.

[77] IIZUKA. A. *et al.*, 'Evolution of DRAM in Silicon MOS Technology', *VLSI and COMPUTERS*, Proceedings of First International Conference on Computer Technology, Systems and Applications, IEEE Comput Soc Press, 1987, ISBN 0-8186-0773-4.

[78] INMOS LIMITED, *The Transputer Family*, Product Information, INMOS Limited, Bristol, U.K., March 1986.

[79] INMOS LIMITED., *Occam 2*, Product Definition, INMOS Limited, Bristol, UK, June 1986.

[80] KANT K. and SILBERSCHATZ A., 'Error Propagation and Recovery in Concurrent Environments', *The Computer Journal*, Vol 28, No 5, 1985.

[81] KAPLAN I., 'The LDF 100: A Large Grain Dataflow Parallel Processor', *Computer Architecture News*, Vol 15, No 3, June 1987.

[82] KARPLUS W.J., 'Parallelism and Pipelining: The Road to more Cost-Effective Scientific Computing', *Multiprocessors and Array Processors*, Proceedings of the third Conference on Multiprocessors and Array Processors, Society for Computer Simulation, 1987.

[83] KERNIGHAN B.W. and RITCHIE D.M., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978, ISBN 0-13-110163-3.

[84] KIMURA T., KURIHARA K. and MATSUKAWA M., 'Application of the Fast Array Processor for JT-60 Plasma Control', *Fusion Technology 1986*, Volume 2, Proceedings of the fourteenth Symposium, Pergamon Press, 1986.

[85] KOREN I. and PELED I., 'The Concept and Implementation of Data-Driven Processor Arrays', *IEEE Computer*, July 1987.

[86] KOWARIK O., KRAUS R. and HOFFMAN K., 'Self Repairing Semiconductor Memories', *VLSI and COMPUTERS*, Proceedings of First International Conference on Computer Technology, Systems and Applications, IEEE Comput Soc Press, 1987, ISBN 0-8186-0773-4.

[87] KRUSKAL C.P., RUDOLPH L. and CYTRON R., 'The Architecture of Parallel Computers', *Control Flow and Data Flow: Concepts of Distributed Programming*, Proceedings of NATO Advanced Institute International Summer School, Springer-Verlag, Berlin, Germany, 1985, ISBN 0-38-713919-2.

[88] KUEHN J.T. and SIEGEL H.J., 'Extensions to the C Programming Language for SIMD/MIMD parallelism', Proceedings of the 1984 International Conference on Parallel Processing, IEEE Comput Soc Press, Washington, DC, USA, 1985.

[89] KUNG S.Y. *et al.*, 'Wavefront Array Processors - Concept to Implementation', *IEEE Computer*, July 1987.

[90] KUNZ P.F., 'Resume on Vector and Parallel Processing in HEP', *Computing in High Energy Physics*, Proceedings of the Conference on Computing in High Energy Physics, North-Holland, Amsterdam, Netherlands, 1986, ISBN 0-444-87973-0.

[91] LEBEE P., GUILLEMONT M. and FONTENIER G., 'A Heterogeneous Parallel Distributed Machine Prototype: the HPDM', Sixth Annual Conference on Computers and Communications, IEEE Comput Soc Press, 1987.

[92] LESTER B.P., 'Analysis of Firing Rates in Petri Nets using Linear Algebra', Proceedings of the 1984 International Conference on Parallel Processing, IEEE Comput Soc Press, 1985.

[93] LEVITAN S.P., 'Evaluation Criteria for Communication Structures in Parallel Architectures', Proceedings of the 1984 International Conference on Parallel Processing, IEEE Comput Soc Press, Washington, DC, USA, 1985.

[94] LINDNER R., 'Introduction to a Simple but unconventional multiprocessor and an outline of an application', *Computer Architectures for Spatially Distributed Data*, Proceedings of NATO Advanced Study Institute, Springer-Verlag, Berlin, Germany, 1985.

[95] LOHMANN A.W., 'Optical Computers', *VLSI and Computers*, Proceedings of First International Conference on Computer Technology, Systems and Applications, IEEE Comput Soc Press, 1987, ISBN 0-8186-0773-4.

[96] MAENG S.R. and CHO J.W., 'A Control and Data Flow Multiprocessor', *The Australian Computer Journal*, Vol 18, No 1, February 1986.

[97] MANUEL T. and BARNEY C., 'The Big Drag on Computer Throughput', *Electronics*, November 1986.

[98] MARK P.B, 'The Sequoia Computer: A Fault-Tolerant Tightly-Coupled Multiprocessor Architecture', Twelth Annual International Symposium on Computer Architecture, IEEE Comput Soc Press, 1985.

[99] MARSAN M.A., CONTE G. and BALBO G., 'Performance Analysis of Multiprocessor Systems', *Multi-Microprocessor Systems for Real-Time Applications*, ed Conte. G and Del Corso. D., D. Reidel Publishing Company, 1985, ISBN 90-277-2954-1.

[100] MATELAN N., 'The Flex/32 Multicomputer', Twelth Annual International Symposium on Computer Architecture, IEEE Comput Soc Press, 1985, ISBN 0-8186-0634-7.

[101] MAY D. and SHEPHERD R., 'The INMOS Transputer', *Parallel Processing, State of the Art Report*, Pergamon Infotech Limited, Maidenhead, UK, 1987, ISBN 0-08-034113-6.

[102] MAY D., KINGSMITH T. and PEARSON I., 'The T414 Transputer - the end of the beginning', *Electronic Engineering*, November 1985.

[103] MENEZES B.L. and JENEVEIN R.M., 'KYKLOS: A Linear Growth Fault-Tolerant Interconnection Network', Proceedings of the 1984 International Conference on Parallel Processing, IEEE Comput Soc Press, Washington, DC, USA, 1985.

[104] MEYER E.L., 'Survey of Multiprocessors', *VLSI Systems Design*, November 1985.

[105] MEYER F.J. and PRADHAN D.K., 'Dynamic Testing Strategy for Distributed Systems', *FTCS 15*, Proceedings of Fifteenth Annual International Symposium on Fault-Tolerant Computing, IEEE Comput Soc Press, 1985, ISBN 0-8186-0618-5.

[106] MISHIN A.I. and SEDUKHIN S.G., 'Cellular Computing Systems and Parallel Computation', *Automation, Control and Computer Science*, Vol 15, No 1, 1981.

[107] MORTON S.G., ABREU E. and TSE F., 'ITT CAP-Toward a Personal Supercomputer', *IEEE Micro*, December 1985.

[108] MOUNT R., 'Alternatives in High Volume HEP Computing', *Computing in High Energy Physics*, Proceedings of the Conference on Computing in High Energy Physics, North-Holland, Amsterdam, Netherlands, 1986, ISBN 0-444-87973-0.

[109] MOUNT R.P., 'Computer Architectures for High Energy Physics', Proceedings of the 1986 CERN School of Computing, CERN, Geneva, April 1987.

[110] NAEINI R., 'A few Statement types adapt C language to parallel processing', *Electronics*, June, 1984.

[111] NEAL L.R., 'Novel Computer Architectures - Part 2: Data Flow Computation', *Computer Education*, June 1987.

[112] NESTLE E. and INSELBERG A., 'The Synapse N+1 System: Architectural Characteristics and Performance Data of a Tightly-Coupled Multiprocessor System', Twelth Annual International Symposium on Computer Architecture, IEEE Comput Soc Press, 1985.

[113] PALMER J.F., 'The NCUBE family of Parallel Supercomputers', Fifth Annual International Phoenix Conference on Computers and Communications, IEEE Comput Soc Press, 1986, ISBN 0-8186-0691-6.

[114] PAPAZOGLOU M., 'A Proposal for Two Multimicroprocessor Architectures for Execution of Procedural Languages', *Microcomputers, Usage and Design*, Eleventh Euromicro Symposium on Microprocessing and Microprogramming, North-Holland, Amsterdam, Netherlands, 1985.

[115] PETERSON J.C. *et al.*, 'The Mark III Hypercube-Ensemble Concurrent Computer', Proceedings of the 1984 International Conference on Parallel Processing, IEEE Comput Soc Press, Washington, DC, USA, 1985.

[116] PFISTER G.F. *et al.*, 'The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture', Proceedings of the 1984 International Conference on Parallel Processing, IEEE Comput Soc Press, Washington, DC, USA, 1985.

[117] PLUM T., *Learning to Program in C*, Prentice-Hall, Englewood Cliffs, NJ, 1983, ISBN 0-13-527854-6.

[118] PRADHAN D.K. and REDDY S.M., 'A Fault-Tolerant Communication Architecture for Distributed Systems', *IEEE Transactions on Computers*, Vol C-31, No 9, September 1982.

[119] PROTOPAPAS D.A and DENENBERG J.N., 'A New Model for Performance Analysis of Large Scale Multimicrocomputer Networks', Sixth Annual International Phoenix Conference on Computers and Communications, IEEE Comput Soc Press, Washington, DC, USA, 1987.

[120] QUINTON P. and VERJUS J.P., 'Distributed Synchronisation of Parallel Programs: Why and How?', Parallel Algorithms & Architectures, Proceedings of the International Workshop on Parallel Algorithms & Architectures, North-Holland, Amsterdam, 1986.

[121] RAMMIG F.J., 'Multilevel Simulation Techniques', *VLSI and Computers*, First International Conference on Computer Technology, Systems and Applications, IEEE Comput Soc Press, 1987, ISBN 0-8186-0773-4.

[122] REED D.A. and SCHWETMAN H.D., *'Cost-Performance Bounds* for Multimicrocomputer Networks', *IEEE Transactions on Computers*, Vol C-32, No 1, January 1983.

[123] ROBERTS D.H., 'Electrons, Phonons and Photons - The Physical Particles of Electronics', Colloquium, Dept of Pure and Applied Physics, University of Salford, Salford, U.K., 25 May 1988.

[124] RODDA L., SAVIONI R. and SECHI G.R., 'A Hierarchical Architecture with Independant Processors for Real-Time Systems', *Microprocessing and Microprogramming*, Vol 15, 1985.

[125] ROSE J., LOUCKS W. and VRANESIC Z., 'FERMTOR: A Tunable Multiprocessor Architecture', *IEEE Micro*, August 1985.

[126] SANGUINETTI J. and KUMAR B., 'Performance of a Message-Based Multiprocessor', Twelth Annual International Symposium on Computer Architecture, IEEE Comput Soc Press, 1985, ISBN 0-8186-0634-7.

[127] SATYANARAYANAN M., 'Multiprocessing: An Annotated Bibliography', *IEEE Computer*, May 1980.

[128] SAWCHUK A.A. and STRAND T.C., 'Digital Optical Computing', *Proceedings of the IEEE*, Vol 72, No 7, July 1984.

[129] SCHOLTEN J., HOFSTEDE J. and SMIT G.J.M., 'Proposal for an architecture for TUMULT based on a serial link', *Microprocessing and Microprogramming*, Vol 21, Part 1-5, August 1987.

[130] SCHOLTEN J., SMIT G.J.M. and VAN'T HOFF E.L., 'Realisation of an Interconnection Network for TUMULT', *Microcomputers: Usage and Design*, Eleventh Euromicro Symposium on Microprocessing and Microprogramming, North-Holland, Amsterdam, Netherlands, 1985.

[131] SCOLLO G. *et al.*, 'Specification and Implementation of the MONDAN System', *Protocol Specification, Testing and Verification, IV*, Elsevier Science Publishers B.V., 1985.

[132] SEDGWICK R., *Algorithms*, Addison-Wesley, 1984, ISBN 0-201-06672-6.

[133] SEITZ C.L., 'Concurrent VLSI Architectures', *IEEE Transactions on Computers*, C-33, No 12, December 1984.

[134] SEITZ C.L., 'The Cosmic Cube', *Communications of the ACM*, Vol 28, No 1, January 1985.

[135] SERLIN O., 'Parallel Processing: Fact or Fancy?', *Datamation*, December 1985.

[136] SHARP J.A., *An Introduction to Distributed and Parallel Processing*, Blackwell Scientific Publications, London, 1987, ISBN 0-632-01745-7.

[137] SHIMODA T. and ISHIHARA M., 'Simple ΔE-E Particle Identification with a with a wide dynamic range', *Nuclear Instruments and Methods*, Vol 165, 1979.

[138] SHORE J.E., 'Second Thoughts on Parallel Processing', *Computer Elec Eng*, Vol 1, 1973.

[139] SMITH J.R., 'Parallel Algorithms for Depth First Searches in Planar Graphs', *SIAM Journal of Computing*, Vol 15, No 3, August 1986.

[140] SNYDER L., 'Introduction to the Configurable Highly Parallel Computer', *IEEE Computer*, January 1982.

[141] SOWA M., 'A Method for Speeding up Serial Processing in Dataflow Computers by Means of a Program Counter', *The Computer Journal*, Vol 30, No 4, 1987.

[142] STONE H.S., *Introduction to Computer Architecture*, Second edition, Science Research Associates Inc, 1980, ISBN 0-574-21225-6.

[143] TAYLOR S., AV-RON E. and SHAPIRO E., 'A Layered Method for Process and Code Mapping', *New Generation Computing*, Vol 5, No 2, 1987.

[144] THEAKER C.J. and BROOKES R.B., *A Practical Course on Operating Systems*, Macmillan, 1984, ISBN 0-333-34678-5.

[145] TRELEAVEN P.C., BROWNBRIDGE D.R. and HOPKINS R.P., 'Data-Driven and Demand-Driven Computer Architecture', *Computing Surveys*, Vol 14, No 1, March 1982.

[146] TSENG P.S., HWANG K. and PRASANNA KUMAR V.K., 'A VLSI-Based Multiprocessor Architecture for Implementing Parallel Algorithms', Proceedings of the 1984 International Conference on Parallel Processing, IEEE Comput Soc Press, Washington, DC, USA, 1985.

[147] TUAZON J. *et al.*, 'Caltech/JPL Mark II Hypercube Concurrent Processor', Proceedings of the 1984 International Conference on Parallel Processing, IEEE Comput Soc Press, Washington, DC, USA, 1985.

[148] VAJDA F., 'Critical Issues of the Application of a Transputer in a Concurrent System', *Microcomputers: Usage and Design*, Eleventh Euromicro Symposium on Microprocessing and Microprogramming, North-Holland, Amsterdam, Netherlands, 1985.

[149] VAN ZANDT J., 'C^3I Beyond the Von Neumann Bottleneck', *Defense Electronics*, January 1986.

[150] VEDDER R. and FINN D., 'The Hughes Data Flow Multiprocessor: Architecture for Efficient Signal and Data Processing', Twelth Annual International Symposium on Computer Architecture, IEEE Comput Soc Press, 1985, ISBN 0-8186-0634-7.

[151] VIITANEN J. and VÄNNI P., 'The TAMIPS Multiprocessor', Proceedings of the 1984 International Conference on Parallel Processing, IEEE Comput Soc Press, Washington, DC, USA, 1985.

[152] VON CONTA, C., 'Torus and Other Networks as Communication Networks With Up To Some Hundred Points', *IEEE Transactions on Computers*, Vol C-32, No 7, July 1983.

[153] WATSON I. and GURD J., 'A Practical Data Flow Computer', *IEEE Computer*, February 1982.

[154] WHERRETT P., 'Potential of Optical Computing', Colloqium, Dept of Physics, University of Salford, Salford, UK, 17 February 1988.

[155] WHITBY-STREVENS C., 'The Transputer', Twelth Annual International Symposium on Computer Architecture, IEEE Comput Soc Press, 1985, ISBN 0-8186-0634-7.

[156] WILSON P., 'Highly Concurrent Systems Using the Transputer', *NorthCon 84*, Mini-Micro NorthWest 1984, Conference Record, Seattle, USA, 1984.

[157] WILSON P., 'OCCAM - A Programming Language for Concurrent Systems', *Multiprocessors and Array Processors*, Proceedings of Third Conference on Multiprocessors and Array Processors, Society for Computer Simulation, 1987.

[158] WITTIE L.D., 'Communication Structures for Large Networks of Microcomputers', *IEEE Transactions on Computers*, Vol C-30, No 4, April 1981.

[159] WU S.B. and LIU M.T., 'A Cluster Structure as an Interconnection Network for Large Multimicrocomputer Systems', *IEEE Transactions on Computers*, Vol C-30, No 4, April 1981.

[160] ZABLOTSKII V.N. *et al.*, 'Approximate Analytic Model of Multiprocessor Computer System With a Ring Structure', *Automation, Control and Computer Science*, Vol 18, No 3, 1984.

[161] ZAKHAROV V., 'Parallelism and Array Processing', *IEEE Transactions on Computers*, C-33, No 1, January 1984.

[162] ZANELLA P., 'Trends in Computing for HEP', *Computing in High Energy Physics*, Proceedings of the Conference on Computing in High Energy Physics, North-Holland, Amsterdam, Netherlands, 1986, ISBN 0-444-87973-0.

[163] ZILOG CORPORATION, *Z-80 Product description*, Zilog Corporation, 1977.

A COMPUTING STRUCTURE FOR DATA ACQUISITION IN HIGH ENERGY PHYSICS

by

GARRY ALEXANDER LESTER

Submitted for the Degree of

Doctor of Philosophy

at the

University of Salford

in the

Department of Electronic and Electrical Engineering

1988

MCMLXXXVIII

Volume II

APPENDICES

Contents

APPENDIX 1

1 Simulation Programs of Distinct Node Rings

```
program distring1(input,output,distdata,distrecord);
const
(* CONSTANTS FOR RANDOM NUMBER GENERATION *)
  m=100000000;m1=10000;b=31415821;
type
(* PROCESSING NODE TYPE *)
  proc_ptr=^proc_type;
  proc_type=record
              proc_id:integer;
              ring_input:integer;
              ring_output:integer;
              input_full:boolean;
              output_full:boolean;
              data_input:integer;
              data_full:boolean;
              p_type:integer;
              p_state:integer;
              next:proc_ptr;
              end;

var
  a:integer;
  pconst:integer;
  distdata,distrecord:text;
  ring:proc_ptr;
  processing:array[1..100] of integer;
  processed_data:array[1..100] of integer;
  consumed_data:array[1..100] of integer;
  largen,iterations,procs:integer;

  function mult(p,q:integer):integer;
  (* EXTENDED MULTIPLICATION *)
  var p1,p0,q1,q0:integer;
  begin
  p1:=p div m1;  p0:=p mod m1;
  q1:=q div m1;  q0:=q mod m1;
  mult:=(((p0*q1+p1*q0) mod m1)*m1+p0*q0) mod m;
  end;

  function random:real;
  (* RANDOM NUMBER GENERATOR *)
  (* PG37 - ALGORITHMS - SEDGEWICK *)
  (* ADDISON - WESLEY 1984 *)
  begin
  a:=(mult(a,b)+1)mod m;
  random:=a/m;
  end;

  procedure input_parameters;
  (* INITIALISE INPUT AND OUTPUT FILES *)
  (* READ IN THE PARAMETERS OF THE RING TO MODEL *)
  var
```

```
  n:integer;
begin (* INPUT_PARAMETERS *)
reset(distdata);rewrite(distrecord);
writeln(distrecord,'distring1');
writeln(distrecord,'Ring State record for simulation using :-');
(* READ IN THE NUMBER OF PROCESSORS *)
readln(distdata,procs);
writeln(distrecord,'Number of processors = ',procs:3);
(* READ IN THE NUMBER OF ITERATIONS TO PERFORM *)
readln(distdata,largen);
writeln(distrecord,'Number of iterations performed = ',largen:3);
(* READ IN THE AMOUNT OF PROCESSING REQUIRED PER DATA ITEM *)
readln(distdata,pconst);
writeln(distrecord,'Processing required per data item = ',pconst:3,' units')
writeln(distrecord);
end;  (* INPUT_PARAMETERS *)

procedure initial_states;
(* SETS UP THE INITIAL STATE OF THE RING DATA STRUCTURE *)
(* AND PRINTS A HEADING TO FILE RINGRECORD *)
var
  p:proc_ptr;
  n:integer;

  procedure set_vars;
  begin (* SET_VARS *)
  p^.input_full:=false;
  p^.output_full:=false;
  p^.data_full:=false;
  p^.p_type:=0;
  p^.p_state:=0;
  end;  (* SET_VARS *)

begin (* INITIAL_STATES *)
write(distrecord,'     ');
new(ring);
p:=ring;
p^.proc_id:=1;
write(distrecord,' ',p^.proc_id:3);
set_vars;
for n:=1 to (procs-1) do
  begin
  new(p^.next);
  p:=p^.next;
  p^.proc_id:=(n+1);
  write(distrecord,' ',p^.proc_id:3);
  set_vars;
  end;
p^.next:=ring;
writeln(distrecord);
for n:=1 to 100 do
  begin
  processing[n]:=0;
  processed_data[n]:=0;
  consumed_data[n]:=0;
  end;
end;  (* INITIAL_STATES *)
```

```
procedure communicate;
(* PERFORMS THE FUNCTION OF THE COMMUNICATION HARDWARE *)
var
  p:proc_ptr;
begin (* COMMUNICATE *)
p:=ring;
repeat
  begin
  if (p^.output_full and not(p^.next^.input_full)) then
    begin
    p^.next^.ring_input:=p^.ring_output;
    p^.next^.input_full:=true;
    p^.output_full:=false;
    end;
  p:=p^.next;
  end;
until (p=ring);
end;  (* COMMUNICATE *)

procedure create_new_data;
(* PERFOMS THE FUNCTION OF THE DATA INPUT MECHANISM *)
var
  p:proc_ptr;

begin (* CREATE_NEW_DATA *)
p:=ring;
repeat
  begin
  if (not(p^.data_full)) then
    begin
    p^.data_input:=1+trunc(random*(procs-0.0001));
    consumed_data[p^.data_input]:=consumed_data[p^.data_input]+1;
    p^.data_full:=true;
    end;
  p:=p^.next;
  end;
until (p=ring);
end;  (* CREATE_NEW_DATA *)

procedure compute;
(* PERFORMS THE FUNCTION OF THE PROCESSING ELEMENT *)
var
  p:proc_ptr;

  procedure computing_algorithm;
  (* PERFORMS THE FUNCTION OF THE COMMUNICATING PROCESSES WITHIN *)
  (* THE PROCESSOR *)
  var
    time_unit:integer;

    procedure comms;
    (* PERFORMS THE ACTIVITIES OF THE COMMUNICATION PROCESS *)

      function processor_idle:boolean;
      begin
      processor_idle:=(p^.p_state<=0);
      end;
```

```
function ring_data:boolean;
begin
ring_data:=p^.input_full;
end;

function new_data:boolean;
begin
new_data:=p^.data_full;
end;

function new_data_right:boolean;
begin
new_data_right:=(p^.data_input=p^.proc_id);
end;

function ring_data_right:boolean;
begin
ring_data_right:=(p^.ring_input=p^.proc_id);
end;

function ring_ready:boolean;
begin
ring_ready:=not(p^.output_full);
end;

procedure take_ring_data;
begin
if (pconst<0) then
  p^.p_state:=trunc(random*(-pconst))
else
  p^.p_state:=pconst;
p^.p_type:=p^.p_state;
p^.input_full:=false;
time_unit:=time_unit-1;
end;

procedure take_new_data;
begin
if (pconst<0) then
  p^.p_state:=trunc(random*(-pconst))
else
  p^.p_state:=pconst;
p^.p_type:=p^.p_state;
p^.data_full:=false;
time_unit:=time_unit-1;
end;

procedure ring_data_on;
begin
p^.ring_output:=p^.ring_input;
p^.input_full:=false;
p^.output_full:=true;
time_unit:=time_unit-1;
end;

procedure new_data_on;
begin
p^.ring_output:=p^.data_input;
```

```
  p^.data_full:=false;
  p^.output_full:=true;
  time_unit:=time_unit-1;
  end;

begin (* COMMS1 *)
if processor_idle then
  if new_data then
    if new_data_right then
      take_new_data
    else
      if ring_data then
        if ring_data_right then
          take_ring_data
        else
          (* NULL *)
      else
        (* NULL *)
  else
    if ring_data then
      if ring_data_right then
        take_ring_data
      else
        (* NULL *)
    else
      (* NULL *)
else
  (* NULL *);
if ring_ready then
if new_data then
  if not new_data_right then
    new_data_on
  else
    if ring_data then
      if not ring_data_right then
        ring_data_on
      else
        (* NULL *)
    else
     (* NULL *)
else
  if ring_data then
    if not ring_data_right then
      ring_data_on
    else
      (* NULL *)
  else
    (* NULL *)
else
  (* NULL *);
end;  (* COMMS1 *)

procedure process(units:integer);
(* PERFORMS THE ACTIVITIES OF THE PROCESSING PROCESS *)
begin (* PROCESS *)
p^.p_state:=p^.p_state-units;
if (p^.p_state<=0) then
  begin
```

```
    if (p^.p_type>0) then
      begin
      processed_data[p^.p_type]:=processed_data[p^.p_type]+1;
      processing[p^.proc_id]:=processing[p^.proc_id]+1;
      end;
    p^.p_type:=0;
    p^.p_state:=0;
    end;
  end;  (* PROCESS *)

begin (* COMPUTING_ALGORITHM *)
time_unit:=4;
(* TOTAL PROCESSING EFFORT AVAILABLE *)
  (* PER ITERATION *)
  comms;
  (* COMMS USES UP SOME PROCESSING EFFORT *)
  process(time_unit);
  end;  (* COMPUTING_ALGORITHM *)

begin (* COMPUTE *)
p:=ring;
repeat
  begin
  computing_algorithm;
  p:=p^.next;
  end;
until (p=ring);
end;  (* COMPUTE *)

procedure print_state;
(* PRINTS THE STATE OF THE RING AFTER THE LAST ITERATION *)
var p:proc_ptr;
begin (* PRINT_STATE *)
write(distrecord,'I=',iterations:3);
p:=ring;
repeat
  begin
  write(distrecord,' ',p^.p_state:3);
  p:=p^.next;
  end;
until (p=ring);
writeln(distrecord);
end;  (* PRINT_STATE *)

procedure print_results;
var
  n,totalproc,totalcons,wtotproc:integer;
begin (* PRINT_RESULTS *)
totalproc:=0;totalcons:=0;wtotproc:=0;
writeln(distrecord);writeln(distrecord);
writeln(distrecord,'Data Consumed and Processed');
writeln(distrecord,'---- -------- --- ---------');writeln(distrecord);
for n:=1 to procs do
  begin
  totalcons:=totalcons+consumed_data[n];
  totalproc:=totalproc+processing[n];
  write(distrecord,'P.E.:',n:3,' number consumed:',consumed_data[n]:3);
  writeln(distrecord,' number processed:',processing[n]:3);
```

```
  end;
writeln(distrecord);writeln(distrecord);
writeln(distrecord,'Types Processed');
writeln(distrecord,'----- ----------');writeln(distrecord);
for n:=1 to 50 do
  begin
  wtotproc:=wtotproc+processed_data[n]*n;
  writeln(distrecord,'Type:',n:3,' number processed:',processed_data[n]:3);
  end;
write(distrecord,'Total:  ',' number consumed:',totalcons:3);
writeln(distrecord,' number processed:',totalproc:3);
writeln(distrecord,'Total:  ',' Weighted Total Processed:',wtotproc:5);
  end;  (* PRINT_RESULTS *)

begin (* DISTRING1 *)
a:=1234567;
(* SETS THE SEED FOR THE RANDOM NUMBER GENERATOR *)
input_parameters;
writeln('Parameters input');
initial_states;
writeln('Initial state set up');
writeln('Starting Computation');
for iterations:=1 to largen do
  begin
  communicate;
  create_new_data;
  compute;
  print_state;
  end;
writeln('Finished Computation');
writeln('Printing Results');
print_results;
end.  (* DISTRING1 *)
```

```
program distring2(input,output,distdata,distrecord);
const
(* CONSTANTS FOR RANDOM NUMBER GENERATION *)
  m=100000000;m1=10000;b=31415821;
type
(* PROCESSING NODE TYPE *)
  proc_ptr=^proc_type;
  proc_type=record
              proc_id:integer;
              ring_input:integer;
              ring_output:integer;
              input_full:boolean;
              output_full:boolean;
              data_input:integer;
              data_full:boolean;
              p_type:integer;
              p_state:integer;
              next:proc_ptr;
              end;

var
  a:integer;
  pconst:integer;
  distdata,distrecord:text;
  ring:proc_ptr;
  processing:array[1..100] of integer;
  processed_data:array[1..100] of integer;
  consumed_data:array[1..100] of integer;
  largen,iterations,procs:integer;

function mult(p,q:integer):integer;
(* EXTENDED MULTIPLICATION *)
var p1,p0,q1,q0:integer;
begin
p1:=p div m1;  p0:=p mod m1;
q1:=q div m1;  q0:=q mod m1;
mult:=(((p0*q1+p1*q0) mod m1)*m1+p0*q0) mod m;
end;

function random:real;
(* RANDOM NUMBER GENERATOR *)
(* PG37 - ALGORITHMS - SEDGEWICK *)
(* ADDISON - WESLEY 1984 *)
begin
a:=(mult(a,b)+1)mod m;
random:=a/m;
end;

procedure input_parameters;
(* INITIALISE INPUT AND OUTPUT FILES *)
(* READ IN THE PARAMETERS OF THE RING TO MODEL *)
var
  n:integer;
begin (* INPUT_PARAMETERS *)
reset(distdata);rewrite(distrecord);
writeln(distrecord,'distring2');
writeln(distrecord,'Ring State record for simulation using :-');
(* READ IN THE NUMBER OF PROCESSORS *)
```

```
readln(distdata,procs);
writeln(distrecord,'Number of processors = ',procs:3);
(* READ IN THE DATA_TYPE (PROCESSING TIME) FOR EACH PROCESSOR *)
readln(distdata,largen);
writeln(distrecord,'Number of iterations performed = ',largen:3);
(* READ IN THE AMOUNT OF PROCESSING REQUIRED PER DATA ITEM *)
readln(distdata,pconst);
writeln(distrecord,'Processing required per data item = ',pconst:3,' units');
writeln(distrecord);
end;  (* INPUT_PARAMETERS *)

procedure initial_states;
(* SETS UP THE INITIAL STATE OF THE RING DATA STRUCTURE *)
(* AND PRINTS A HEADING TO FILE RINGRECORD *)
var
  p:proc_ptr;
  n:integer;

  procedure set_vars;
  begin (* SET_VARS *)
  p^.input_full:=false;
  p^.output_full:=false;
  p^.data_full:=false;
  p^.p_type:=0;
  p^.p_state:=0;
  end;  (* SET_VARS *)

begin (* INITIAL_STATES *)
write(distrecord,'     ');
new(ring);
p:=ring;
p^.proc_id:=1;
write(distrecord,' ',p^.proc_id:3);
set_vars;
for n:=1 to (procs-1) do
  begin
  new(p^.next);
  p:=p^.next;
  p^.proc_id:=(n+1);
  write(distrecord,' ',p^.proc_id:3);
  set_vars;
  end;
p^.next:=ring;
writeln(distrecord);
for n:=1 to 100 do
  begin
  processing[n]:=0;
  processed_data[n]:=0;
  consumed_data[n]:=0;
  end;
end;  (* INITIAL_STATES *)

procedure communicate;
(* PERFORMS THE FUNCTION OF THE COMMUNICATION HARDWARE *)
var
  p:proc_ptr;
begin (* COMMUNICATE *)
p:=ring;
```

```
repeat
  begin
  if (p^.output_full and not(p^.next^.input_full)) then
    begin
    p^.next^.ring_input:=p^.ring_output;
    p^.next^.input_full:=true;
    p^.output_full:=false;
    end;
  p:=p^.next;
  end;
until (p=ring);
end;  (* COMMUNICATE *)

procedure create_new_data;
(* PERFOMS THE FUNCTION OF THE DATA INPUT MECHANISM *)
var
  p:proc_ptr;

begin (* CREATE_NEW_DATA *)
p:=ring;
repeat
  begin
  if (not(p^.data_full)) then
    begin
    p^.data_input:=1+trunc(random*(procs-0.0001));
    consumed_data[p^.data_input]:=consumed_data[p^.data_input]+1;
    p^.data_full:=true;
    end;
  p:=p^.next;
  end;
until (p=ring);
end;  (* CREATE_NEW_DATA *)

procedure compute;
(* PERFORMS THE FUNCTION OF THE PROCESSING ELEMENT *)
var
  p:proc_ptr;

  procedure computing_algorithm;
  (* PERFORMS THE FUNCTION OF THE COMMUNICATING PROCESSES WITHIN *)
  (* THE PROCESSOR *)
  var
    time_unit:integer;

    procedure comms;
    (* PERFORMS THE ACTIVITIES OF THE COMMUNICATION PROCESS *)

    function processor_idle:boolean;
    begin
    processor_idle:=(p^.p_state<=0);
    end;

    function ring_data:boolean;
    begin
    ring_data:=p^.input_full;
    end;

    function new_data:boolean;
```

```
begin
new_data:=p^.data_full;
end;

function new_data_right:boolean;
begin
new_data_right:=(p^.data_input=p^.proc_id);
end;

function ring_data_right:boolean;
begin
ring_data_right:=(p^.ring_input=p^.proc_id);
end;

function ring_ready:boolean;
begin
ring_ready:=not(p^.output_full);
end;

procedure take_ring_data;
begin
if (pconst<0) then
  p^.p_state:=trunc(random*(-pconst))
else
  p^.p_state:=pconst;
p^.p_type:=p^.p_state;
p^.input_full:=false;
time_unit:=time_unit-1;
end;

procedure take_new_data;
begin
p^.p_type:=p^.data_input;
if (pconst<0) then
  p^.p_state:=trunc(random*(-pconst))
else
  p^.p_state:=pconst;
p^.p_type:=p^.p_state;
p^.data_full:=false;
time_unit:=time_unit-1;
end;

procedure ring_data_on;
begin
p^.ring_output:=p^.ring_input;
p^.input_full:=false;
p^.output_full:=true;
time_unit:=time_unit-1;
end;

procedure new_data_on;
begin
p^.ring_output:=p^.data_input;
p^.data_full:=false;
p^.output_full:=true;
time_unit:=time_unit-1;
end;
```

```
begin (* COMMS2 *)
if processor_idle then
  if ring_data then
    if ring_data_right then
      take_ring_data
    else
      if new_data then
        if new_data_right then
          take_new_data
        else
          (* NULL *)
      else
        (* NULL *)
  else
    if new_data then
      if new_data_right then
        take_new_data
      else
        (* NULL *)
    else
      (* NULL *)
else
  (* NULL *);
if ring_ready then
  if ring_data then
    if not ring_data_right then
      ring_data_on
    else
      if new data then
        if not new_data_right then
          new_data_on
        else
          (* NULL *)
      else
       (* NULL *)
  else
    if new_data then
      if not new_data_right then
        new_data_on
      else
        (* NULL *)
    else
      (* NULL *)
else
  (* NULL *);
end;  (* COMMS2 *)
```

```
procedure process(units:integer);
(* PERFORMS THE ACTIVITIES OF THE PROCESSING PROCESS *)
begin (* PROCESS *)
p^.p_state:=p^.p_state-units;
if (p^.p_state<=0) then
  begin
  if (p^.p_type>0) then
    begin
    processed_data[p^.p_type]:=processed_data[p^.p_type]+1;
    processing[p^.proc_id]:=processing[p^.proc_id]+1;
    end;
```

```
    p^.p_type:=0;
    p^.p_state:=0;
    end;
  end;  (* PROCESS *)

begin (* COMPUTING_ALGORITHM *)
time_unit:=4;
(* TOTAL PROCESSING EFFORT AVAILABLE *)
(* PER ITERATION *)
comms;
(* COMMS USES UP SOME PROCESSING EFFORT *)
process(time_unit);
end;  (* COMPUTING_ALGORITHM *)

begin (* COMPUTE *)
p:=ring;
repeat
  begin
  computing_algorithm;
  p:=p^.next;
  end;
until (p=ring);
end;  (* COMPUTE *)

procedure print_state;
(* PRINTS THE STATE OF THE RING AFTER THE LAST ITERATION *)
var p:proc_ptr;
begin (* PRINT_STATE *)
write(distrecord,'I=',iterations:3);
p:=ring;
repeat
  begin
  write(distrecord,' ',p^.p_state:3);
  p:=p^.next;
  end;
until (p=ring);
writeln(distrecord);
end;  (* PRINT_STATE *)

procedure print_results;
var
  n,totalproc,totalcons,wtotproc:integer;
begin (* PRINT_RESULTS *)
totalproc:=0;totalcons:=0;wtotproc:=0;
writeln(distrecord);writeln(distrecord);
writeln(distrecord);writeln(distrecord,'Data Consumed and Processed');
writeln(distrecord,'---- -------- --- ----------');writeln(distrecord);
for n:=1 to procs do
  begin
  totalproc:=totalproc+processing[n];
  totalcons:=totalcons+consumed_data[n];
  write(distrecord,'P.E.:',n:3,' number consumed:',consumed_data[n]:3);
  writeln(distrecord,' number processed:',processing[n]:3);
  end;
writeln(distrecord);writeln(distrecord);
writeln(distrecord,'Types Processed');
writeln(distrecord,'----- ----------');writeln(distrecord);
for n:=1 to 50 do
```

```
    begin
    wtotproc:=wtotproc+processed_data[n]*n;
    writeln(distrecord,'Type:',n:3,' number processed:',processing[n]:3);
    end;
  write(distrecord,'Total:  ',' number consumed:',totalcons:3);
  writeln(distrecord,' number processed:',totalproc:3);
  writeln(distrecord,'Total:  ',' Weighted Total Processed:',wtotproc:5);
  end;  (* PRINT_RESULTS *)

begin (* DISTRING2 *)
a:=1234567;
(* SETS THE SEED FOR THE RANDOM NUMBER GENERATOR *)
input_parameters;
writeln('Parameters input');
initial_states;
writeln('Initial state set up');
writeln('Starting Computation');
for iterations:=1 to largen do
  begin
  communicate;
  create_new_data;
  compute;
  print_state;
  end;
writeln('Finished Computation');
writeln('Printing Results');
print_results;
end.  (* DISTRING2 *)
```

APPENDIX 2

2 Simulation Programs of Distinct Node Cylinders

```
program diststack1(input,output,distdatas,distrecords);
const
(* CONSTANTS FOR RANDOM NUMBER GENERATION *)
  m=100000000;m1=10000;b=31415821;
type
(* PROCESSING NODE TYPE *)
proc_ptr=^proc_type;
proc_type=record
            proc_r:integer;
            proc_l:integer;
            ring_input:integer;
            ring_output:integer;
            input_full:boolean;
            output_full:boolean;
            data_input:integer;
            down_output:integer;
            data_full:boolean;
            down_full:boolean;
            p_type:integer;
            p_state:integer;
            next:proc_ptr;
            down:proc_ptr;
          end;

var
  a:integer;
  pconst:integer;
  distdatas,distrecords:text;
  stack:proc_ptr;
  processed_data:array[1..100] of integer;
  processing:array[1..20,1..20] of integer;
  consumed_data:array[1..20,1..20] of integer;
  largen,iterations,ring,layers:integer;

  function mult(p,q:integer):integer;
  (* EXTENDED MULTIPLICATION *)
  var p1,p0,q1,q0:integer;
  begin
  p1:=p div m1;  p0:=p mod m1;
  q1:=q div m1;  q0:=q mod m1;
  mult:=(((p0*q1+p1*q0) mod m1)*m1+p0*q0) mod m;
  end;

  function random:real;
  (* RANDOM NUMBER GENERATOR *)
  (* PG37 - ALGORITHMS - SEDGEWICK *)
  (* ADDISON - WESLEY 1984 *)
  begin
  a:=(mult(a,b)+1)mod m;
  random:=a/m;
  end;
```

```
procedure input_parameters;
(* INITIALISE INPUT AND OUTPUT FILES *)
(* READ IN THE PARAMETERS OF THE RING TO MODEL *)
(* WRITES A HEADING TO STACKRECORD *)
var
  l,r,n:integer;
begin (* INPUT_PARAMETERS *)
reset(distdatas);rewrite(distrecords);
writeln(distrecords,'diststack1');
writeln(distrecords,'Stack State record for simulation using :-');
(* READ IN THE NUMBER OF PROCESSORS PER RING *)
readln(distdatas,ring);
writeln(distrecords,'Number of processors per layer = ',ring:3);
(* READ IN THE NUMBER OF LAYERS *)
readln(distdatas,layers);
writeln(distrecords,'Number of layers = ',layers:3);
(* READ IN THE NUMBER OF ITERATIONS TO PERFORM *)
readln(distdatas,largen);
writeln(distrecords,'Number of iterations performed = ',largen:3);
(* READ IN THE AMOUNT OF PROCESSING REQUIRED PER DATA ITEM *)
readln(distdatas,pconst);
writeln(distrecords,'processing required per data item = ',pconst:3,' units');
writeln(distrecords);
end;  (* INPUT_PARAMETERS *)

procedure initial_states;
(* SETS UP THE INITIAL STATE OF THE STACK DATA STRUCTURE *)
(* AND PRINTS A HEADING TO FILE STACKRECORD *)
var
  p,pt:proc_ptr;
  r,l,n:integer;

  procedure set_vars;
  begin (* SET_VARS *)
  p^.input_full:=false;
  p^.output_full:=false;
  p^.data_full:=false;
  p^.down_full:=false;
  p^.p_type:=0;
  p^.p_state:=0;
  end;  (* SET_VARS *)

begin (* INITIAL_STATES *)
stack:=nil;
for l:=layers downto 1 do
  begin
  new(p);
  p^.down:=stack;
  stack:=p;
  p^.proc_r:=1;
  p^.proc_l:=1;
  p^.next:=p;
  set_vars;
  end;
for r:=ring downto 2 do
  begin
  pt:=stack;
  new(p);
```

```
p^.next:=pt^.next;
pt^.next:=p;
p^.proc_r:=r;
p^.proc_l:=l;
set_vars;
for l:=2 to layers do
  begin
  pt:=pt^.down;
  new(p^.down);
  p:=p^.down;
  p^.next:=pt^.next;
  pt^.next:=p;
    p^.proc_r:=r;
    p^.proc_l:=l;
    set_vars;
    end;
  p^.down:=nil;
  end;
for l:=1 to 20 do
  for r:=1 to 20 do
    begin
    processing[l,r]:=0;
    consumed_data[l,r]:=0;
    end;
for n:=1 to 50 do
  processed_data[n]:=0;
end;  (* INITIAL_STATES *)

procedure communicate;
(* PERFORMS THE FUNCTION OF THE COMMUNICATION HARDWARE *)
var
  p,pt:proc_ptr;
begin (* COMMUNICATE *)
pt:=stack;
while (pt<>nil) do
  begin
  p:=pt;
  repeat
    begin
    if (p^.down<>nil) then
      if (p^.down_full and not(p^.down^.data_full)) then
        begin
        p^.down^.data_input:=p^.down_output;
        p^.down_full:=false;
        p^.down^.data_full:=true;
        end;
    if (p^.output_full and not(p^.next^.input_full)) then
      begin
      p^.next^.ring_input:=p^.ring_output;
      p^.next^.input_full:=true;
      p^.output_full:=false;
      end;
    p:=p^.next;
    end
  until (p=pt);
  pt:=pt^.down;
  end;
end;  (* COMMUNICATE *)
```

```
procedure create_new_data;
(* PERFOMS THE FUNCTION OF THE DATA INPUT MECHANISM *)
var
  p:proc_ptr;
  l,r:integer;

begin (* CREATE_NEW_DATA*)
p:=stack;
repeat
  begin
  if (not(p^.data_full)) then
    begin
    r:=1+trunc(random*(ring-0.0001));
    l:=1+trunc(random*(layers-0.0001));
    p^.data_input:=l*100+r;
    consumed_data[l,r]:=consumed_data[l,r]+1;
    p^.data_full:=true;
    end;
  p:=p^.next;
  end;
until (p=stack);
end; (* CREATE_NEW_DATA *)

procedure compute;
(* PERFORMS THE FUNCTION OF THE PROCESSING ELEMENT *)
var
  p,pt:proc_ptr;

procedure computing_algorithm;
(* PERFORMS THE FUNCTION OF THE COMMUNICATING PROCESSES WITHIN *)
(* THE PROCESSOR *)
var
  time_unit:integer;

  procedure comms;
  (* PERFORMS THE ACTIVITIES OF THE COMMUNICATION PROCESS *)

    function processor_idle:boolean;
    begin
    processor_idle:=(p^.p_state<=0);
    end;

    function ring_data:boolean;
    begin
    ring_data:=p^.input_full;
    end;

    function new_data:boolean;
    begin
    new_data:=p^.data_full;
    end;

    function new_data_right:boolean;
    begin
    new_data_right:=(p^.data_input=(p^.proc_r+(100*p^.proc_l)));
    end;
```

```
function ring_data_right:boolean;
begin
ring_data_right:=(p^.ring_input=(p^.proc_r+(100*p^.proc_l)));
end;

function column_right_data:boolean;
begin
column_right_data:=
  (p^.proc_r=(p^.data_input-(100*trunc(p^.data_input/100))));
end;

function column_right_ring:boolean;
begin
column_right_ring:=
  (p^.proc_r=(p^.ring_input-(100*trunc(p^.ring_input/100))));
end;

function level_right_data:boolean;
begin
level_right_data:=(p^.proc_l=(p^.data_input div 100));
end;

function level_right_ring:boolean;
begin
level_right_ring:=(p^.proc_l=(p^.ring_input div 100));
end;

function ring_ready:boolean;
begin
ring_ready:=not(p^.output_full);
end;

function down_ready:boolean;
begin
down_ready:=not(p^.down_full);
end;

procedure take_ring_data;
begin
if (pconst<0) then
  p^.p_state:=trunc(random*(-pconst))
else
  p^.p_state:=pconst;
p^.p_type:=p^.p_state;
p^.input_full:=false;
time_unit:=time_unit-1;
end;

procedure take_new_data;
begin
if (pconst<0) then
  p^.p_state:=trunc(random*(-pconst))
else
  p^.p_state:=pconst;
p^.p_type:=p^.p_state;
p^.data_full:=false;
time_unit:=time_unit-1;
end;
```

```
procedure ring_data_on;
begin
p^.ring_output:=p^.ring_input;
p^.input_full:=false;
p^.output_full:=true;
time_unit:=time_unit-1;
end;

procedure ring_data_down;
begin
p^.down_output:=p^.ring_input;
p^.input_full:=false;
p^.down_full:=true;
time_unit:=time_unit-1;
end;

procedure new_data_on;
begin
p^.ring_output:=p^.data_input;
p^.data_full:=false;
p^.output_full:=true;
time_unit:=time_unit-1;
end;

procedure new_data_down;
begin
p^.down_output:=p^.data_input;
p^.data_full:=false;
p^.down_full:=true;
time_unit:=time_unit-1;
end;

begin (* COMMS1 *)
if processor_idle then
  if new_data then
    if new_data_right then
      take_new_data
    else
      if ring_data then
        if ring_data_right then
          take_ring_data
        else
          (* NULL *)
      else
        (* NULL *)
  else
    if ring_data then
      if ring_data_right then
        take_ring_data
      else
        (* NULL *)
    else
        (* NULL *)
else
  (* NULL *);
if ring_ready then
  if new_data then
```

```
    if level_right_data and not column_right_data then
      new_data_on
    else
      if ring_data then
        if level_right_ring and not column_right_ring then
          ring_data_on
        else
          (* NULL *)
      else
        (* NULL *)
  else
    if ring_data then
      if level_right_ring and not column_right_ring then
        ring_data_on
      else
        (* NULL *)
    else
      (* NULL *)
else
  (* NULL *);
if down_ready then
  if new_data then
    if not level_right_data then
      new_data_down
    else
      if ring_data then
        if not level_right_ring then
          ring_data_down
        else
          (* NULL *)
      else
        (* NULL *)
  else
    if ring_data then
      if level_right_ring then
        ring_data_down
      else
        (* NULL *)
    else
      (* NULL *)
else
  (* NULL *)
end;  (* COMMS1 *)

procedure process(units:integer);
begin (* PROCESS *)
p^.p_state:=p^.p_state-units;
if (p^.p_state<=0) then
  begin
  if (p^.p_type>0) then
    begin
    processed_data[p^.p_type]:=processed_data[p^.p_type]+1;
    processing[p^.proc_l,p^.proc_r]:=
    processing[p^.proc_l,p^.proc_r]+1;
    end;
  p^.p_type:=0;
  p^.p_state:=0;
  end;
```

```
    end;  (* PROCESS *)

  begin (* COMPUTING_ALGORITHM *)
  time_unit:=4;
  (* TOTAL PROCESSING EFFORT AVAILABLE *)
  (* PER ITERATION *)
  comms;
  (* COMMS USES UP SOME PROCESSING EFFORT *)
  process(time_unit);
  end;  (* COMPUTING_ALGORITHM *)

begin (* COMPUTE *)
pt:=stack;
while (pt<>nil) do
  begin
  p:=pt;
  repeat
    begin
    computing_algorithm;
    p:=p^.next;
    end
  until (p=pt);
  pt:=pt^.down;
  end;
end;  (* COMPUTE *)

procedure print_state;
(* PRINTS THE STATE OF THE STACK AFTER THE LAST ITERATION *)
var p,pt:proc_ptr;
    l,r:integer;
begin (* PRINT_STATE *)
write(distrecords,'Iteration:',iterations:3);
writeln(distrecords);write(distrecords,'r=   ');
for r:=1 to ring do
  write(distrecords,' ',r:3);
writeln(distrecords);
pt:=stack;
l:=1;
while(pt<>nil) do
  begin
  p:=pt;
  write(distrecords,'l=',l:3);
  repeat
    begin
    write(distrecords,' ',p^.p_state:3);
    p:=p^.next;
    end;
  until (p=pt);
  pt:=pt^.down;
  l:=l+1;
  writeln(distrecords);
  end;
writeln(distrecords);
end;  (* PRINT_STATE *)

procedure print_results;
var
  l,r,n,totalproc,totalcons,wtotproc:integer;
```

```
begin (* PRINT_RESULTS *)
totalproc:=0;totalcons:=0;wtotproc:=0;
writeln(distrecords);writeln(distrecords,'Data Consumed');
writeln(distrecords,'---- --------');writeln(distrecords);
write(distrecords,'r=   ');
for r:= 1 to ring do
  write(distrecords,' ',r:3);
writeln(distrecords);
for l:=1 to layers do
  begin
  write(distrecords,'l=',l:3);
  for r:=1 to ring do
    begin
    totalcons:=totalcons+consumed_data[l,r];
    write(distrecords,' ',consumed_data[l,r]:3);
    end;
  writeln(distrecords);
end;
writeln(distrecords,'Data Processed');
writeln(distrecords,'---- ---------');
write(distrecords,'r=   ');
for r:=1 to ring do
  write(distrecords,' ',r:3);
writeln(distrecords);
for l:=1 to layers do
  begin
  write(distrecords,'l=',l:3);
  for r:=1 to ring do
    begin
    totalproc:=totalproc+processing[l,r];
    write(distrecords,' ',processing[l,r]:3);
    end;
  writeln(distrecords);
  end;
writeln(distrecords);writeln(distrecords);
writeln(distrecords,'Types Processed');
writeln(distrecords,'----- ---------');
for n:=1 to 50 do
  begin
  wtotproc:=wtotproc+processed_data[n]*n;
  writeln(distrecords,'Type:',n:3,' number processed:',processed_data[n]:3);
  end;
write(distrecords,'Total:  ',' number consumed:',totalcons:3);
writeln(distrecords,' number processed:',totalproc:3);
writeln(distrecords,'Total:  ',' Weighted Total Processed:',wtotproc:5);
end;  (* PRINT_RESULTS *)

begin (* DISTSTACK1 *)
a:=1234567;
(* SETS THE SEED FOR THE RANDOM NUMBER GENERATOR *)
input_parameters;
writeln('Parameters input');
initial_states;
writeln('Initial state set up');
writeln('Starting Computation');
for iterations:=1 to largen do
  begin
  communicate;
```

```
   create_new_data;
   compute;
   print_state;
   end;
writeln('Finished Computation');
writeln('Printing Results');
print_results;
end.  (* DISTSTACK1 *)
```

```
program diststack2(input,output,distdatas,distrecords);
const
(* CONSTANTS FOR RANDOM NUMBER GENERATION *)
  m=100000000;m1=10000;b=31415821;
type
(* PROCESSING NODE TYPE *)
  proc_ptr=^proc_type;
  proc_type=record
              proc_r:integer;
              proc_l:integer;
              ring_input:integer;
              ring_output:integer;
              input_full:boolean;
              output_full:boolean;
              data_input:integer;
              down_output:integer;
              data_full:boolean;
              down_full:boolean;
              p_type:integer;
              p_state:integer;
              next:proc_ptr;
              down:proc_ptr;
            end;

var
  a:integer;
  pconst:integer;
  distdatas,distrecords:text;
  stack:proc_ptr;
  processed_data:array[1..100] of integer;
  processing:array[1..20,1..20] of integer;
  consumed_data:array[1..20,1..20] of integer;
  largen,iterations,ring,layers:integer;

function mult(p,q:integer):integer;
(* EXTENDED MULTIPLICATION *)
var p1,p0,q1,q0:integer;
begin
p1:=p div m1;  p0:=p mod m1;
q1:=q div m1;  q0:=q mod m1;
mult:=(((p0*q1+p1*q0) mod m1)*m1+p0*q0) mod m;
end;

function random:real;
(* RANDOM NUMBER GENERATOR *)
(* PG37 - ALGORITHMS - SEDGEWICK *)
(* ADDISON - WESLEY 1984 *)
begin
a:=(mult(a,b)+1)mod m;
random:=a/m;
end;

procedure input_parameters;
(* INITIALISE INPUT AND OUTPUT FILES *)
(* READ IN THE PARAMETERS OF THE RING TO MODEL *)
(* WRITES A HEADING TO STACKRECORD *)
var
  l,r,n:integer;
```

```
begin (* INPUT_PARAMETERS *)
reset(distdatas);rewrite(distrecords);
writeln(distrecords,'diststack2');
writeln(distrecords,'Stack State record for simulation using :-');
(* READ IN THE NUMBER OF PROCESSORS PER RING *)
readln(distdatas,ring);
writeln(distrecords,'Number of processors per layer = ',ring:3);
(* READ IN THE NUMBER OF LAYERS *)
readln(distdatas,layers);
writeln(distrecords,'Number of layers = ',layers:3);
(* READ IN THE NUMBER OF ITERATIONS TO PERFORM *)
readln(distdatas,largen);
writeln(distrecords,'Number of iterations performed = ',largen:3);
(* READ IN THE AMOUNT OF PROCESSING REQUIRED PER DATA ITEM *)
readln(distdatas,pconst);
writeln(distrecords,'processing required per data item = ',pconst:3,' units');
writeln(distrecords);
end;  (* INPUT_PARAMETERS *)

procedure initial_states;
(* SETS UP THE INITIAL STATE OF THE STACK DATA STRUCTURE *)
(* AND PRINTS A HEADING TO FILE STACKRECORD *)
var
  p,pt:proc_ptr;
  r,l,n:integer;

  procedure set_vars;
  begin (* SET_VARS *)
  p^.input_full:=false;
  p^.output_full:=false;
  p^.data_full:=false;
  p^.down_full:=false;
  p^.p_type:=0;
  p^.p_state:=0;
  end;  (* SET_VARS *)

begin (* INITIAL_STATES *)
stack:=nil;
for l:=layers downto 1 do
  begin
  new(p);
  p^.down:=stack;
  stack:=p;
  p^.proc_r:=1;
  p^.proc_l:=1;
  p^.next:=p;
  set_vars;
  end;
for r:=ring downto 2 do
  begin
  pt:=stack;
  new(p);
  p^.next:=pt^.next;
  pt^.next:=p;
  p^.proc_r:=r;
  p^.proc_l:=1;
  set_vars;
  for l:=2 to layers do
```

```
    begin
    pt:=pt^.down;
    new(p^.down);
    p:=p^.down;
    p^.next:=pt^.next;
    pt^.next:=p;
    p^.proc_r:=r;
    p^.proc_l:=l;
    set_vars;
    end;
  p^.down:=nil;
  end;
  for l:=1 to 20 do
    for r:=1 to 20 do
      begin
      processing[l,r]:=0;
      consumed_data[l,r]:=0;
      end;
  for n:=1 to 50 do
    processed_data[n]:=0;
  end;  (* INITIAL_STATES *)

  procedure communicate;
  (* PERFORMS THE FUNCTION OF THE COMMUNICATION HARDWARE *)
  var
    p,pt:proc_ptr;
  begin (* COMMUNICATE *)
  pt:=stack;
  while (pt<>nil) do
  begin
  p:=pt;
  repeat
    begin
    if (p^.down<>nil) then
      if (p^.down_full and not(p^.down^.data_full)) then
        begin
        p^.down^.data_input:=p^.down_output;
        p^.down_full:=false;
        p^.down^.data_full:=true;
        end;
    if (p^.output_full and not(p^.next^.input_full)) then
      begin
      p^.next^.ring_input:=p^.ring_output;
      p^.next^.input_full:=true;
      p^.output_full:=false;
      end;
      p:=p^.next;
      end
    until (p=pt);
    pt:=pt^.down;
    end;
  end;  (* COMMUNICATE *)

  procedure create_new_data;
  (* PERFOMS THE FUNCTION OF THE DATA INPUT MECHANISM *)
  var
    p:proc_ptr;
    l,r:integer;
```

```
begin (* CREATE_NEW_DATA*)
p:=stack;
repeat
  begin
  if (not(p^.data_full)) then
    begin
    r:=1+trunc(random*(ring-0.0001));
    l:=1+trunc(random*(layers-0.0001));
    p^.data_input:=l*100+r;
    consumed_data[l,r]:=consumed_data[l,r]+1;
    p^.data_full:=true;
    end;
  p:=p^.next;
  end;
until (p=stack);
end; (* CREATE_NEW_DATA *)

procedure compute;
(* PERFORMS THE FUNCTION OF THE PROCESSING ELEMENT *)
var
  p,pt:proc_ptr;

  procedure computing_algorithm;
  (* PERFORMS THE FUNCTION OF THE COMMUNICATING PROCESSES WITHIN *)
  (* THE PROCESSOR *)
  var
    time_unit:integer;

  procedure comms;
  (* PERFORMS THE ACTIVITIES OF THE COMMUNICATION PROCESS *)

    function processor_idle:boolean;
    begin
    processor_idle:=(p^.p_state<=0);
    end;

    function ring_data:boolean;
    begin
    ring_data:=p^.input_full;
    end;

    function new_data:boolean;
    begin
    new_data:=p^.data_full;
    end;

      function new_data_right:boolean;
      begin
      new_data_right:=(p^.data_input=(p^.proc_r+(100*p^.proc_l)));
      end;

      function ring_data_right:boolean;
      begin
      ring_data_right:=(p^.ring_input=(p^.proc_r+(100*p^.proc_l)));
      end;

      function column_right_data:boolean;
```

```
begin
column_right_data:=
  (p^.proc_r=(p^.data_input-(100*trunc(p^.data_input/100))));
end;

function column_right_ring:boolean;
begin
column_right_ring:=
  (p^.proc_r=(p^.ring_input-(100*trunc(p^.ring_input/100))));
end;

function level_right_data:boolean;
begin
level_right_data:=(p^.proc_l=(p^.data_input div 100));
end;

function level_right_ring:boolean;
begin
level_right_ring:=(p^.proc_l=(p^.ring_input div 100));
end;

function ring_ready:boolean;
begin
ring_ready:=not(p^.output_full);
end;

function down_ready:boolean;
begin
down_ready:=not(p^.down_full);
end;

procedure take_ring_data;
begin
if (pconst<0) then
  p^.p_state:=trunc(random*(-pconst))
else
  p^.p_state:=pconst;
p^.p_type:=p^.p_state;
p^.input_full:=false;
time_unit:=time_unit-1;
end;

procedure take_new_data;
begin
if (pconst<0) then
  p^.p_state:=trunc(random*(-pconst))
else
  p^.p_state:=pconst;
p^.p_type:=p^.p_state;
p^.data_full:=false;
time_unit:=time_unit-1;
end;

procedure ring_data_on;
begin
p^.ring_output:=p^.ring_input;
p^.input_full:=false;
p^.output_full:=true;
```

```
time_unit:=time_unit-1;
end;

procedure ring_data_down;
begin
p^.down_output:=p^.ring_input;
p^.input_full:=false;
p^.down_full:=true;
time_unit:=time_unit-1;
end;

  procedure new_data_on;
  begin
  p^.ring_output:=p^.data_input;
  p^.data_full:=false;
  p^.output_full:=true;
  time_unit:=time_unit-1;
  end;

  procedure new_data_down;
  begin
  p^.down_output:=p^.data_input;
  p^.data_full:=false;
  p^.down_full:=true;
  time_unit:=time_unit-1;
  end;

begin (* COMMS2 *)
if processor_idle then
if ring_data then
  if ring_data_right then
    take_ring_data
  else
    if new_data then
      if new_data_right then
        take_new_data
      else
        (* NULL *)
    else
      (* NULL *)
else
  if new_data then
    if new_data_right then
      take_new_data
    else
      (* NULL *)
  else
      (* NULL *)
else
  (* NULL *);
if ring_ready then
  if ring_data then
    if level_right_ring and not column_right_ring then
      ring_data_on
    else
      if new_data then
        if level_right_data and not column_right_data then
          new_data_on
```

```
          else
            (* NULL *)
        else
          (* NULL *)
    else
      if new_data then
        if level_right_data and not column_right_data then
          new_data_on
        else
          (* NULL *)
      else
        (* NULL *)
else
  (* NULL *);
if down_ready then
  if ring_data then
    if not level_right_ring then
      ring_data_down
    else
      if new_data then
        if not level_right_data then
          new_data_down
        else
          (* NULL *)
      else
        (* NULL *)
  else
    if new_data then
      if not level_right_data then
        new_data_down
      else
        (* NULL *)
    else
      (* NULL *)
else
  (* NULL *)
end;  (* COMMS2 *)

procedure process(units:integer);
begin (* PROCESS *)
p^.p_state:=p^.p_state-units;
if (p^.p_state<=0) then
  begin
  if (p^.p_type>0) then
    begin
    processed_data[p^.p_type]:=processed_data[p^.p_type]+1;
    processing[p^.proc_l,p^.proc_r]:=
    processing[p^.proc_l,p^.proc_r]+1;
    end;
  p^.p_type:=0;
  p^.p_state:=0;
  end;
end;  (* PROCESS *)

begin (* COMPUTING_ALGORITHM *)
time_unit:=4;
(* TOTAL PROCESSING EFFORT AVAILABLE *)
(* PER ITERATION *)
```

```
  comms;
  (* COMMS USES UP SOME PROCESSING EFFORT *)
  process(time_unit);
  end;  (* COMPUTING_ALGORITHM *)

begin (* COMPUTE *)
pt:=stack;
while (pt<>nil) do
  begin
  p:=pt;
  repeat
    begin
    computing_algorithm;
    p:=p^.next;
    end
  until (p=pt);
  pt:=pt^.down;
  end;
end;  (* COMPUTE *)

procedure print_state;
(* PRINTS THE STATE OF THE STACK AFTER THE LAST ITERATION *)
var p,pt:proc_ptr;
    l,r:integer;
begin (* PRINT_STATE *)
write(distrecords,'Iteration:',iterations:3);
writeln(distrecords);write(distrecords,'r=   ');
for r:=1 to ring do
  write(distrecords,' ',r:3);
writeln(distrecords);
pt:=stack;
l:=1;
while(pt<>nil) do
  begin
  p:=pt;
  write(distrecords,'l=',l:3);
  repeat
    begin
    write(distrecords,' ',p^.p_state:3);
    p:=p^.next;
    end;
  until (p=pt);
  pt:=pt^.down;
  l:=l+1;
  writeln(distrecords);
  end;
  writeln(distrecords);
  end;  (* PRINT_STATE *)

  procedure print_results;
  var
    l,r,n,totproc,totcons,wtotproc:integer;
  begin (* PRINT_RESULTS *)
  totproc:=0;totcons:=0;wtotproc:=0;
  writeln(distrecords);writeln(distrecords,'Data Consumed');
  writeln(distrecords,'---- --------');writeln(distrecords);
  write(distrecords,'r=   ');
  for r:= 1 to ring do
```

```
  write(distrecords,' ',r:3);
writeln(distrecords);
for l:=1 to layers do
  begin
  write(distrecords,'l=',l:3);
  for r:=1 to ring do
    begin
    totcons:=totcons+consumed_data[l,r];
    write(distrecords,' ',consumed_data[l,r]:3);
    end;
  writeln(distrecords);
end;
writeln(distrecords,'Data Processed');
writeln(distrecords,'---- ----------');
write(distrecords,'r=   ');
for r:=1 to ring do
  write(distrecords,' ',r:3);
writeln(distrecords);
for l:=1 to layers do
  begin
  write(distrecords,'l=',l:3);
  for r:=1 to ring do
    begin
    totproc:=totproc+processing[l,r];
    write(distrecords,' ',processing[l,r]:3);
    end;
  writeln(distrecords);
  end;
writeln(distrecords);writeln(distrecords);
  writeln(distrecords,'Types Processed');
  writeln(distrecords,'----- ----------');
  for n:=1 to 50 do
    begin
    wtotproc:=wtotproc+processed_data[n]*n;
    writeln(distrecords,'Type:',n:3,' number processed:',processed_data[n]:3);
    end;
  write(distrecords,'Total:  ',' number consumed:',totcons:3);
  writeln(distrecords,' number processed:',totproc:3);
  writeln(distrecords,'Total:  ',' Weighted Total Processed:',wtotproc:5);
  end;  (* PRINT_RESULTS *)

begin (* DISTSTACK2 *)
a:=1234567;
(* SETS THE SEED FOR THE RANDOM NUMBER GENERATOR *)
input_parameters;
writeln('Parameters input');
initial_states;
writeln('Initial state set up');
writeln('Starting Computation');
for iterations:=1 to largen do
  begin
  communicate;
  create_new_data;
  compute;
  print_state;
  end;
writeln('Finished Computation');
writeln('Printing Results');
```

```
print_results;
end.  (* DISTSTACK2 *)
```

APPENDIX 3

3 Simulation Programs of Homogeneous Rings

```
program ringprog1(input,output,ringdata,ringrecord);
const
(* CONSTANTS FOR RANDOM NUMBER GENERATION *)
  m=100000000;m1=10000;b=31415821;
type
(* PROCESSING NODE TYPE *)
  proc_ptr=^proc_type;
  proc_type=record
              proc_id:integer;
              ring_input:integer;
              ring_output:integer;
              input_full:boolean;
              output_full:boolean;
              data_input:integer;
              data_full:boolean;
              p_type:integer;
              p_state:integer;
              next:proc_ptr;
              end;

var
  a:integer;
ringdata,ringrecord:text;
ring:proc_ptr;
data_types:array[1..50] of integer;
processed_data:array[1..100] of integer;
consumed_data:array[1..100] of integer;
largen,iterations,procs:integer;

function mult(p,q:integer):integer;
var p1,p0,q1,q0:integer;
begin
p1:=p div m1;  p0:=p mod m1;
q1:=q div m1;  q0:=q mod m1;
mult:=(((p0*q1+p1*q0) mod m1)*m1+p0*q0) mod m;
end;

function random:real;
(* RANDOM NUMBER GENERATOR *)
(* PG37 - ALGORITHMS - SEDGEWICK *)
(* ADDISON - WESLEY 1984 *)
begin
a:=(mult(a,b)+1)mod m;
random:=a/m;
end;

procedure input_parameters;
(* INITIALISE INPUT AND OUTPUT FILES *)
(* READ IN THE PARAMETERS OF THE RING TO MODEL *)
var
  n:integer;
begin (* INPUT_PARAMETERS *)
```

```
reset(ringdata);rewrite(ringrecord);
writeln(ringrecord,'ringprog1');
writeln(ringrecord,'Ring State record for simulation using :-');
(* READ IN THE NUMBER OF PROCESSORS *)
readln(ringdata,procs);
writeln(ringrecord,'Number of processors = ',procs:3);
(* READ IN THE DATA_TYPE (PROCESSING TIME) FOR EACH PROCESSOR *)
readln(ringdata,largen);
writeln(ringrecord,'Number of iterations performed = ',largen:3);
writeln(ringrecord,'Data types fed to processors');writeln;
for n:=1 to procs do
  begin
  readln(ringdata,data_types[n]);
  writeln(ringrecord,'Processor :',n:3,' Data type :',data_types[n]:3);
  end;
writeln;
end;  (* INPUT_PARAMETERS *)

procedure initial_states;
(* SETS UP THE INITIAL STATE OF THE RING DATA STRUCTURE *)
(* AND PRINTS A HEADING TO FILE RINGRECORD *)
var
  p:proc_ptr;
  n:integer;

  procedure set_vars;
  begin (* SET_VARS *)
  p^.input_full:=false;
  p^.output_full:=false;
  p^.data_full:=false;
  p^.p_type:=0;
  p^.p_state:=0;
  end;  (* SET_VARS *)

begin (* INITIAL_STATES *)
write(ringrecord,'     ');
new(ring);
p:=ring;
p^.proc_id:=1;
write(ringrecord,' ',p^.proc_id:3);
set_vars;
for n:=1 to (procs-1) do
  begin
  new(p^.next);
  p:=p^.next;
  p^.proc_id:=(n+1);
  write(ringrecord,' ',p^.proc_id:3);
  set_vars;
  end;
p^.next:=ring;
writeln(ringrecord);
write(ringrecord,'Data ');
for n:=1 to procs do
  write(ringrecord,' ',data_types[n]:3);
writeln(ringrecord);
for n:=1 to 50 do
  begin
  processed_data[n]:=0;
```

```
  consumed_data[n]:=0;
  end;
end;  (* INITIAL_STATES *)

procedure communicate;
(* PERFORMS THE FUNCTION OF THE COMMUNICATION HARDWARE *)
var
  p:proc_ptr;
begin (* COMMUNICATE *)
p:=ring;
repeat
  begin
  if (p^.output_full and not(p^.next^.input_full)) then
    begin
    p^.next^.ring_input:=p^.ring_output;
    p^.next^.input_full:=true;
    p^.output_full:=false;
    end;
  p:=p^.next;
  end;
until (p=ring);
end;  (* COMMUNICATE *)

procedure create_new_data;
(* PERFOMS THE FUNCTION OF THE DATA INPUT MECHANISM *)
var
  p:proc_ptr;
begin (* CREATE_NEW_DATA *)
p:=ring;
repeat
  begin
  if (not(p^.data_full) and (data_types[p^.proc_id]<>0)) then
    begin
    if (data_types[p^.proc_id]<0) then
      p^.data_input:=1+trunc(random*(-data_types[p^.proc_id])-0.0001)
    else
      p^.data_input:=data_types[p^.proc_id];
    consumed_data[p^.data_input]:=
      consumed_data[p^.data_input]+1;
    p^.data_full:=true;
    end;
  p:=p^.next;
  end;
until (p=ring);
end;  (* CREATE_NEW_DATA *)

  procedure compute;
  (* PERFORMS THE FUNCTION OF THE PROCESSING ELEMENT *)
  var
    p:proc_ptr;

    procedure computing_algorithm;
    (* PERFORMS THE FUNCTION OF THE COMMUNICATING PROCESSES WITHIN *)
    (* THE PROCESSOR *)
    var
      time_unit:integer;

      procedure comms;
```

```
(* PERFORMS THE ACTIVITIES OF THE COMMUNICATION PROCESS *)

  function processor_idle:boolean;
  begin
  processor_idle:=(p^.p_state<=0);
  end;

  function ring_data:boolean;
  begin
  ring_data:=p^.input_full;
  end;

  function new_data:boolean;
  begin
  new_data:=p^.data_full;
  end;

  function ring_output_ready:boolean;
  begin
  ring_output_ready:=not(p^.output_full);
  end;

  procedure take_ring_data;
  begin
  p^.p_type:=p^.ring_input;
  p^.p_state:=p^.p_type;
  p^.input_full:=false;
  time_unit:=time_unit-1;
  end;

  procedure take_new_data;
  begin
  p^.p_type:=p^.data_input;
  p^.p_state:=p^.p_type;
  p^.data_full:=false;
  time_unit:=time_unit-1;
  end;

  procedure ring_data_on;
  begin
  p^.ring_output:=p^.ring_input;
  p^.input_full:=false;
  p^.output_full:=true;
  time_unit:=time_unit-1;
  end;

    procedure new_data_on;
    begin
    p^.ring_output:=p^.data_input;
    p^.data_full:=false;
    p^.output_full:=true;
    time_unit:=time_unit-1;
    end;

  begin (* COMMS1 *)
  if processor_idle then
    if new_data then
      take_new_data
```

```
    else
      if ring_data then
        take_ring_data
      else
        (* NULL *)
  else
    (* NULL *);
  if ring_output_ready then
    if new_data then
      new_data_on
    else
      if ring_data then
        ring_data_on
      else
        (* NULL *)
  else
    (* NULL *);
  end;  (* COMMS1 *)

  procedure process(units:integer);
  (* PERFORMS THE ACTIVITIES OF THE PROCESSING PROCESS *)
  begin (* PROCESS *)
  p^.p_state:=p^.p_state-units;
  if (p^.p_state<=0) then
    begin
    if (p^.p_type>0) then
      processed_data[p^.p_type]:=processed_data[p^.p_type]+1;
    p^.p_type:=0;
    p^.p_state:=0;
      end;
    end;  (* PROCESS *)

    begin (* COMPUTING_ALGORITHM *)
    time_unit:=4;
    (* TOTAL PROCESSING EFFORT AVAILABLE *)
    (* PER ITERATION *)
    comms;
    (* COMMS USES UP SOME PROCESSING EFFORT *)
    process(time_unit);
    end;  (* COMPUTING_ALGORITHM *)

  begin (* COMPUTE *)
  p:=ring;
  repeat
    begin
    computing_algorithm;
    p:=p^.next;
    end;
  until (p=ring);
  end;  (* COMPUTE *)

  procedure print_state;
  (* PRINTS THE STATE OF THE RING AFTER THE LAST ITERATION *)
  var p:proc_ptr;
  begin (* PRINT_STATE *)
  write(ringrecord,'I=',iterations:3);
  p:=ring;
  repeat
```

```
  begin
  write(ringrecord,' ',p^.p_state:3);
  p:=p^.next;
  end;
until (p=ring);
writeln(ringrecord);
end;  (* PRINT_STATE *)

procedure print_consumption;
var
  n,totproc,totcons,wtotproc:integer;
begin (* PRINT_CONSUMPTION *)
totproc:=0;totcons:=0;wtotproc:=0;
writeln(ringrecord);
writeln(ringrecord);writeln(ringrecord,'Data Consumed and Processed');
writeln(ringrecord,'---- -------- --- ----------');writeln(ringrecord);
for n:=1 to 50 do
  begin
  totcons:=totcons+consumed_data[n];
  totproc:=totproc+processed_data[n];
  wtotproc:=wtotproc+processed_data[n]*n;
  write(ringrecord,'Type:',n:3,' number consumed:',consumed_data[n]:3);
  writeln(ringrecord,' number processed:',processed_data[n]:3);
  end;
write(ringrecord,'Total:  ',' number consumed:',totcons:3);
writeln(ringrecord,' number processed:',totproc:3);
writeln(ringrecord,'Total:  ',' Weighted Total Processed:',wtotproc:5);
end;  (* PRINT_CONSUMPTION *)

begin (* RINGPROG1 *)
a:=1234567;
(* SETS THE SEED FOR THE RANDOM NUMBER GENERATOR *)
input_parameters;
writeln('Parameters input');
initial_states;
writeln('Initial state set up');
writeln('Starting Computation');
for iterations:=1 to largen do
  begin
  communicate;
  create_new_data;
  compute;
  print_state;
  end;
writeln('Finished Computation');
writeln('Printing Results');
print_consumption;
end.  (* RINGPROG1 *)
```

```
program ringprog2(input,output,ringdata,ringrecord);
const
(* CONSTANTS FOR RANDOM NUMBER GENERATION *)
  m=100000000;m1=10000;b=31415821;
type
(* PROCESSING NODE TYPE *)
  proc_ptr=^proc_type;
  proc_type=record
              proc_id:integer;
              ring_input:integer;
              ring_output:integer;
              input_full:boolean;
              output_full:boolean;
              data_input:integer;
              data_full:boolean;
              p_type:integer;
              p_state:integer;
              next:proc_ptr;
              end;

var
  a:integer;
  ringdata,ringrecord:text;
  ring:proc_ptr;
  data_types:array[1..50] of integer;
  processed_data:array[1..100] of integer;
  consumed_data:array[1..100] of integer;
  largen,iterations,procs:integer;

function mult(p,q:integer):integer;
(* EXTENDED MULTIPLICATION *)
var p1,p0,q1,q0:integer;
begin
p1:=p div m1;  p0:=p mod m1;
q1:=q div m1;  q0:=q mod m1;
mult:=(((p0*q1+p1*q0) mod m1)*m1+p0*q0) mod m;
end;

function random:real;
(* RANDOM NUMBER GENERATOR *)
(* PG37 - ALGORITHMS - SEDGEWICK *)
(* ADDISON - WESLEY 1984 *)
begin
a:=(mult(a,b)+1)mod m;
random:=a/m;
end;

procedure input_parameters;
(* INITIALISE INPUT AND OUTPUT FILES *)
(* READ IN THE PARAMETERS OF THE RING TO MODEL *)
var
  n:integer;
begin (* INPUT_PARAMETERS *)
reset(ringdata);rewrite(ringrecord);
writeln(ringrecord,'ringprog2');
writeln(ringrecord,'Ring State record for simulation using :-');
(* READ IN THE NUMBER OF PROCESSORS *)
readln(ringdata,procs);
```

```
writeln(ringrecord,'Number of processors = ',procs:3);
(* READ IN THE DATA_TYPE (PROCESSING TIME) FOR EACH PROCESSOR *)
readln(ringdata,largen);
writeln(ringrecord,'Number of iterations performed = ',largen:3);
writeln(ringrecord,'Data types fed to processors');writeln;
for n:=1 to procs do
  begin
  readln(ringdata,data_types[n]);
  writeln(ringrecord,'Processor :',n:3,' Data type :',data_types[n]:3);
  end;
writeln;
end;  (* INPUT_PARAMETERS *)

procedure initial_states;
(* SETS UP THE INITIAL STATE OF THE RING DATA STRUCTURE *)
(* AND PRINTS A HEADING TO FILE RINGRECORD *)
var
  p:proc_ptr;
  n:integer;

  procedure set_vars;
  begin (* SET_VARS *)
  p^.input_full:=false;
  p^.output_full:=false;
  p^.data_full:=false;
  p^.p_type:=0;
  p^.p_state:=0;
  end;  (* SET_VARS *)

begin (* INITIAL_STATES *)
write(ringrecord,'     ');
new(ring);
p:=ring;
p^.proc_id:=1;
write(ringrecord,' ',p^.proc_id:3);
set_vars;
for n:=1 to (procs-1) do
  begin
  new(p^.next);
  p:=p^.next;
  p^.proc_id:=(n+1);
  write(ringrecord,' ',p^.proc_id:3);
  set_vars;
  end;
p^.next:=ring;
writeln(ringrecord);
write(ringrecord,'Data ');
for n:=1 to procs do
  write(ringrecord,' ',data_types[n]:3);
writeln(ringrecord);
for n:=1 to 50 do
  begin
  processed_data[n]:=0;
  consumed_data[n]:=0;
  end;
end;  (* INITIAL_STATES *)

procedure communicate;
```

```
(* PERFORMS THE FUNCTION OF THE COMMUNICATION HARDWARE *)
var
  p:proc_ptr;
begin (* COMMUNICATE *)
p:=ring;
repeat
  begin
  if (p^.output_full and not(p^.next^.input_full)) then
    begin
    p^.next^.ring_input:=p^.ring_output;
    p^.next^.input_full:=true;
    p^.output_full:=false;
    end;
  p:=p^.next;
  end;
until (p=ring);
end;  (* COMMUNICATE *)

procedure create_new_data;
(* PERFOMS THE FUNCTION OF THE DATA INPUT MECHANISM *)
var
  p:proc_ptr;
begin (* CREATE_NEW_DATA *)
p:=ring;
repeat
  begin
  if (not(p^.data_full) and (data_types[p^.proc_id]<>0)) then
    begin
    if (data_types[p^.proc_id]<0) then
      p^.data_input:=1+trunc(random*(-data_types[p^.proc_id]-0.0001))
    else
      p^.data_input:=data_types[p^.proc_id];
    consumed_data[p^.data_input]:=
      consumed_data[p^.data_input]+1;
    p^.data_full:=true;
    end;
  p:=p^.next;
  end;
until (p=ring);
end;  (* CREATE_NEW_DATA *)

procedure compute;
(* PERFORMS THE FUNCTION OF THE PROCESSING ELEMENT *)
var
  p:proc_ptr;

  procedure computing_algorithm;
  (* PERFORMS THE FUNCTION OF THE COMMUNICATING PROCESSES WITHIN *)
  (* THE PROCESSOR *)
  var
    time_unit:integer;

    procedure comms;
    (* PERFORMS THE ACTIVITIES OF THE COMMUNICATION PROCESS *)

      function processor_idle:boolean;
      begin
      processor_idle:=(p^.p_state<=0);
```

```
end;

function ring_data:boolean;
begin
ring_data:=p^.input_full;
end;

function new_data:boolean;
begin
new_data:=p^.data_full;
end;

function ring_output_ready:boolean;
begin
ring_output_ready:=not(p^.output_full);
end;

procedure take_ring_data;
begin
p^.p_type:=p^.ring_input;
p^.p_state:=p^.p_type;
p^.input_full:=false;
time_unit:=time_unit-1;
end;

procedure take_new_data;
begin
p^.p_type:=p^.data_input;
p^.p_state:=p^.p_type;
p^.data_full:=false;
time_unit:=time_unit-1;
end;

procedure ring_data_on;
begin
p^.ring_output:=p^.ring_input;
p^.input_full:=false;
p^.output_full:=true;
time_unit:=time_unit-1;
end;

procedure new_data_on;
begin
p^.ring_output:=p^.data_input;
p^.data_full:=false;
p^.output_full:=true;
time_unit:=time_unit-1;
end;

begin (* COMMS2 *)
if processor_idle then
  if ring_data then
    take_ring_data
  else
    if new_data then
      take_new_data
    else
      (* NULL *)
```

```
else
  (* NULL *);
if ring_output_ready then
  if ring_data then
    ring_data_on
  else
    if new_data then
      new_data_on
    else
      (* NULL *)
else
  (* NULL *);
  end;  (* COMMS2 *)

  procedure process(units:integer);
  (* PERFORMS THE ACTIVITIES OF THE PROCESSING PROCESS *)
  begin (* PROCESS *)
  p^.p_state:=p^.p_state-units;
  if (p^.p_state<=0) then
    begin
    if (p^.p_type>0) then
      processed_data[p^.p_type]:=processed_data[p^.p_type]+1;
    p^.p_type:=0;
    p^.p_state:=0;
    end;
  end;  (* PROCESS *)

begin (* COMPUTING_ALGORITHM *)
time_unit:=4;
  (* TOTAL PROCESSING EFFORT AVAILABLE *)
  (* PER ITERATION *)
  comms;
  (* COMMS USES UP SOME PROCESSING EFFORT *)
  process(time_unit);
  end;  (* COMPUTING_ALGORITHM *)

begin (* COMPUTE *)
p:=ring;
repeat
  begin
  computing_algorithm;
  p:=p^.next;
  end;
until (p=ring);
end;  (* COMPUTE *)

procedure print_state;
(* PRINTS THE STATE OF THE RING AFTER THE LAST ITERATION *)
var p:proc_ptr;
begin (* PRINT_STATE *)
write(ringrecord,'I=',iterations:3);
p:=ring;
repeat
  begin
  write(ringrecord,' ',p^.p_state:3);
  p:=p^.next;
  end;
until (p=ring);
```

```
writeln(ringrecord);
end;  (* PRINT_STATE *)

procedure print_consumption;
var
  n,totproc,totcons,wtotproc:integer;
begin (* PRINT_CONSUMPTION *)
totproc:=0;totcons:=0;wtotproc:=0;
writeln(ringrecord);
writeln(ringrecord);writeln(ringrecord,'Data Consumed and Processed');
writeln(ringrecord,'---- -------- --- ----------');writeln(ringrecord);
for n:=1 to 50 do
    begin
    totcons:=totcons+consumed_data[n];
    totproc:=totproc+processed_data[n];
    wtotproc:=wtotproc+processed_data[n]*n;
    write(ringrecord,'Type:',n:3,' number consumed:',consumed_data[n]:3);
    writeln(ringrecord,' number processed:',processed_data[n]:3);
    end;
  write(ringrecord,'Total:  ',' number consumed:',totcons:3);
  writeln(ringrecord,' number processed:',totproc:3);
  writeln(ringrecord,'Total:  ',' Weighted Total Processed:',wtotproc:5);
  end;  (* PRINT_CONSUMPTION *)

begin (* RINGPROG2 *)
a:=1234567;
(* SET THE SEED FOR THE RANDOM NUMBER GENERATOR *)
input_parameters;
writeln('Parameters input');
initial_states;
writeln('Initial state set up');
writeln('Starting Computation');
for iterations:=1 to largen do
  begin
  communicate;
  create_new_data;
  compute;
  print_state;
  end;
writeln('Finished Computation');
writeln('Printing Results');
print_consumption;
end.  (* RINGPROG2 *)
```

```
program ringprog1proc(input,output,ringdata,ringrecord);
const
(* CONSTANTS FOR RANDOM NUMBER GENERATION *)
  m=100000000;m1=10000;b=31415821;
type
(* PROCESSING NODE TYPE *)
  proc_ptr=^proc_type;
  proc_type=record
              proc_id:integer;
              ring_input:integer;
              ring_output:integer;
              input_full:boolean;
              output_full:boolean;
              data_input:integer;
              data_full:boolean;
              p_type:integer;
              p_state:integer;
              next:proc_ptr;
              end;

var
  a:integer;
  ringdata,ringrecord:text;
  ring:proc_ptr;
  data_types:array[1..50] of integer;
  processed_data:array[1..50] of integer;
  consumed_data:array[1..50] of integer;
  largen,iterations,procs:integer;

  function mult(p,q:integer):integer;
  (* EXTENDED MULTIPLICATION *)
  var p1,p0,q1,q0:integer;
  begin
  p1:=p div m1;  p0:=p mod m1;
  q1:=q div m1;  q0:=q mod m1;
  mult:=(((p0*q1+p1*q0) mod m1)*m1+p0*q0) mod m;
  end;

  function random:real;
  (* RANDOM NUMBER GENERATOR *)
  (* PG37 - ALGORITHMS - SEDGEWICK *)
  (* ADDISON - WESLEY 1984 *)
  begin
  a:=(mult(a,b)+1)mod m;
  random:=a/m;
  end;

  procedure input_parameters;
  (* INITIALISE INPUT AND OUTPUT FILES *)
  (* READ IN THE PARAMETERS OF THE RING TO MODEL *)
  var
    n:integer;
  begin (* INPUT_PARAMETERS *)
  reset(ringdata);rewrite(ringrecord);
  writeln(ringrecord,'ringprog1-processing done preferential');
  writeln(ringrecord,'Ring State record for simulation using :-');
  (* READ IN THE NUMBER OF PROCESSORS *)
  readln(ringdata,procs);
```

```
writeln(ringrecord,'Number of processors = ',procs:3);
(* READ IN THE DATA_TYPE (PROCESSING TIME) FOR EACH PROCESSOR *)
readln(ringdata,largen);
writeln(ringrecord,'Number of iterations performed = ',largen:3);
writeln(ringrecord,'Data types fed to processors');writeln;
for n:=1 to procs do
  begin
  readln(ringdata,data_types[n]);
  writeln(ringrecord,'Processor :',n:3,' Data type :',data_types[n]:3);
  end;
writeln;
end;  (* INPUT_PARAMETERS *)

procedure initial_states;
(* SETS UP THE INITIAL STATE OF THE RING DATA STRUCTURE *)
(* AND PRINTS A HEADING TO FILE RINGRECORD *)
var
  p:proc_ptr;
  n:integer;

  procedure set_vars;
  begin (* SET_VARS *)
  p^.input_full:=false;
  p^.output_full:=false;
  p^.data_full:=false;
  p^.p_type:=0;
  p^.p_state:=0;
  end;  (* SET_VARS *)

begin (* INITIAL_STATES *)
write(ringrecord,'     ');
new(ring);
p:=ring;
p^.proc_id:=1;
write(ringrecord,' ',p^.proc_id:3);
set_vars;
for n:=1 to (procs-1) do
  begin
  new(p^.next);
  p:=p^.next;
  p^.proc_id:=(n+1);
  write(ringrecord,' ',p^.proc_id:3);
  set_vars;
  end;
p^.next:=ring;
writeln(ringrecord);
write(ringrecord,'Data ');
for n:=1 to procs do
  write(ringrecord,' ',data_types[n]:3);
writeln(ringrecord);
for n:=1 to 50 do
  begin
  processed_data[n]:=0;
  consumed_data[n]:=0;
  end;
end;  (* INITIAL_STATES *)

procedure communicate;
```

```
(* PERFORMS THE FUNCTION OF THE COMMUNICATION HARDWARE *)
var
  p:proc_ptr;
begin (* COMMUNICATE *)
p:=ring;
repeat
  begin
  if (p^.output_full and not(p^.next^.input_full)) then
    begin
    p^.next^.ring_input:=p^.ring_output;
    p^.next^.input_full:=true;
    p^.output_full:=false;
    end;
  p:=p^.next;
  end;
until (p=ring);
end;  (* COMMUNICATE *)

procedure create_new_data;
(* PERFOMS THE FUNCTION OF THE DATA INPUT MECHANISM *)
var
  p:proc_ptr;
begin (* CREATE_NEW_DATA *)
p:=ring;
repeat
  begin
  if (not(p^.data_full) and (data_types[p^.proc_id]<>0)) then
    begin
    if (data_types[p^.procid]<0) then
      p^.data_input:=1+trunc(random*(-data_types[p^.proc_id]-0.0001))
    else
      p^.data_input:=data_types[p^.proc_id];
    consumed_data[p^.data_input]:=
      consumed_data[p^.data_input]+1;
    p^.data_full:=true;
    end;
  p:=p^.next;
  end;
until (p=ring);
end;  (* CREATE_NEW_DATA *)

procedure compute;
(* PERFORMS THE FUNCTION OF THE PROCESSING ELEMENT *)
var
  p:proc_ptr;

  procedure computing_algorithm;
  (* PERFORMS THE FUNCTION OF THE COMMUNICATING PROCESSES WITHIN *)
  (* THE PROCESSOR *)
  var
    time_unit:integer;

    procedure comms(time_unit:integer);
    (* PERFORMS THE ACTIVITIES OF THE COMMUNICATION PROCESS *)

      function processor_idle:boolean;
      begin
      processor_idle:=(p^.p_state<=0);
```

```
end;

function ring_data:boolean;
begin
ring_data:=p^.input_full;
end;

function new_data:boolean;
begin
new_data:=p^.data_full;
end;

function ring_output_ready:boolean;
begin
ring_output_ready:=not(p^.output_full);
end;

procedure take_ring_data;
begin
p^.p_type:=p^.ring_input;
p^.p_state:=p^.p_type;
p^.input_full:=false;
time_unit:=time_unit-1;
end;

procedure take_new_data;
begin
p^.p_type:=p^.data_input;
p^.p_state:=p^.p_type;
p^.data_full:=false;
time_unit:=time_unit-1;
end;

procedure ring_data_on;
begin
p^.ring_output:=p^.ring_input;
p^.input_full:=false;
p^.output_full:=true;
time_unit:=time_unit-1;
end;

procedure new_data_on;
begin
p^.ring_output:=p^.data_input;
p^.data_full:=false;
p^.output_full:=true;
time_unit:=time_unit-1;
end;

begin (* COMMS1 *)
if processor_idle then
  if new_data and (time_unit>0) then
    take_new_data
  else
    if ring_data and (time_unit>0) then
      take_ring_data
    else
      (* NULL *)
```

```
else
  (* NULL *);
if ring_output_ready then
  if new_data and (time_unit>0) then
    new_data_on
  else
    if ring_data and (time_unit>0) then
      ring_data_on
    else
      (* NULL *)
else
  (* NULL *);
end;  (* COMMS1 *)

procedure process(var units:integer);
(* PERFORMS THE ACTIVITIES OF THE PROCESSING PROCESS *)
begin (* PROCESS *)
if (p^.p_state>=units) then
  begin
  p^.p_state:=p^.p_state-units;
  units:=0;
  end
else
  begin
  p^.p_state:=0;
  units:=units-p^.p_state;
  end;
if (p^.p_state<=0) then
  begin
    if (p^.p_type>0) then
      processed_data[p^.p_type]:=processed_data[p^.p_type]+1;
    p^.p_type:=0;
    p^.p_state:=0;
    end;
  end;  (* PROCESS *)

begin (* COMPUTING_ALGORITHM *)
time_unit:=4;
(* TOTAL PROCESSING EFFORT AVAILABLE *)
(* PER ITERATION *)
process(time_unit);
(* PROCESSING DONE IN PREFERENCE TO COMMUNICATION *)
(* COMMS USES UP SOME PROCESSING EFFORT *)
(* BUT ONLY IF GIVEN THE CHANCE *)
comms(time_unit);
  end;  (* COMPUTING_ALGORITHM *)

  begin (* COMPUTE *)
  p:=ring;
  repeat
    begin
    computing_algorithm;
    p:=p^.next;
    end;
  until (p=ring);
  end;  (* COMPUTE *)

  procedure print_state;
```

```
(* PRINTS THE STATE OF THE RING AFTER THE LAST ITERATION *)
var p:proc_ptr;
begin (* PRINT_STATE *)
write(ringrecord,'I=',iterations:3);
p:=ring;
repeat
  begin
  write(ringrecord,' ',p^.p_state:3);
  p:=p^.next;
  end;
until (p=ring);
writeln(ringrecord);
end;  (* PRINT_STATE *)

procedure print_consumption;
var
  n,totproc,totcons:integer;
begin (* PRINT_CONSUMPTION *)
totproc:=0;totcons:=0;
writeln(ringrecord);
writeln(ringrecord);writeln(ringrecord,'Data Consumed and Processed');
writeln(ringrecord,'---- -------- --- ----------');writeln(ringrecord);
for n:=1 to 50 do
  begin
  totcons:=totcons+consumed_data[n];
  totproc:=totproc+processed_data[n];
  write(ringrecord,'Type:',n:3,' number consumed:',consumed_data[n]:3);
  writeln(ringrecord,' number processed:',processed_data[n]:3);
  end;
  write(ringrecord,'Total:  ',' number consumed:',totcons:3);
  writeln(ringrecord,' number processed:',totproc:3);
  end;  (* PRINT_CONSUMPTION *)

begin (* RINGPROG1PROC *)
a:=1234567;
(* SETS THE SEED FOR THE RANDOM NUMBER GENERATOR *)
input_parameters;
writeln('Parameters input');
initial_states;
writeln('Initial state set up');
writeln('Starting Computation');
for iterations:=1 to largen do
  begin
  communicate;
  create_new_data;
  compute;
  print_state;
  end;
writeln('Finished Computation');
writeln('Printing Results');
print_consumption;
end.  (* RINGPROG1PROC *)
```

APPENDIX 4

4 Simulation Programs of Homogeneous Cylinders

```
program stackprog1(input,output,stackdata,stackrecord);
(* CONSTANTS FOR RANDOM NUMBER GENERATION *)
const
  m=100000000;m1=10000;b=31415821;
type
(* PROCESSING NODE TYPE *)
proc_ptr=^proc_type;
proc_type=record
            proc_r:integer;
            proc_l:integer;
            ring_input:integer;
            ring_output:integer;
            input_full:boolean;
            output_full:boolean;
            data_input:integer;
            down_output:integer;
            data_full:boolean;
            down_full:boolean;
            p_type:integer;
            p_state:integer;
            next:proc_ptr;
            down:proc_ptr;
            end;
    state_type=(faulty,good);

var
  a:integer;
  stackdata,stackrecord:text;
  stack:proc_ptr;
  condition:array[1..50,1..50] of state_type;
  data_types:array[1..50] of integer;
  processed_data:array[1..200] of integer;
  consumed_data:array[1..200] of integer;
  largen,iterations,ring,layers:integer;

  function mult(p,q:integer):integer;
  (* EXTENDED MULTIPLICATION *)
  var p1,p0,q1,q0:integer;
  begin
  p1:=p div m1;  p0:=p mod m1;
  q1:=q div m1;  q0:=q mod m1;
  mult:=(((p0*q1+p1*q0) mod m1)*m1+p0*q0) mod m;
  end;

  function random:real;
  (* RANDOM NUMBER GENERATOR *)
  (* PG37 - ALGORITHMS - SEDGEWICK *)
  (* ADDISON - WESLEY 1984 *)
  begin
  a:=(mult(a,b)+1)mod m;
  random:=a/m;
  end;
```

```
procedure input_parameters;
(* INITIALISE INPUT AND OUTPUT FILES *)
(* READ IN THE PARAMETERS OF THE RING TO MODEL *)
(* WRITES A HEADING TO STACKRECORD *)
var
  l,r,n:integer;
begin (* INPUT_PARAMETERS *)
reset(stackdata);rewrite(stackrecord);
writeln(stackrecord,'stackprog1');
writeln(stackrecord,'Stack State record for simulation using :-');
(* READ IN THE NUMBER OF PROCESSORS PER RING *)
readln(stackdata,ring);
writeln(stackrecord,'Number of processors per layer = ',ring:3);
(* READ IN THE NUMBER OF LAYERS *)
readln(stackdata,layers);
writeln(stackrecord,'Number of layers = ',layers:3);
(* READ IN THE NUMBER OF ITERATIONS TO PERFORM *)
readln(stackdata,largen);
writeln(stackrecord,'Number of iterations performed = ',largen:3);
(* READ IN THE FAULTY PROCESSORS *)
writeln(stackrecord,'With faulty processors:-');
for l:=1 to 50 do
  for r:= 1 to 50 do
    condition[l,r]:=good;
while not(stackdata^='D') do
  begin
  readln(stackdata,l,r);
  writeln(stackrecord,'layer:',l:3,' ring:',r:3);
  condition[l,r]:=faulty;
  end;
readln(stackdata);
(* READ IN THE DATA_TYPE (PROCESSING TIME) FOR EACH PROCESSOR *)
writeln(stackrecord,'Data types fed to processors');writeln;
for n:=1 to ring do
  begin
  readln(stackdata,data_types[n]);
  writeln(stackrecord,'Processor :',n:3,' Data type :',data_types[n]:3);
  end;
writeln(stackrecord);
end;  (* INPUT_PARAMETERS *)

procedure initial_states;
(* SETS UP THE INITIAL STATE OF THE STACK DATA STRUCTURE *)
(* AND PRINTS A HEADING TO FILE STACKRECORD *)
var
  p,pt:proc_ptr;
  r,l,n:integer;

  procedure set_vars;
  begin (* SET_VARS *)
  p^.input_full:=false;
  p^.output_full:=false;
  p^.data_full:=false;
  p^.down_full:=false;
  p^.p_type:=0;
  p^.p_state:=0;
  end;  (* SET_VARS *)
```

```
begin (* INITIAL_STATES *)
stack:=nil;
for l:=layers downto 1 do
  begin
  new(p);
  p^.down:=stack;
  stack:=p;
  p^.proc_r:=1;
  p^.proc_l:=1;
  p^.next:=p;
  set_vars;
  end;
for r:=ring downto 2 do
  begin
  pt:=stack;
  new(p);
  p^.next:=pt^.next;
  pt^.next:=p;
  p^.proc_r:=r;
  p^.proc_l:=1;
  set_vars;
  for l:=2 to layers do
    begin
    pt:=pt^.down;
    new(p^.down);
    p:=p^.down;
    p^.next:=pt^.next;
    pt^.next:=p;
    p^.proc_r:=r;
    p^.proc_l:=l;
    set_vars;
    end;
  p^.down:=nil;
  end;
for n:=1 to 200 do
  begin
  processed_data[n]:=0;
  consumed_data[n]:=0;
  end;
end;  (* INITIAL_STATES *)

procedure communicate;
(* PERFORMS THE FUNCTION OF THE COMMUNICATION HARDWARE *)
var
  p,pt:proc_ptr;
begin (* COMMUNICATE *)
pt:=stack;
while (pt<>nil) do
  begin
  p:=pt;
  repeat
    begin
    if (p^.down<>nil) then
      if (p^.down_full and not(p^.down^.data_full)) then
        begin
        p^.down^.data_input:=p^.down_output;
        p^.down_full:=false;
```

```
        p^.down^.data_full:=true;
        end;
    if (p^.output_full and not(p^.next^.input_full)) then
      begin
      p^.next^.ring_input:=p^.ring_output;
      p^.next^.input_full:=true;
      p^.output_full:=false;
      end;
    p:=p^.next;
    end
until (p=pt);
pt:=pt^.down;
  end;
end;  (* COMMUNICATE *)

procedure create_new_data;
(* PERFOMS THE FUNCTION OF THE DATA INPUT MECHANISM *)
var
  p:proc_ptr;
begin (* CREATE_NEW_DATA *)
p:=stack;
repeat
  begin
  if (not(p^.data_full) and (data_types[p^.proc_r]<>0)) then
    begin
    if (data_types[p^.proc_r]<0) then
      p^.data_input:=1+trunc(random*(-data_types[p^.proc_r]-0.0001))
    else
      p^.data_input:=data_types[p^.proc_r];
    consumed_data[p^.data_input]:=
      consumed_data[p^.data_input]+1;
    p^.data_full:=true;
    end;
  p:=p^.next;
  end;
until (p=stack);
end; (* CREATE_NEW_DATA *)

procedure compute;
(* PERFORMS THE FUNCTION OF THE PROCESSING ELEMENT *)
var
  p,pt:proc_ptr;

  procedure computing_algorithm;
  (* PERFORMS THE FUNCTION OF THE COMMUNICATING PROCESSES WITHIN *)
  (* THE PROCESSOR *)
  var
    time_unit:integer;

    procedure comms;
    (* PERFORMS THE ACTIVITIES OF THE COMMUNICATION PROCESS *)

      function processor_idle:boolean;
      begin
      processor_idle:=(p^.p_state<=0);
      end;

      function ring_data:boolean;
```

```
begin
ring_data:=p^.input_full;
end;

function new_data:boolean;
begin
new_data:=p^.data_full;
end;

function ring_ready:boolean;
begin
ring_ready:=not(p^.output_full);
end;

function down_ready:boolean;
begin
down_ready:=not(p^.down_full);
end;

procedure take_ring_data;
begin
p^.p_type:=p^.ring_input;
p^.p_state:=p^.p_type;
p^.input_full:=false;
time_unit:=time_unit-1;
end;

procedure take_new_data;
begin
p^.p_type:=p^.data_input;
p^.p_state:=p^.p_type;
p^.data_full:=false;
time_unit:=time_unit-1;
end;

procedure ring_data_on;
begin
p^.ring_output:=p^.ring_input;
p^.input_full:=false;
p^.output_full:=true;
time_unit:=time_unit-1;
end;

procedure ring_data_down;
begin
p^.down_output:=p^.ring_input;
p^.input_full:=false;
p^.down_full:=true;
time_unit:=time_unit-1;
end;

procedure new_data_on;
begin
p^.ring_output:=p^.data_input;
p^.data_full:=false;
p^.output_full:=true;
time_unit:=time_unit-1;
end;
```

```
  procedure new_data_down;
  begin
  p^.down_output:=p^.data_input;
  p^.data_full:=false;
  p^.down_full:=true;
  time_unit:=time_unit-1;
  end;

begin (* COMMS1 *)
if processor_idle then
  if new_data then
    take new data
  else
    if ring_data then
      take_ring_data
    else
      (* NULL *)
else
  (* NULL *);
if ring_ready then
  if new_data then
    new_data_on
  else
    if ring_data then
      ring_data_on
    else
      (* NULL *)
else
  (* NULL *);
if down_ready then
  if new_data then
    new_data_down
  else
    if ring_data then
      ring_data_down
    else
      (* NULL *)
else
  (* NULL *);
end;  (* COMMS1 *)

procedure process(units:integer);
begin (* PROCESS *)
p^.p_state:=p^.p_state-units;
if (p^.p_state<=0) then
    begin
    if (p^.p_type>0) then
      processed_data[p^.p_type]:=processed_data[p^.p_type]+1;
    p^.p_type:=0;
    p^.p_state:=0;
    end;
  end;  (* PROCESS *)

begin (* COMPUTING_ALGORITHM *)
time_unit:=4;
(* TOTAL PROCESSING EFFORT AVAILABLE *)
(* PER ITERATION *)
```

```
  if (condition[p^.proc_l,p^.proc_r]=good) then
    begin
    comms;
    (* COMMS USES UP SOME PROCESSING EFFORT *)
    process(time_unit);
    end
  else
    (* NULL *)
  end;  (* COMPUTING_ALGORITHM *)

begin (* COMPUTE *)
pt:=stack;
while (pt<>nil) do
  begin
  p:=pt;
  repeat
    begin
    computing_algorithm;
    p:=p^.next;
    end
  until (p=pt);
  pt:=pt^.down;
  end;
end;  (* COMPUTE *)

procedure print_state;
(* PRINTS THE STATE OF THE STACK AFTER THE LAST ITERATION *)
var p,pt:proc_ptr;
    l,r:integer;
begin (* PRINT_STATE *)
write(stackrecord,'Iteration:',iterations:3);
writeln(stackrecord);write(stackrecord,'r=   ');
for r:=1 to ring do
  write(stackrecord,' ',r:3);
writeln(stackrecord);
pt:=stack;
write(stackrecord,'Data ');
for r:=1 to ring do
  write(stackrecord,' ',data_types[r]:3);
writeln(stackrecord);
l:=1;
while(pt<>nil) do
  begin
  p:=pt;
  write(stackrecord,'l=',l:3);
  repeat
    begin
    if (condition[p^.proc_l,p^.proc_r]=good) then
      write(stackrecord,' ',p^.p_state:3)
    else
      write(stackrecord,'   F');
    p:=p^.next;
    end;
  until (p=pt);
  pt:=pt^.down;
  l:=l+1;
  writeln(stackrecord);
  end;
```

```
writeln(stackrecord);
end;  (* PRINT_STATE *)

procedure print_consumption;
var
  n,totproc,totcons,wtotproc:integer;
begin (* PRINT_CONSUMPTION *)
totproc:=0;totcons:=0;wtotproc:=0;
writeln(stackrecord);writeln(stackrecord,'Data Consumed and Processed');
writeln(stackrecord,'---- -------- --- ----------');writeln(stackrecord);
for n:=1 to 200 do
  begin
  totcons:=totcons+consumed_data[n];
  totproc:=totproc+processed_data[n];
  wtotproc:=wtotproc+processed_data[n]*n;
  write(stackrecord,'Type:',n:3,' number consumed:',consumed_data[n]:3);
  writeln(stackrecord,' number processed:',processed_data[n]:3);
  end;
write(stackrecord,'Total:  ',' number consumed:',totcons:3);
writeln(stackrecord,' number processed:',totproc:3);
writeln(stackrecord,'Total:  ',' Weighted Total Processed:',wtotproc:5);
end;  (* PRINT_CONSUMPTION *)

begin (* STACKPROG1 *)
a:=1234567;
(* SETS THE SEED FOR THE RANDOM NUMBER GENERATOR *)
input_parameters;
writeln('Parameters input');
initial_states;
writeln('Initial state set up');
writeln('Starting Computation');
for iterations:=1 to largen do
  begin
  communicate;
  create_new_data;
  compute;
  print_state;
  end;
writeln('Finished Computation');
writeln('Printing Results');
print_consumption;
end.  (* STACKPROG1 *)
```

```
program stackprog2(input,output,stackdata,stackrecord);
const
(* CONSTANTS FOR RANDOM NUMBER GENERATOR *)
  m=100000000;m1=10000;b=31415821;
type
(* PROCESSING NODE TYPE *)
  proc_ptr=^proc_type;
  proc_type=record
              proc_r:integer;
              proc_l:integer;
              ring_input:integer;
              ring_output:integer;
              input_full:boolean;
              output_full:boolean;
              data_input:integer;
              down_output:integer;
              data_full:boolean;
              down_full:boolean;
              p_type:integer;
              p_state:integer;
              next:proc_ptr;
              down:proc_ptr;
              end;
    state_type=(faulty,good);

var
  a:integer;
  stackdata,stackrecord:text;
  stack:proc_ptr;
  condition:array[1..50,1..50] of state_type;
  data_types:array[1..50] of integer;
  processed_data:array[1..200] of integer;
  consumed_data:array[1..200] of integer;
  largen,iterations,ring,layers:integer;

function mult(p,q:integer):integer;
(* EXTENDED MULTIPLICATION *)
var p1,p0,q1,q0:integer;
begin
p1:=p div m1;  p0:=p mod m1;
q1:=q div m1;  q0:=q mod m1;
mult:=(((p0*q1+p1*q0) mod m1)*m1+p0*q0) mod m;
end;

function random:real;
(* RANDOM NUMBER GENERATOR *)
(* PG37 - ALGORITHMS - SEDGEWICK *)
(* ADDISON - WESLEY 1984 *)
begin
a:=(mult(a,b)+1)mod m;
random:=a/m;
end;

procedure input_parameters;
(* INITIALISE INPUT AND OUTPUT FILES *)
(* READ IN THE PARAMETERS OF THE RING TO MODEL *)
(* WRITES A HEADING TO STACKRECORD *)
var
```

```
  l,r,n:integer;
begin (* INPUT_PARAMETERS *)
reset(stackdata);rewrite(stackrecord);
writeln(stackrecord,'stackprog2');
writeln(stackrecord,'Stack State record for simulation using :-');
(* READ IN THE NUMBER OF PROCESSORS PER RING *)
readln(stackdata,ring);
writeln(stackrecord,'Number of processors per layer = ',ring:3);
(* READ IN THE NUMBER OF LAYERS *)
readln(stackdata,layers);
writeln(stackrecord,'Number of layers = ',layers:3);
(* READ IN THE NUMBER OF ITERATIONS TO PERFORM *)
readln(stackdata,largen);
writeln(stackrecord,'Number of iterations performed = ',largen:3);
(* READ IN THE FAULTY PROCESSORS *)
writeln(stackrecord,'With faulty processors:-');
for l:=1 to 50 do
  for r:= 1 to 50 do
    condition[l,r]:=good;
while not(stackdata^='D') do
  begin
  readln(stackdata,l,r);
  writeln(stackrecord,'layer:',l:3,' ring:',r:3);
  condition[l,r]:=faulty;
  end;
readln(stackdata);
(* READ IN THE DATA_TYPE (PROCESSING TIME) FOR EACH PROCESSOR *)
writeln(stackrecord,'Data types fed to processors');writeln;
for n:=1 to ring do
  begin
  readln(stackdata,data_types[n]);
  writeln(stackrecord,'Processor :',n:3,' Data type :',data_types[n]:3);
  end;
writeln(stackrecord);
end;  (* INPUT_PARAMETERS *)

procedure initial_states;
(* SETS UP THE INITIAL STATE OF THE STACK DATA STRUCTURE *)
(* AND PRINTS A HEADING TO FILE STACKRECORD *)
var
  p,pt:proc_ptr;
  r,l,n:integer;

  procedure set_vars;
  begin (* SET_VARS *)
  p^.input_full:=false;
  p^.output_full:=false;
  p^.data_full:=false;
  p^.down_full:=false;
  p^.p_type:=0;
  p^.p_state:=0;
  end;  (* SET_VARS *)

begin (* INITIAL_STATES *)
stack:=nil;
for l:=layers downto 1 do
  begin
  new(p);
```

```
    p^.down:=stack;
    stack:=p;
    p^.proc_r:=1;
    p^.proc_l:=1;
    p^.next:=p;
    set_vars;
    end;
  for r:=ring downto 2 do
    begin
    pt:=stack;
    new(p);
    p^.next:=pt^.next;
    pt^.next:=p;
    p^.proc_r:=r;
    p^.proc_l:=1;
    set_vars;
    for l:=2 to layers do
      begin
      pt:=pt^.down;
      new(p^.down);
      p:=p^.down;
      p^.next:=pt^.next;
      pt^.next:=p;
      p^.proc_r:=r;
      p^.proc_l:=l;
      set_vars;
      end;
    p^.down:=nil;
    end;
  for n:=1 to 200 do
    begin
    processed_data[n]:=0;
    consumed_data[n]:=0;
    end;
end;  (* INITIAL_STATES *)

procedure communicate;
(* PERFORMS THE FUNCTION OF THE COMMUNICATION HARDWARE *)
var
  p,pt:proc_ptr;
begin (* COMMUNICATE *)
pt:=stack;
while (pt<>nil) do
  begin
  p:=pt;
  repeat
  begin
  if (p^.down<>nil) then
    if (p^.down_full and not(p^.down^.data_full)) then
      begin
      p^.down^.data_input:=p^.down_output;
      p^.down_full:=false;
      p^.down^.data_full:=true;
      end;
  if (p^.output_full and not(p^.next^.input_full)) then
    begin
    p^.next^.ring_input:=p^.ring_output;
    p^.next^.input_full:=true;
```

```
        p^.output_full:=false;
        end;
      p:=p^.next;
      end
    until (p=pt);
    pt:=pt^.down;
    end;
end;  (* COMMUNICATE *)

procedure create_new_data;
(* PERFOMS THE FUNCTION OF THE DATA INPUT MECHANISM *)
var
  p:proc_ptr;
begin (* CREATE_NEW_DATA *)
p:=stack;
repeat
  begin
  if (not(p^.data_full) and (data_types[p^.proc_r]<>0)) then
    begin
    if (data_types[p^.proc_r]<0) then
      p^.data_input:=1+trunc(random*(-data_types[p^.proc_r]-0.0001))
    else
      p^.data_input:=data_types[p^.proc_r];
    consumed_data[p^.data_input]:=
      consumed_data[p^.data_input]+1;
    p^.data_full:=true;
    end;
  p:=p^.next;
  end;
until (p=stack);
end; (* CREATE_NEW_DATA *)

procedure compute;
(* PERFORMS THE FUNCTION OF THE PROCESSING ELEMENT *)
var
  p,pt:proc_ptr;

  procedure computing_algorithm;
  (* PERFORMS THE FUNCTION OF THE COMMUNICATING PROCESSES WITHIN *)
  (* THE PROCESSOR *)
  var
    time_unit:integer;

    procedure comms;
    (* PERFORMS THE ACTIVITIES OF THE COMMUNICATION PROCESS *)

    function processor_idle:boolean;
    begin
    processor_idle:=(p^.p_state<=0);
    end;

    function ring_data:boolean;
    begin
    ring_data:=p^.input_full;
    end;

    function new_data:boolean;
    begin
```

```
new_data:=p^.data_full;
end;

function ring_ready:boolean;
begin
ring_ready:=not(p^.output_full);
end;

function down_ready:boolean;
begin
down_ready:=not(p^.down_full);
end;
```

```
procedure take_ring_data;
begin
p^.p_type:=p^.ring_input;
p^.p_state:=p^.p_type;
p^.input_full:=false;
time_unit:=time_unit-1;
end;

procedure take_new_data;
begin
p^.p_type:=p^.data_input;
p^.p_state:=p^.p_type;
p^.data_full:=false;
time_unit:=time_unit-1;
end;
```

```
procedure ring_data_on;
begin
p^.ring_output:=p^.ring_input;
p^.input_full:=false;
p^.output_full:=true;
time_unit:=time_unit-1;
end;

procedure ring_data_down;
begin
p^.down_output:=p^.ring_input;
p^.input_full:=false;
p^.down_full:=true;
time_unit:=time_unit-1;
end;

procedure new_data_on;
begin
p^.ring_output:=p^.data_input;
p^.data_full:=false;
p^.output_full:=true;
time_unit:=time_unit-1;
end;

procedure new_data_down;
begin
p^.down_output:=p^.data_input;
p^.data_full:=false;
p^.down_full:=true;
```

```
  time_unit:=time_unit-1;
  end;

begin (* COMMS2 *)
if processor_idle then
  if ring_data then
    take_ring_data
  else
    if new_data then
      take_new_data
    else
      (* NULL *)
else
  (* NULL *);
if ring_ready then
  if ring_data then
    ring_data_on
  else
    if new_data then
      new_data_on
    else
      (* NULL *)
else
  (* NULL *);
if down_ready then
  if ring_data then
    ring_data_down
  else
    if new_data then
      new_data_down
    else
      (* NULL *)
else
  (* NULL *);
end;  (* COMMS2 *)

procedure process(units:integer);
begin (* PROCESS *)
p^.p_state:=p^.p_state-units;
if (p^.p_state<=0) then
  begin
  if (p^.p_type>0) then
    processed_data[p^.p_type]:=processed_data[p^.p_type]+1;
  p^.p_type:=0;
  p^.p_state:=0;
  end;
  end;  (* PROCESS *)

begin (* COMPUTING_ALGORITHM *)
time_unit:=4;
(* TOTAL PROCESSING EFFORT AVAILABLE *)
(* PER ITERATION *)
if (condition[p^.proc_l,p^.proc_r]=good) then
  begin
  comms;
  (* COMMS USES UP SOME PROCESSING EFFORT *)
  process(time_unit);
  end
```

```
    else
      (* NULL *)
    end;  (* COMPUTING_ALGORITHM *)

begin (* COMPUTE *)
pt:=stack;
while (pt<>nil) do
  begin
  p:=pt;
  repeat
    begin
    computing_algorithm;
    p:=p^.next;
    end
  until (p=pt);
  pt:=pt^.down;
  end;
end;  (* COMPUTE *)

procedure print_state;
(* PRINTS THE STATE OF THE STACK AFTER THE LAST ITERATION *)
var p,pt:proc_ptr;
    l,r:integer;
begin (* PRINT_STATE *)
write(stackrecord,'Iteration:',iterations:3);
writeln(stackrecord);write(stackrecord,'r=   ');
for r:=1 to ring do
  write(stackrecord,' ',r:3);
writeln(stackrecord);
pt:=stack;
write(stackrecord,'Data ');
for r:=1 to ring do
  write(stackrecord,' ',data_types[r]:3);
writeln(stackrecord);
l:=1;
while(pt<>nil) do
  begin
  p:=pt;
  write(stackrecord,'l=',l:3);
  repeat
    begin
    if (condition[p^.proc_l,p^.proc_r]=good) then
      write(stackrecord,' ',p^.p_state:3)
    else
      write(stackrecord,'   F');
    p:=p^.next;
    end;
  until (p=pt);
  pt:=pt^.down;
  l:=l+1;
  writeln(stackrecord);
  end;
writeln(stackrecord);
end;  (* PRINT_STATE *)

procedure print_consumption;
var
  n,totproc,totcons,wtotproc:integer;
```

```
begin (* PRINT_CONSUMPTION *)
totproc:=0;totcons:=0;wtotproc:=0;
writeln(stackrecord);writeln(stackrecord,'Data Consumed and Processed');
writeln(stackrecord,'---- -------- --- ----------');writeln(stackrecord);
for n:=1 to 200 do
  begin
  totcons:=totcons+consumed_data[n];
  totproc:=totproc+processed_data[n];
  wtotproc:=wtotproc+processed_data[n]*n;
  write(stackrecord,'Type:',n:3,' number consumed:',consumed_data[n]:3);
  writeln(stackrecord,' number processed:',processed_data[n]:3);
  end;
write(stackrecord,'Total:  ',' number consumed:',totcons:3);
  writeln(stackrecord,' number processed:',totproc:3);
  writeln(stackrecord,'Total:  ',' Weighted Total Processed:',wtotproc:5);
  end;  (* PRINT_CONSUMPTION *)

begin (* STACKPROG2 *)
a:=1234567;
(* SETS THE SEED FOR THE RANDOM NUMBER GENERATOR *)
input_parameters;
writeln('Parameters input');
initial_states;
writeln('Initial state set up');
writeln('Starting Computation');
for iterations:=1 to largen do
  begin
  communicate;
  create_new_data;
  compute;
  print_state;
  end;
writeln('Finished Computation');
writeln('Printing Results');
print_consumption;
end.  (* STACKPROG2 *)
```

```
program stackprog3(input,output,stackdata,stackrecord);
const
(* CONSTANTS FOR RANDOM NUMBER GENERATION *)
  m=100000000;m1=10000;b=31415821;
type
(* PROCESSING NODE TYPE *)
  proc_ptr=^proc_type;
  proc_type=record
              proc_r:integer;
              proc_l:integer;
              ring_input:integer;
              ring_output:integer;
              input_full:boolean;
              output_full:boolean;
              data_input:integer;
              down_output:integer;
              data_full:boolean;
              down_full:boolean;
              p_type:integer;
              p_state:integer;
              next:proc_ptr;
              down:proc_ptr;
              end;
     state_type=(faulty,good);

var
  a:integer;
  stackdata,stackrecord:text;
  stack:proc_ptr;
  condition:array[1..50,1..50] of state_type;
  data_types:array[1..50] of integer;
  processed_data:array[1..200] of integer;
  consumed_data:array[1..200] of integer;
  largen,iterations,ring,layers:integer;

function mult(p,q:integer):integer;
(* EXTENDED MULTIPLICATION *)
var p1,p0,q1,q0:integer;
begin
p1:=p div m1;  p0:=p mod m1;
q1:=q div m1;  q0:=q mod m1;
mult:=(((p0*q1+p1*q0) mod m1)*m1+p0*q0) mod m;
end;

function random:real;
(* RANDOM NUMBER GENERATOR *)
(* PG37 - ALGORITHMS - SEDGEWICK *)
(* ADDISON - WESLEY 1984 *)
begin
a:=(mult(a,b)+1)mod m;
random:=a/m;
end;

procedure input_parameters;
(* INITIALISE INPUT AND OUTPUT FILES *)
(* READ IN THE PARAMETERS OF THE RING TO MODEL *)
(* WRITES A HEADING TO STACKRECORD *)
var
```

```
  l,r,n:integer;
begin (* INPUT_PARAMETERS *)
reset(stackdata);rewrite(stackrecord);
writeln(stackrecord,'stackprog3');
writeln(stackrecord,'Stack State record for simulation using :-');
(* READ IN THE NUMBER OF PROCESSORS PER RING *)
readln(stackdata,ring);
writeln(stackrecord,'Number of processors per layer = ',ring:3);
(* READ IN THE NUMBER OF LAYERS *)
readln(stackdata,layers);
writeln(stackrecord,'Number of layers = ',layers:3);
(* READ IN THE NUMBER OF ITERATIONS TO PERFORM *)
readln(stackdata,largen);
writeln(stackrecord,'Number of iterations performed = ',largen:3);
(* READ IN THE FAULTY PROCESSORS *)
writeln(stackrecord,'With faulty processors:-');
for l:=1 to 50 do
  for r:= 1 to 50 do
    condition[l,r]:=good;
while not(stackdata^='D') do
  begin
  readln(stackdata,l,r);
  writeln(stackrecord,'layer:',l:3,' ring:',r:3);
  condition[l,r]:=faulty;
  end;
readln(stackdata);
(* READ IN THE DATA_TYPE (PROCESSING TIME) FOR EACH PROCESSOR *)
writeln(stackrecord,'Data types fed to processors');writeln;
for n:=1 to ring do
  begin
  readln(stackdata,data_types[n]);
  writeln(stackrecord,'Processor :',n:3,' Data type :',data_types[n]:3);
  end;
writeln(stackrecord);
end;  (* INPUT_PARAMETERS *)

procedure initial_states;
(* SETS UP THE INITIAL STATE OF THE STACK DATA STRUCTURE *)
(* AND PRINTS A HEADING TO FILE STACKRECORD *)
var
  p,pt:proc_ptr;
  r,l,n:integer;

  procedure set_vars;
  begin (* SET_VARS *)
  p^.input_full:=false;
  p^.output_full:=false;
  p^.data_full:=false;
  p^.down_full:=false;
  p^.p_type:=0;
  p^.p_state:=0;
  end;  (* SET_VARS *)

begin (* INITIAL_STATES *)
stack:=nil;
for l:=layers downto 1 do
  begin
  new(p);
```

```
    p^.down:=stack;
    stack:=p;
    p^.proc_r:=1;
    p^.proc_l:=1;
    p^.next:=p;
    set_vars;
    end;
  for r:=ring downto 2 do
    begin
    pt:=stack;
    new(p);
    p^.next:=pt^.next;
    pt^.next:=p;
    p^.proc_r:=r;
    p^.proc_l:=1;
    set_vars;
    for l:=2 to layers do
      begin
      pt:=pt^.down;
      new(p^.down);
      p:=p^.down;
      p^.next:=pt^.next;
      pt^.next:=p;
      p^.proc_r:=r;
      p^.proc_l:=l;
      set_vars;
      end;
    p^.down:=nil;
    end;
  for n:=1 to 200 do
    begin
    processed_data[n]:=0;
    consumed_data[n]:=0;
    end;
  end;  (* INITIAL_STATES *)

  procedure communicate;
  (* PERFORMS THE FUNCTION OF THE COMMUNICATION HARDWARE *)
  var
    p,pt:proc_ptr;
  begin (* COMMUNICATE *)
  pt:=stack;
  while (pt<>nil) do
    begin
    p:=pt;
    repeat
    begin
    if (p^.down<>nil) then
      if (p^.down_full and not(p^.down^.data_full)) then
        begin
        p^.down^.data_input:=p^.down_output;
        p^.down_full:=false;
        p^.down^.data_full:=true;
        end;
    if (p^.output_full and not(p^.next^.input_full)) then
      begin
      p^.next^.ring_input:=p^.ring_output;
      p^.next^.input_full:=true;
```

```
      p^.output_full:=false;
      end;
    p:=p^.next;
    end
  until (p=pt);
  pt:=pt^.down;
  end;
end;  (* COMMUNICATE *)

procedure create_new_data;
(* PERFOMS THE FUNCTION OF THE DATA INPUT MECHANISM *)
var
  p:proc_ptr;
begin (* CREATE_NEW_DATA *)
p:=stack;
repeat
  begin
  if (not(p^.data_full) and (data_types[p^.proc_r]<>0)) then
    begin
    if (data_types[p^.proc_r]<0) then
      p^.data_input:=1+trunc(random*(-data_types[p^.proc_r]-0.0001))
    else
      p^.data_input:=data_types[p^.proc_r];
    consumed_data[p^.data_input]:=
      consumed_data[p^.data_input]+1;
    p^.data_full:=true;
    end;
  p:=p^.next;
  end;
until (p=stack);
end; (* CREATE_NEW_DATA *)

procedure compute;
(* PERFORMS THE FUNCTION OF THE PROCESSING ELEMENT *)
var
  p,pt:proc_ptr;

  procedure computing_algorithm;
  (* PERFORMS THE FUNCTION OF THE COMMUNICATING PROCESSES WITHIN *)
  (* THE PROCESSOR *)
  var
    time_unit:integer;

    procedure comms;
    (* PERFORMS THE ACTIVITIES OF THE COMMUNICATION PROCESS *)

    function processor_idle:boolean;
    begin
    processor_idle:=(p^.p_state<=0);
    end;

    function ring_data:boolean;
    begin
    ring_data:=p^.input_full;
    end;

    function new_data:boolean;
    begin
```

```
new_data:=p^.data_full;
end;

function ring_ready:boolean;
begin
ring_ready:=not(p^.output_full);
end;

function down_ready:boolean;
begin
down_ready:=not(p^.down_full);
end;

procedure take_ring_data;
begin
p^.p_type:=p^.ring_input;
p^.p_state:=p^.p_type;
p^.input_full:=false;
time_unit:=time_unit-1;
end;

procedure take_new_data;
begin
p^.p_type:=p^.data_input;
p^.p_state:=p^.p_type;
p^.data_full:=false;
time_unit:=time_unit-1;
end;

procedure ring_data_on;
begin
p^.ring_output:=p^.ring_input;
p^.input_full:=false;
p^.output_full:=true;
time_unit:=time_unit-1;
end;

procedure ring_data_down;
begin
p^.down_output:=p^.ring_input;
p^.input_full:=false;
p^.down_full:=true;
time_unit:=time_unit-1;
end;

procedure new_data_on;
begin
p^.ring_output:=p^.data_input;
p^.data_full:=false;
p^.output_full:=true;
time_unit:=time_unit-1;
end;

procedure new_data_down;
begin
p^.down_output:=p^.data_input;
p^.data_full:=false;
p^.down_full:=true;
```

```
  time_unit:=time_unit-1;
  end;

begin (* COMMS3 *)
if processor_idle then
  if new_data then
    take_new_data
  else
    if ring_data then
      take_ring_data
    else
      (* NULL *)
else
  (* NULL *);
if down_ready then
  if new_data then
    new_data_down
  else
    if ring_data then
      ring_data_down
    else
      (* NULL *)
else
  (* NULL *);
if ring_ready then
  if new_data then
    new_data_on
  else
    if ring_data then
      ring_data_on
    else
      (* NULL *)
else
  (* NULL *);
end;  (* COMMS3 *)

procedure process(units:integer);
begin (* PROCESS *)
p^.p_state:=p^.p_state-units;
if (p^.p_state<=0) then
  begin
  if (p^.p_type>0) then
    processed_data[p^.p_type]:=processed_data[p^.p_type]+1;
  p^.p_type:=0;
  p^.p_state:=0;
  end;
  end;  (* PROCESS *)

begin (* COMPUTING_ALGORITHM *)
time_unit:=4;
(* TOTAL PROCESSING EFFORT AVAILABLE *)
(* PER ITERATION *)
if (condition[p^.proc_l,p^.proc_r]=good) then
  begin
  comms;
  (* COMMS USES UP SOME PROCESSING EFFORT *)
  process(time_unit);
  end
```

```
    else
      (* NULL *)
    end;  (* COMPUTING_ALGORITHM *)

begin (* COMPUTE *)
pt:=stack;
while (pt<>nil) do
    begin
    p:=pt;
    repeat
      begin
      computing_algorithm;
      p:=p^.next;
      end
    until (p=pt);
    pt:=pt^.down;
    end;
end;  (* COMPUTE *)

procedure print_state;
(* PRINTS THE STATE OF THE STACK AFTER THE LAST ITERATION *)
var p,pt:proc_ptr;
    l,r:integer;
begin (* PRINT_STATE *)
    write(stackrecord,'Iteration:',iterations:3);
    writeln(stackrecord);write(stackrecord,'r=   ');
    for r:=1 to ring do
      write(stackrecord,' ',r:3);
    writeln(stackrecord);
    pt:=stack;
    write(stackrecord,'Data ');
    for r:=1 to ring do
      write(stackrecord,' ',data_types[r]:3);
    writeln(stackrecord);
    l:=1;
    while(pt<>nil) do
      begin
      p:=pt;
      write(stackrecord,'l=',l:3);
      repeat
        begin
        if (condition[p^.proc_l,p^.proc_r]=good) then
          write(stackrecord,' ',p^.p_state:3)
        else
          write(stackrecord,'   F');
        p:=p^.next;
        end;
      until (p=pt);
      pt:=pt^.down;
      l:=l+1;
      writeln(stackrecord);
      end;
    writeln(stackrecord);
    end;  (* PRINT_STATE *)

procedure print_consumption;
var
  n,totproc,totcons,wtotproc:integer;
```

```
begin (* PRINT_CONSUMPTION *)
totproc:=0;totcons:=0;wtotproc:=0;
writeln(stackrecord);writeln(stackrecord,'Data Consumed and Processed');
writeln(stackrecord,'---- -------- --- ----------');writeln(stackrecord);
for n:=1 to 200 do
  begin
  totcons:=totcons+consumed_data[n];
  totproc:=totproc+processed_data[n];
  wtotproc:=wtotproc+processed_data[n]*n;
  write(stackrecord,'Type:',n:3,' number consumed:',consumed_data[n]:3);
  writeln(stackrecord,' number processed:',processed_data[n]:3);
  end;
write(stackrecord,'Total:  ',' number consumed:',totcons:3);
  writeln(stackrecord,' number processed:',totproc:3);
  writeln(stackrecord,'Total:  ',' Weighted Total Processed:',wtotproc:5);
  end;  (* PRINT_CONSUMPTION *)

begin (* STACKPROG3 *)
a:=1234567;
(* SETS THE SEED FOR THE RANDOM NUMBER GENERATOR *)
input_parameters;
writeln('Parameters input');
initial_states;
writeln('Initial state set up');
writeln('Starting Computation');
for iterations:=1 to largen do
  begin
  communicate;
  create_new_data;
  compute;
  print_state;
  end;
writeln('Finished Computation');
writeln('Printing Results');
print_consumption;
end.  (* STACKPROG3 *)
```

```
program stackprog4(input,output,stackdata,stackrecord);
const
(* CONSTANTS FOR RANDOM NUMBER GENERATION *)
  m=100000000;m1=10000;b=31415821;
type
(* PROCESSING NODE TYPE *)
  proc_ptr=^proc_type;
  proc_type=record
              proc_r:integer;
              proc_l:integer;
              ring_input:integer;
              ring_output:integer;
              input_full:boolean;
              output_full:boolean;
              data_input:integer;
              down_output:integer;
              data_full:boolean;
              down_full:boolean;
              p_type:integer;
              p_state:integer;
              next:proc_ptr;
              down:proc_ptr;
              end;
    state_type=(faulty,good);

var
  a:integer;
  stackdata,stackrecord:text;
  stack:proc_ptr;
  condition:array[1..50,1..50] of state_type;
data_types:array[1..50] of integer;
processed_data:array[1..200] of integer;
consumed_data:array[1..200] of integer;
largen,iterations,ring,layers:integer;

function mult(p,q:integer):integer;
(* EXTENDED MULITPLICATION *)
var p1,p0,q1,q0:integer;
begin
p1:=p div m1;  p0:=p mod m1;
q1:=q div m1;  q0:=q mod m1;
mult:=(((p0*q1+p1*q0) mod m1)*m1+p0*q0) mod m;
end;

function random:real;
(* RANDOM NUMBER GENERATOR *)
  (* PG37 - ALGORITHMS - SEDGEWICK *)
  (* ADDISON - WESLEY 1984 *)
  begin
  a:=(mult(a,b)+1)mod m;
  random:=a/m;
  end;

  procedure input_parameters;
  (* INITIALISE INPUT AND OUTPUT FILES *)
  (* READ IN THE PARAMETERS OF THE RING TO MODEL *)
  (* WRITES A HEADING TO STACKRECORD *)
  var
```

```
  l,r,n:integer;
begin (* INPUT_PARAMETERS *)
reset(stackdata);rewrite(stackrecord);
writeln(stackrecord,'stackprog4');
writeln(stackrecord,'Stack State record for simulation using :-');
(* READ IN THE NUMBER OF PROCESSORS PER RING *)
readln(stackdata,ring);
writeln(stackrecord,'Number of processors per layer = ',ring:3);
(* READ IN THE NUMBER OF LAYERS *)
readln(stackdata,layers);
writeln(stackrecord,'Number of layers = ',layers:3);
(* READ IN THE NUMBER OF ITERATIONS TO PERFORM *)
readln(stackdata,largen);
writeln(stackrecord,'Number of iterations performed = ',largen:3);
(* READ IN THE FAULTY PROCESSORS *)
writeln(stackrecord,'With faulty processors:-');
for l:=1 to 50 do
  for r:= 1 to 50 do
    condition[l,r]:=good;
while not(stackdata^='D') do
  begin
  readln(stackdata,l,r);
  writeln(stackrecord,'layer:',l:3,' ring:',r:3);
  condition[l,r]:=faulty;
  end;
readln(stackdata);
(* READ IN THE DATA_TYPE (PROCESSING TIME) FOR EACH PROCESSOR *)
writeln(stackrecord,'Data types fed to processors');writeln;
for n:=1 to ring do
  begin
  readln(stackdata,data_types[n]);
  writeln(stackrecord,'Processor :',n:3,' Data type :',data_types[n]:3);
  end;
writeln(stackrecord);
end;  (* INPUT_PARAMETERS *)

procedure initial_states;
(* SETS UP THE INITIAL STATE OF THE STACK DATA STRUCTURE *)
(* AND PRINTS A HEADING TO FILE STACKRECORD *)
var
  p,pt:proc_ptr;
  r,l,n:integer;

  procedure set_vars;
  begin (* SET_VARS *)
  p^.input_full:=false;
  p^.output_full:=false;
  p^.data_full:=false;
  p^.down_full:=false;
  p^.p_type:=0;
  p^.p_state:=0;
  end;  (* SET_VARS *)

begin (* INITIAL_STATES *)
stack:=nil;
for l:=layers downto 1 do
  begin
  new(p);
```

```
    p^.down:=stack;
    stack:=p;
    p^.proc_r:=1;
    p^.proc_l:=1;
    p^.next:=p;
    set_vars;
    end;
  for r:=ring downto 2 do
    begin
    pt:=stack;
    new(p);
    p^.next:=pt^.next;
    pt^.next:=p;
    p^.proc_r:=r;
    p^.proc_l:=1;
    set_vars;
    for l:=2 to layers do
      begin
      pt:=pt^.down;
      new(p^.down);
      p:=p^.down;
      p^.next:=pt^.next;
      pt^.next:=p;
      p^.proc_r:=r;
      p^.proc_l:=l;
      set_vars;
      end;
    p^.down:=nil;
    end;
  for n:=1 to 200 do
    begin
    processed_data[n]:=0;
    consumed_data[n]:=0;
    end;
end;  (* INITIAL_STATES *)

procedure communicate;
(* PERFORMS THE FUNCTION OF THE COMMUNICATION HARDWARE *)
var
  p,pt:proc_ptr;
begin (* COMMUNICATE *)
pt:=stack;
while (pt<>nil) do
  begin
  p:=pt;
  repeat
  begin
  if (p^.down<>nil) then
    if (p^.down_full and not(p^.down^.data_full)) then
      begin
      p^.down^.data_input:=p^.down_output;
      p^.down_full:=false;
      p^.down^.data_full:=true;
      end;
  if (p^.output_full and not(p^.next^.input_full)) then
    begin
    p^.next^.ring_input:=p^.ring_output;
    p^.next^.input_full:=true;
```

```
      p^.output_full:=false;
      end;
    p:=p^.next;
    end
  until (p=pt);
  pt:=pt^.down;
  end;
end;  (* COMMUNICATE *)

procedure create_new_data;
(* PERFOMS THE FUNCTION OF THE DATA INPUT MECHANISM *)
var
  p:proc_ptr;
begin (* CREATE_NEW_DATA *)
p:=stack;
repeat
  begin
  if (not(p^.data_full) and (data_types[p^.proc_r]<>0)) then
    begin
    if (data_types[p^.proc_r]<0) then
      p^.data_input:=1+trunc(random*(-data_types[p^.proc_r]-0.0001))
    else
      p^.data_input:=data_types[p^.proc_r];
    consumed_data[p^.data_input]:=
      consumed_data[p^.data_input]+1;
    p^.data_full:=true;
    end;
  p:=p^.next;
  end;
until (p=stack);
end; (* CREATE_NEW_DATA *)

procedure compute;
(* PERFORMS THE FUNCTION OF THE PROCESSING ELEMENT *)
var
  p,pt:proc_ptr;

  procedure computing_algorithm;
  (* PERFORMS THE FUNCTION OF THE COMMUNICATING PROCESSES WITHIN *)
  (* THE PROCESSOR *)
  var
    time_unit:integer;

    procedure comms;
    (* PERFORMS THE ACTIVITIES OF THE COMMUNICATION PROCESS *)

    function processor_idle:boolean;
    begin
    processor_idle:=(p^.p_state<=0);
    end;

    function ring_data:boolean;
    begin
    ring_data:=p^.input_full;
    end;

    function new_data:boolean;
    begin
```

```
new_data:=p^.data_full;
end;

function ring_ready:boolean;
begin
ring_ready:=not(p^.output_full);
end;

function down_ready:boolean;
begin
down_ready:=not(p^.down_full);
end;

procedure take_ring_data;
begin
p^.p_type:=p^.ring_input;
p^.p_state:=p^.p_type;
p^.input_full:=false;
time_unit:=time_unit-1;
end;

procedure take_new_data;
begin
p^.p_type:=p^.data_input;
p^.p_state:=p^.p_type;
p^.data_full:=false;
time_unit:=time_unit-1;
end;

procedure ring_data_on;
begin
p^.ring_output:=p^.ring_input;
p^.input_full:=false;
p^.output_full:=true;
time_unit:=time_unit-1;
end;

procedure ring_data_down;
begin
p^.down_output:=p^.ring_input;
p^.input_full:=false;
p^.down_full:=true;
time_unit:=time_unit-1;
end;

procedure new_data_on;
begin
p^.ring_output:=p^.data_input;
p^.data_full:=false;
p^.output_full:=true;
time_unit:=time_unit-1;
end;

procedure new_data_down;
begin
p^.down_output:=p^.data_input;
p^.data_full:=false;
p^.down_full:=true;
```

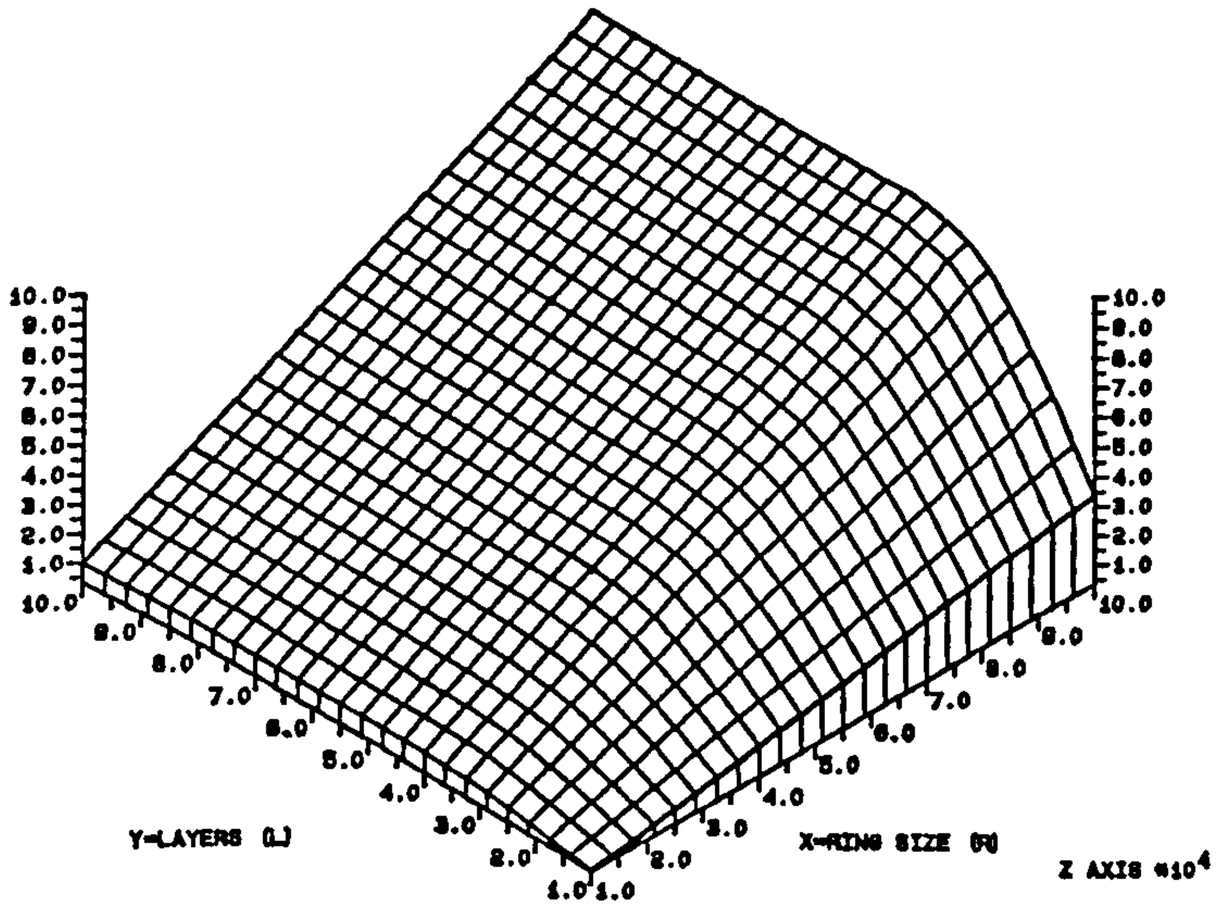
```
  time_unit:=time_unit-1;
  end;

begin (* COMMS4 *)
if processor_idle then
  if ring_data then
    take_ring_data
  else
    if new_data then
      take_new_data
    else
      (* NULL *)
else
  (* NULL *);
if down_ready then
  if ring_data then
    ring_data_down
  else
    if new_data then
      new_data_down
    else
      (* NULL *)
else
  (* NULL *);
if ring_ready then
  if ring_data then
    ring_data_on
  else
    if new_data then
      new_data_on
    else
      (* NULL *)
else
  (* NULL *);
end;  (* COMMS4 *)

procedure process(units:integer);
begin (* PROCESS *)
p^.p_state:=p^.p_state-units;
if (p^.p_state<=0) then
  begin
  if (p^.p_type>0) then
    processed_data[p^.p_type]:=processed_data[p^.p_type]+1;
  p^.p_type:=0;
  p^.p_state:=0;
  end;
  end;  (* PROCESS *)

begin (* COMPUTING_ALGORITHM *)
time_unit:=4;
(* TOTAL PROCESSING EFFORT AVAILABLE *)
(* PER ITERATION *)
if (condition[p^.proc_l,p^.proc_r]=good) then
  begin
  comms;
  (* COMMS USES UP SOME PROCESSING EFFORT *)
  process(time_unit);
  end
```

```
    else
      (* NULL *)
    end;  (* COMPUTING_ALGORITHM *)

begin (* COMPUTE *)
pt:=stack;
while (pt<>nil) do
  begin
  p:=pt;
  repeat
    begin
    computing_algorithm;
    p:=p^.next;
    end
  until (p=pt);
  pt:=pt^.down;
  end;
end;  (* COMPUTE *)

procedure print_state;
(* PRINTS THE STATE OF THE STACK AFTER THE LAST ITERATION *)
var p,pt:proc_ptr;
    l,r:integer;
begin (* PRINT_STATE *)
write(stackrecord,'Iteration:',iterations:3);
writeln(stackrecord);write(stackrecord,'r=   ');
for r:=1 to ring do
  write(stackrecord,' ',r:3);
writeln(stackrecord);
pt:=stack;
write(stackrecord,'Data ');
for r:=1 to ring do
  write(stackrecord,' ',data_types[r]:3);
writeln(stackrecord);
l:=1;
while(pt<>nil) do
  begin
  p:=pt;
  write(stackrecord,'l=',l:3);
  repeat
    begin
    if (condition[p^.proc_l,p^.proc_r]=good) then
      write(stackrecord,' ',p^.p_state:3)
    else
      write(stackrecord,'   F');
    p:=p^.next;
    end;
  until (p=pt);
  pt:=pt^.down;
  l:=l+1;
  writeln(stackrecord);
  end;
writeln(stackrecord);
end;  (* PRINT_STATE *)

procedure print_consumption;
var
  n,totproc,totcons,wtotproc:integer;
```

```
begin (* PRINT_CONSUMPTION *)
totproc:=0;totcons:=0;wtotproc:=0;
writeln(stackrecord);writeln(stackrecord,'Data Consumed and Processed');
writeln(stackrecord,'---- -------- --- ----------');writeln(stackrecord);
for n:=1 to 200 do
  begin
  totcons:=totcons+consumed_data[n];
  totproc:=totproc+processed_data[n];
  wtotproc:=wtotproc+processed_data[n]*n;
  write(stackrecord,'Type:',n:3,' number consumed:',consumed_data[n]:3);
  writeln(stackrecord,' number processed:',processed_data[n]:3);
  end;
write(stackrecord,'Total:  ',' number consumed:',totcons:3);
  writeln(stackrecord,' number processed:',totproc:3);
  writeln(stackrecord,'Total:  ',' Weighted Total Processed:',wtotproc:5);
  end;  (* PRINT_CONSUMPTION *)

begin (* STACKPROG4 *)
a:=1234567;
(* SETS THE SEED FOR THE RANDOM NUMBER GENERATOR *)
input_parameters;
writeln('Parameters input');
initial_states;
writeln('Initial state set up');
writeln('Starting Computation');
for iterations:=1 to largen do
  begin
  communicate;
  create_new_data;
  compute;
  print_state;
  end;
writeln('Finished Computation');
writeln('Printing Results');
print_consumption;
end.  (* STACKPROG4 *)
```

APPENDIX 5

5 Homogeneous Cylinder Simulation Results



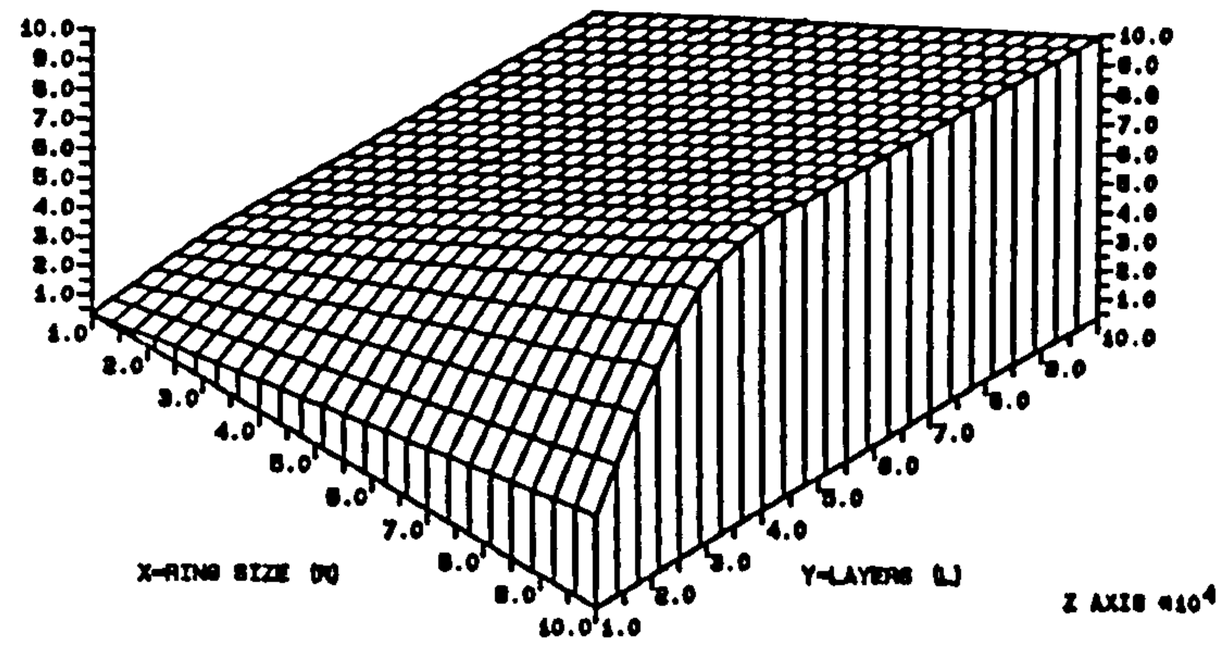
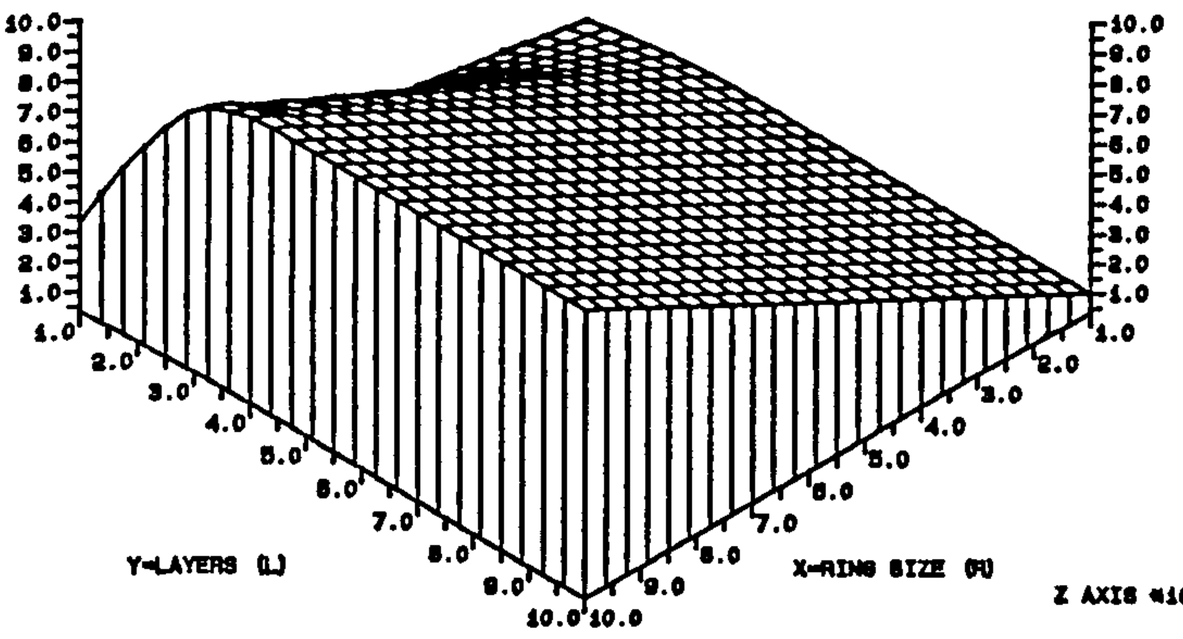
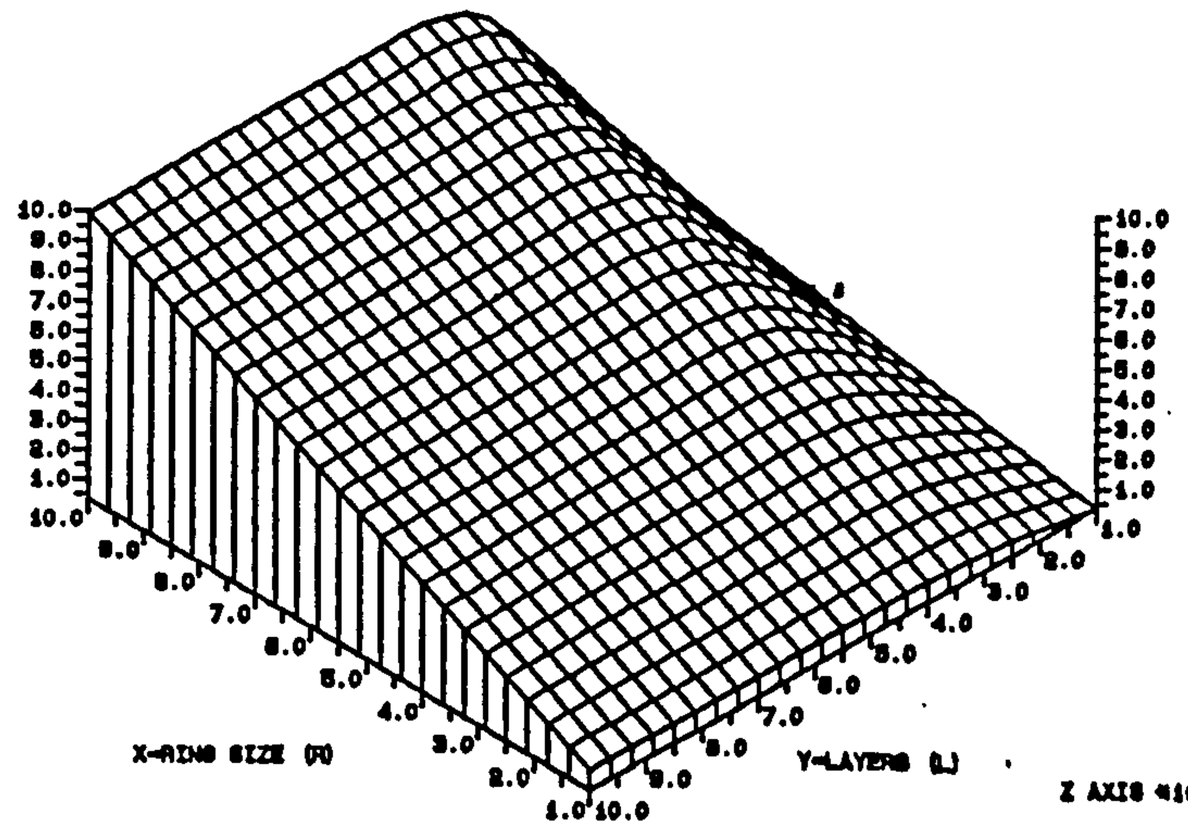


FIG. 5A1.1 HOMOGENEOUS COMMS1 FED AT ALL POINTS WITH DATA OF TYPE 10. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





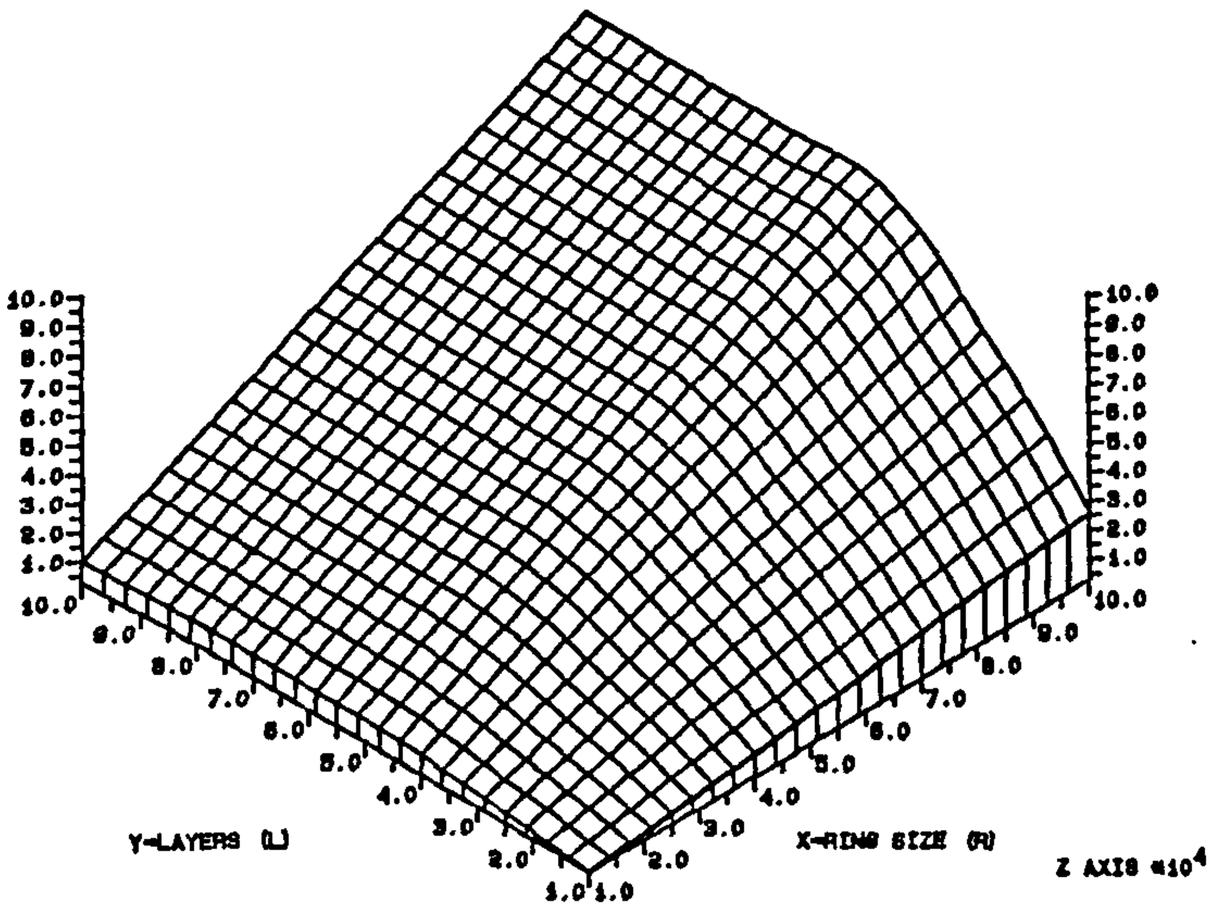
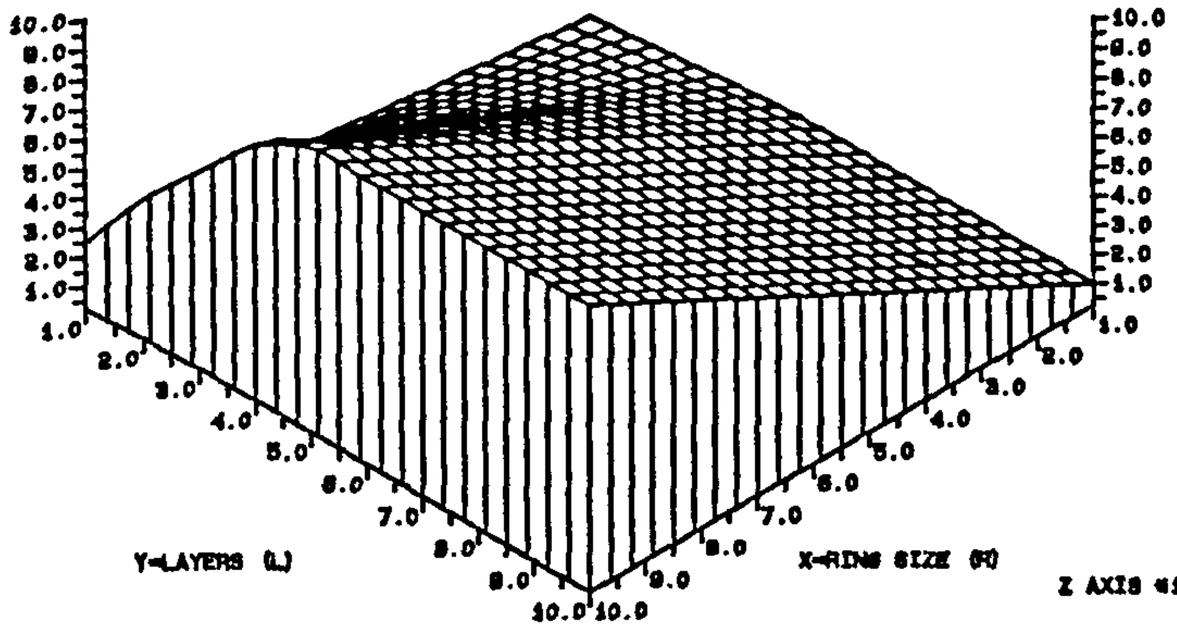
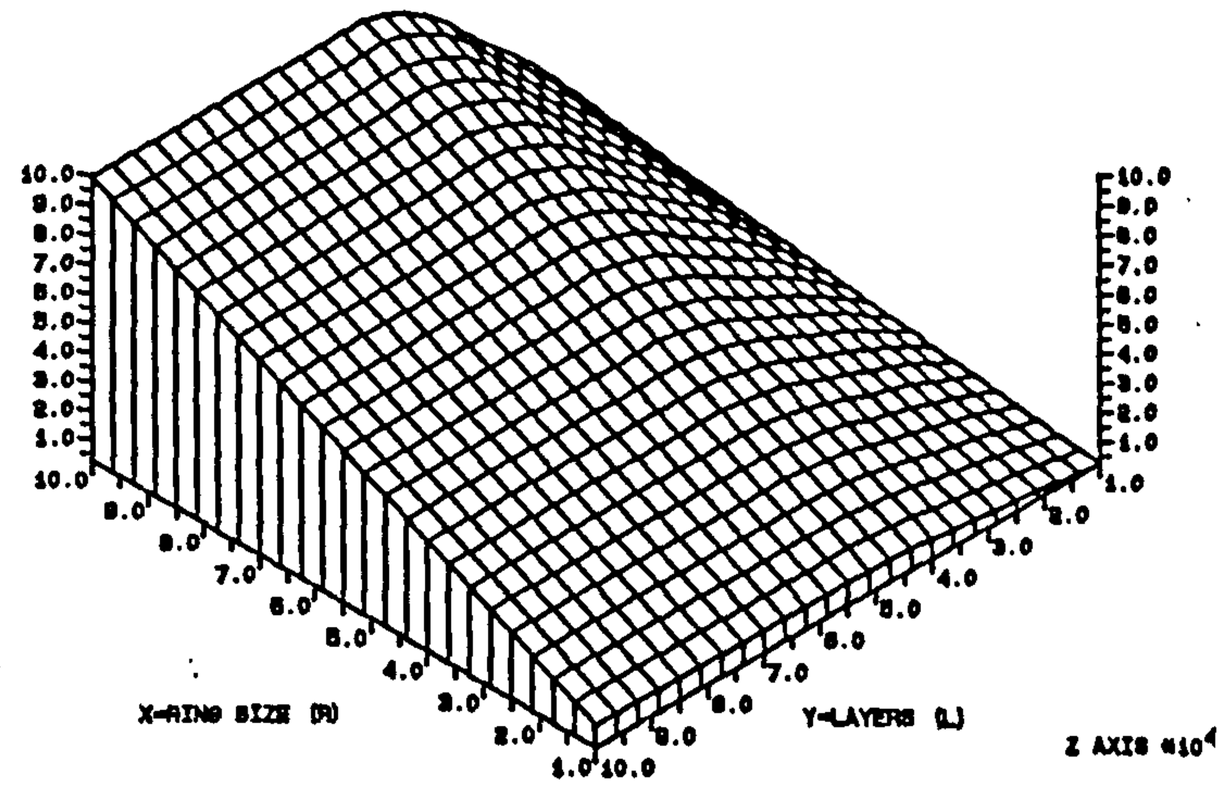
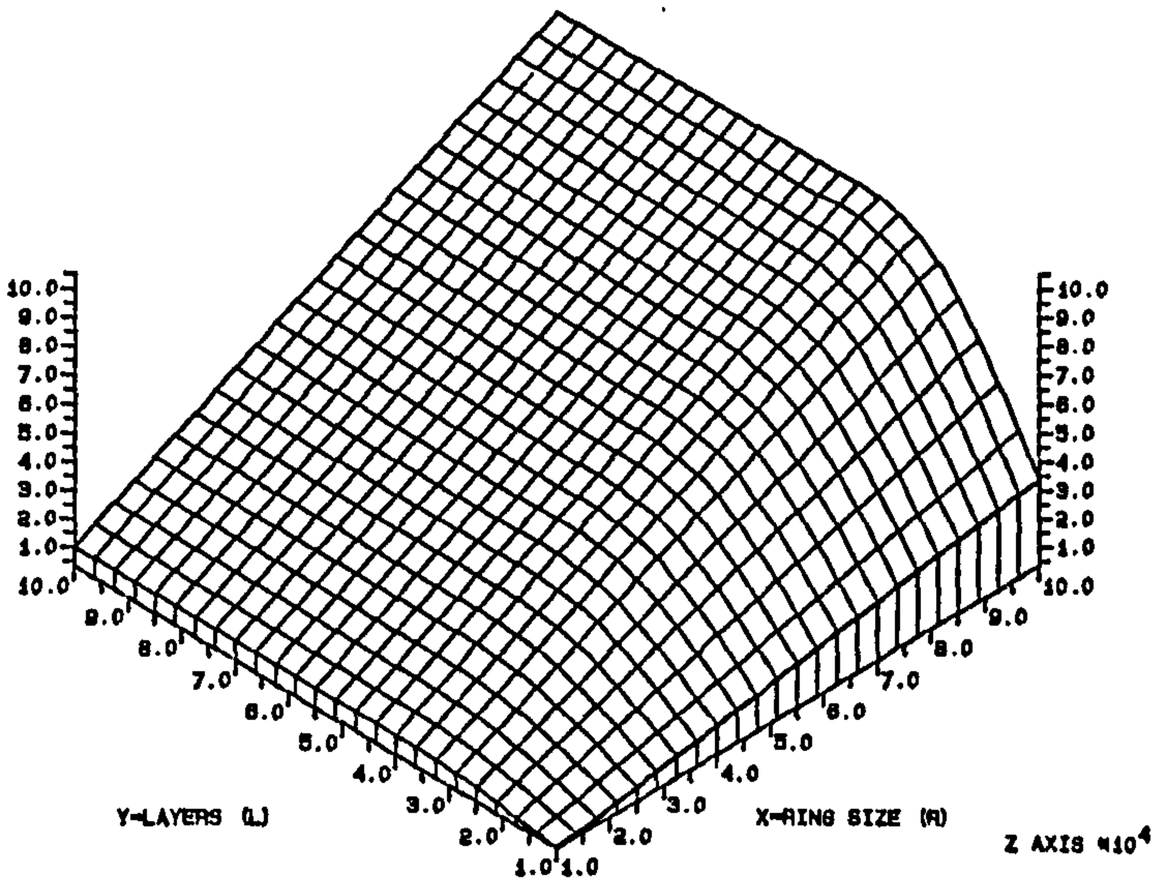


FIG. 5A1.2 HOMOGENEOUS COMMS2 FED AT ALL POINTS WITH DATA OF TYPE 10. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.







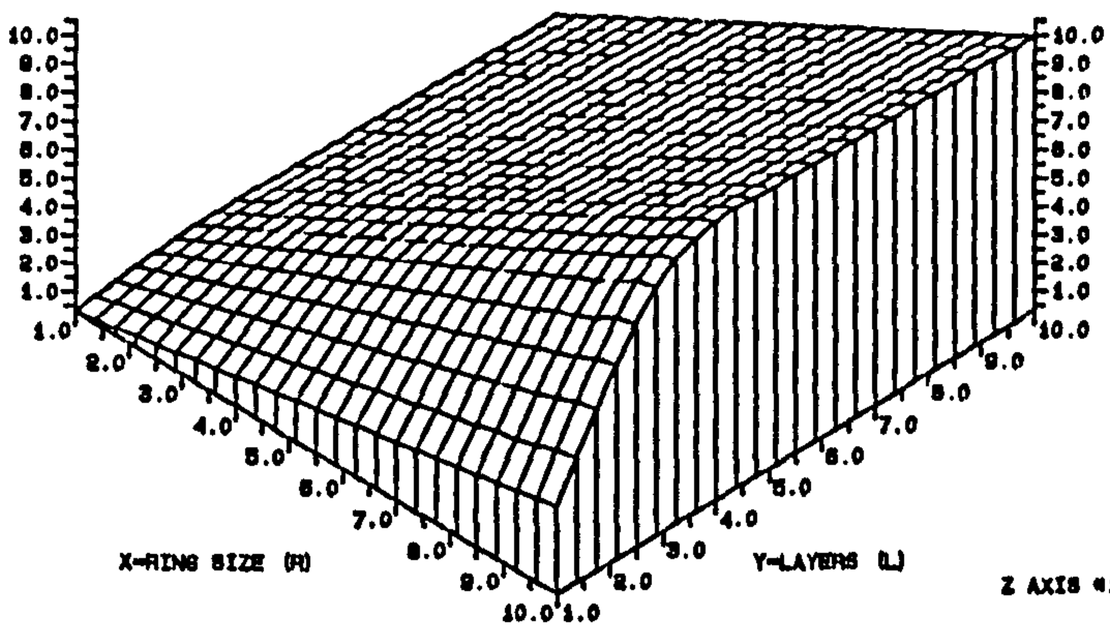
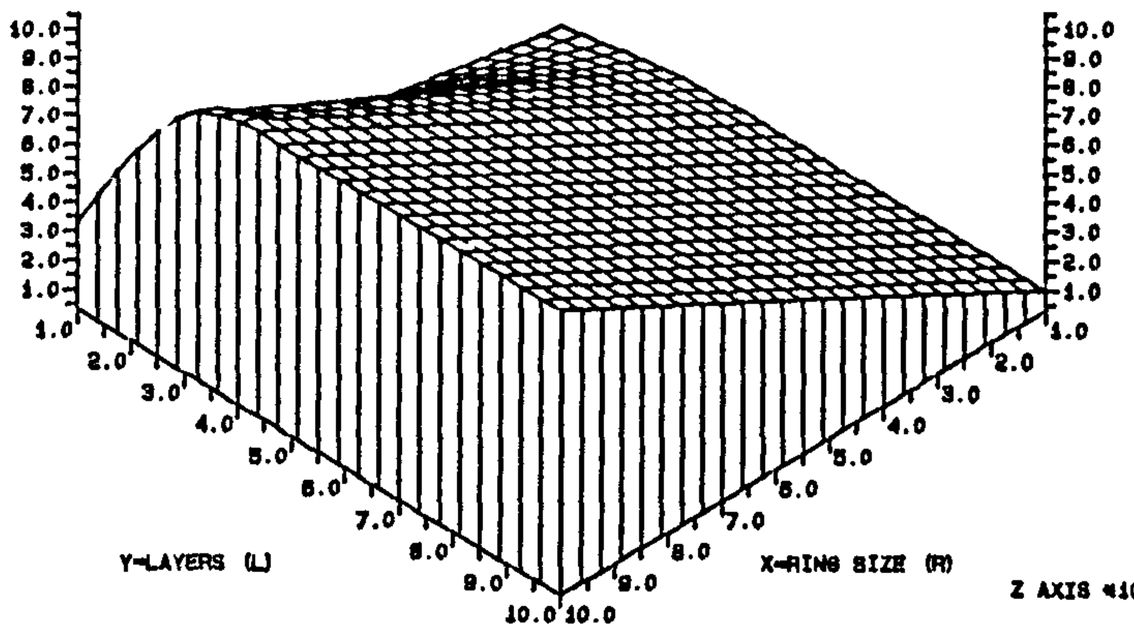
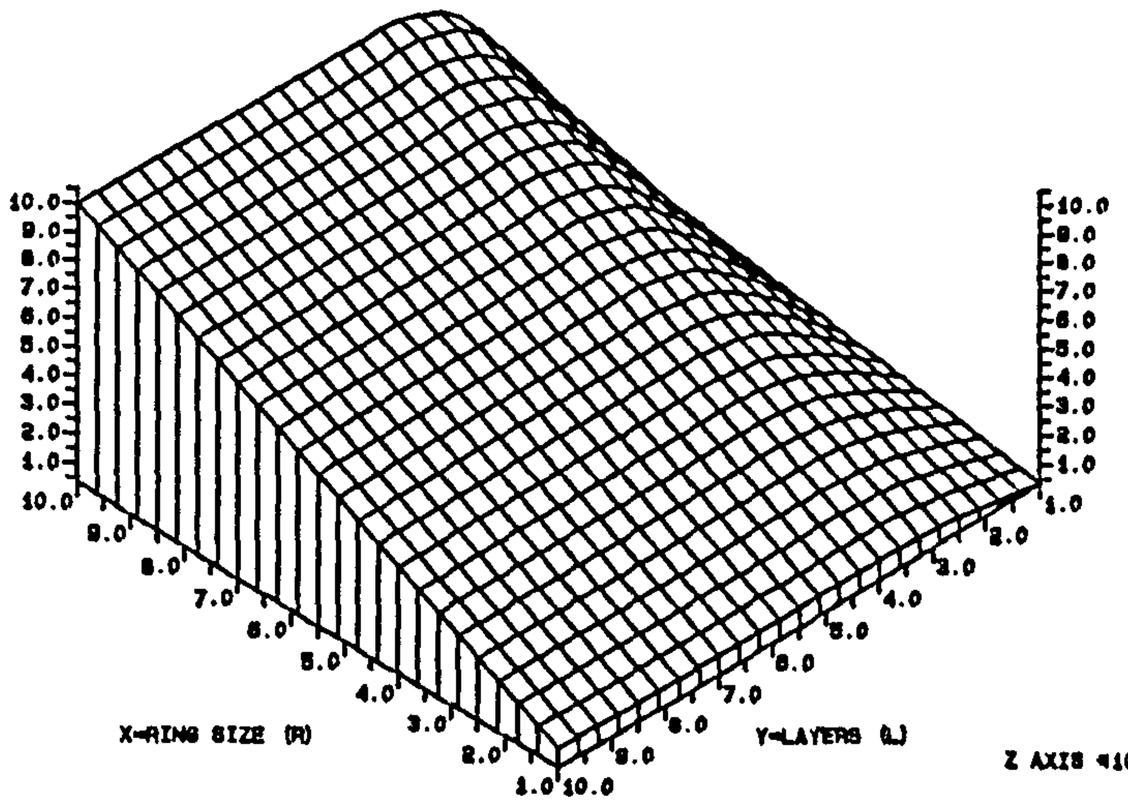
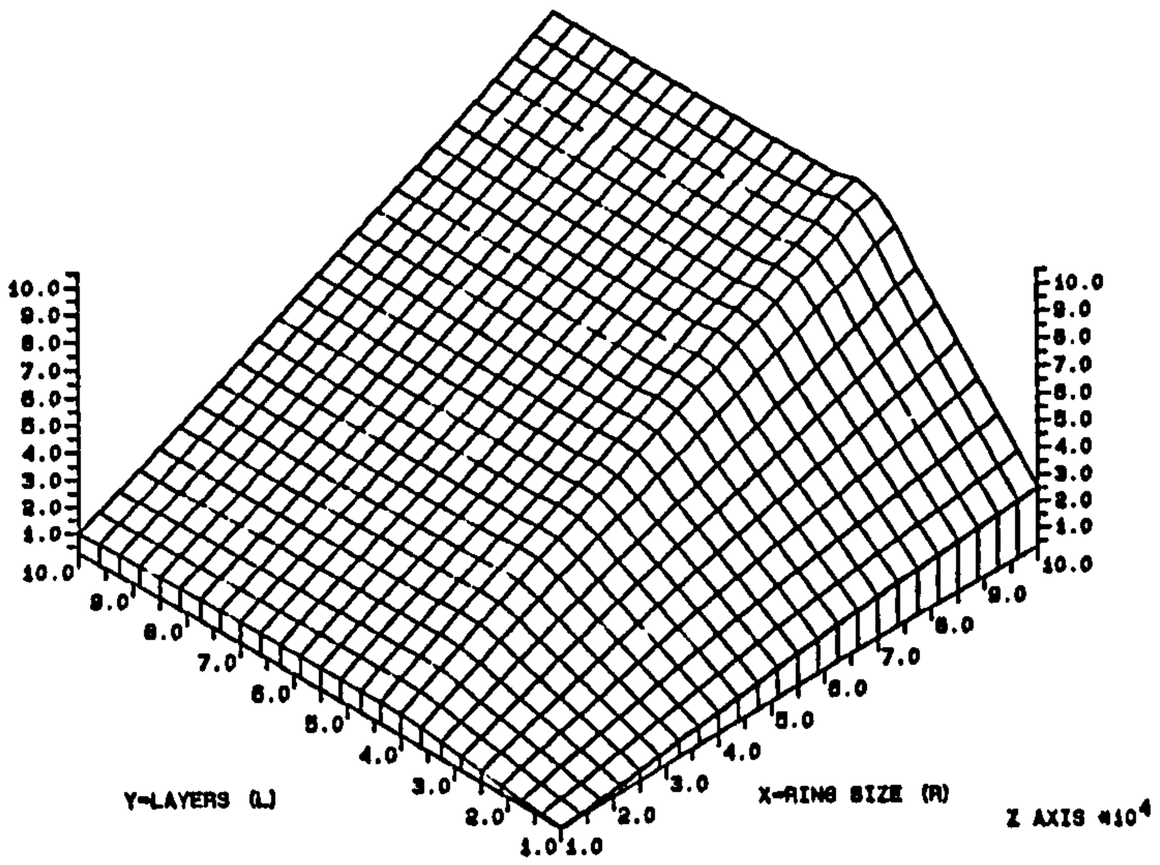


FIG. 5A1.3 HOMOGENEOUS COMMS3 FED AT ALL POINTS WITH DATA OF TYPE 10. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.







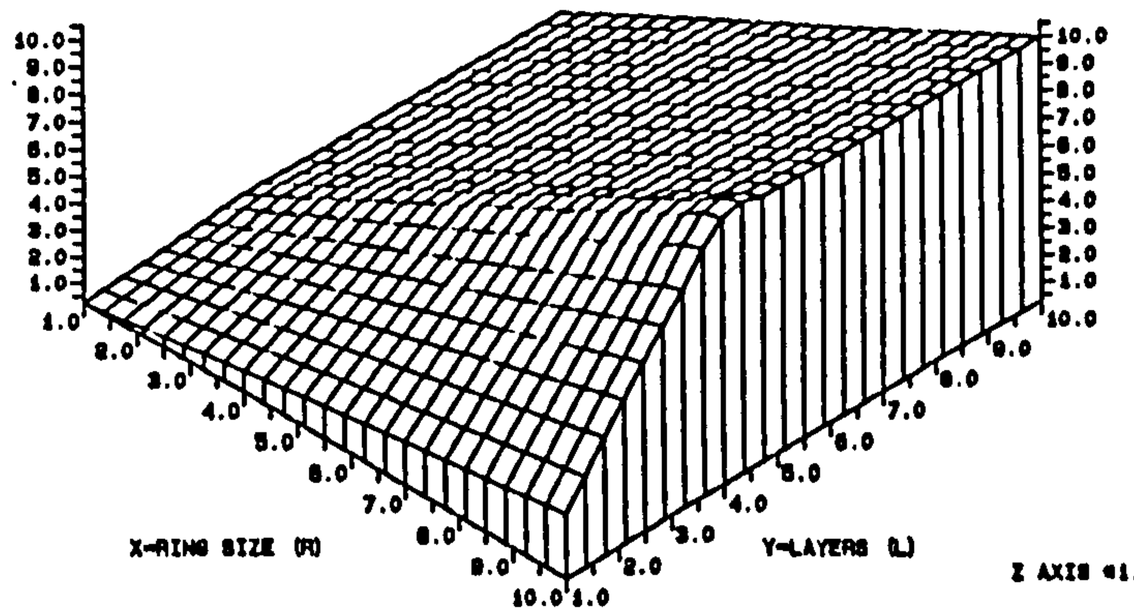
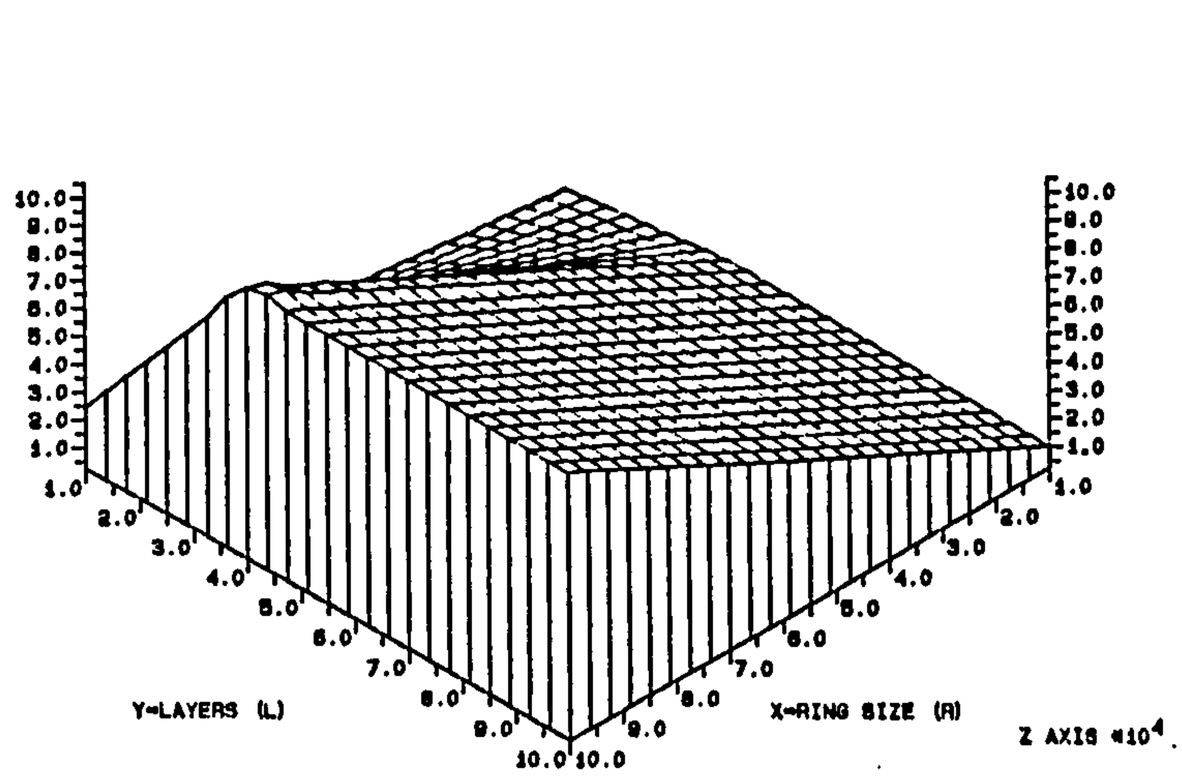
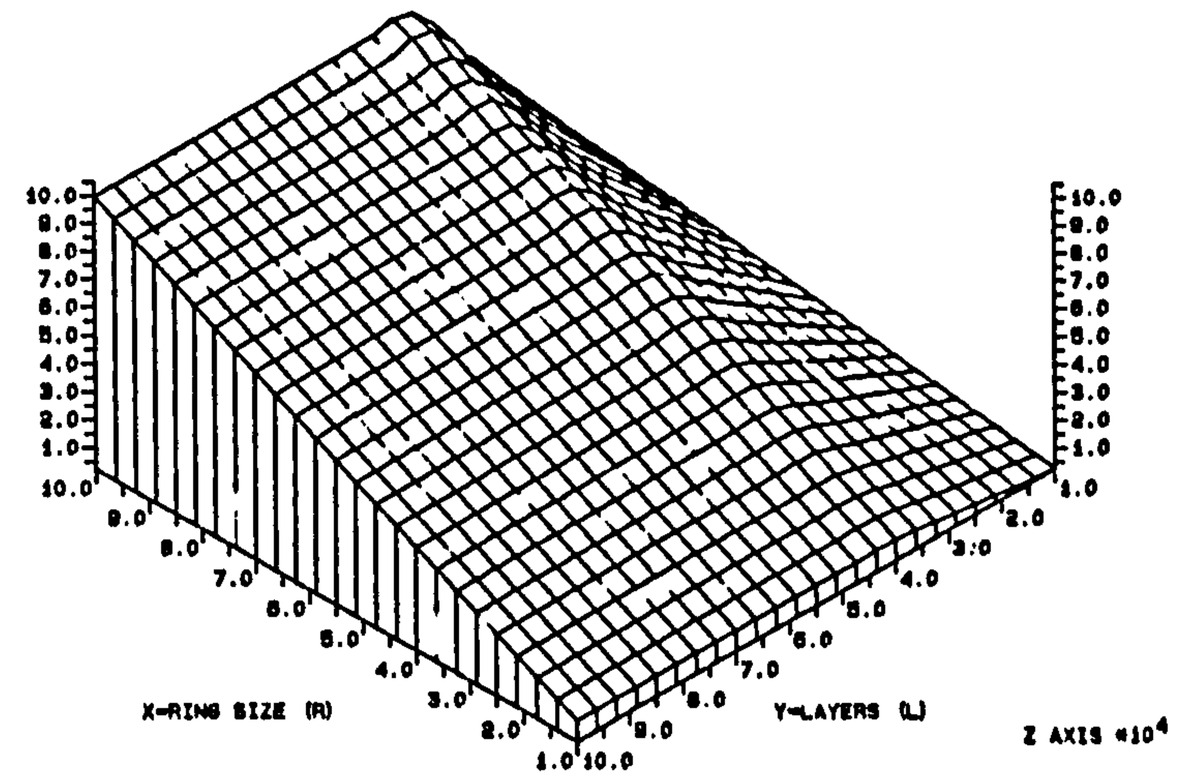


FIG. 5A1.4 HOMOGENEOUS COMMS4 FED AT ALL POINTS WITH DATA OF TYPE 10. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





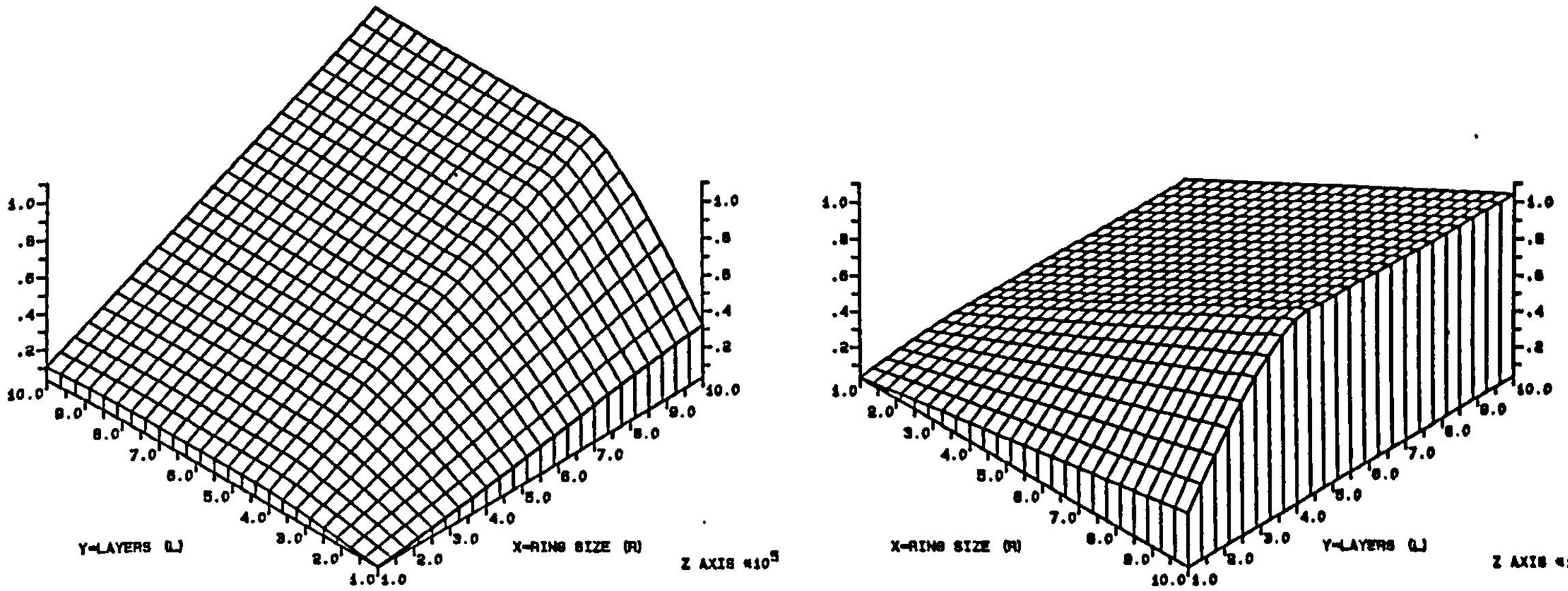
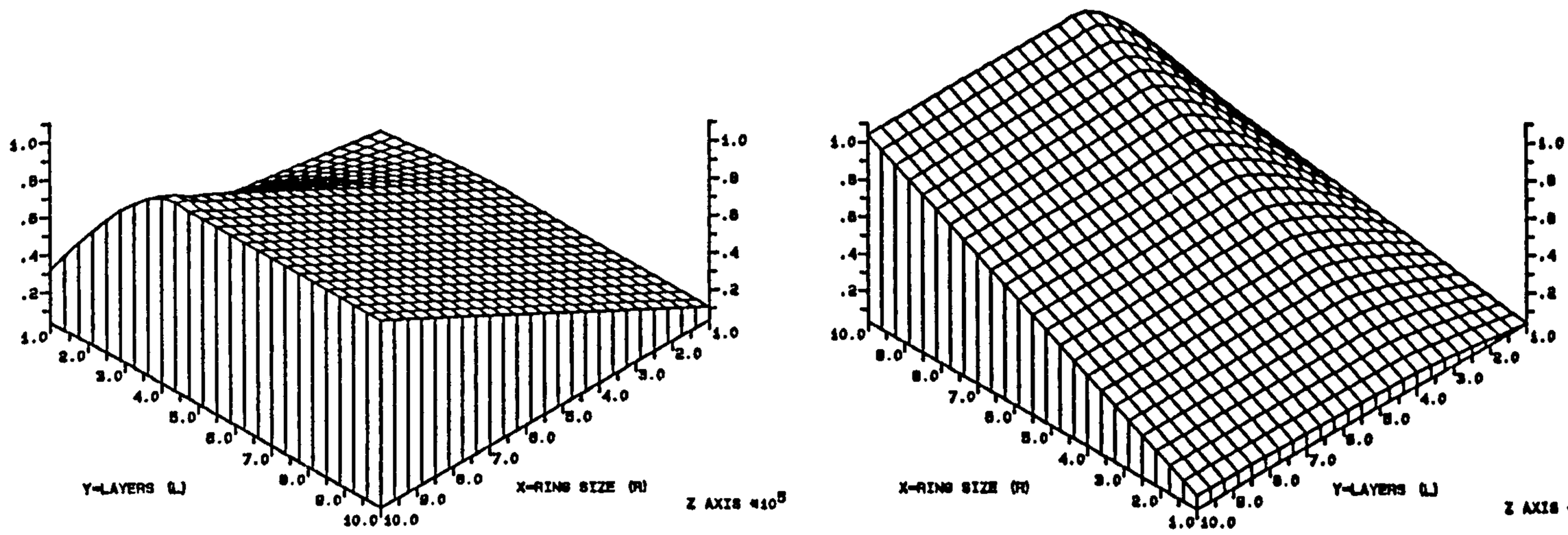


FIG. 5A2.1 HOMOGENEOUS COMMS1 FED AT ALL POINTS WITH DATA OF TYPE R20. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.



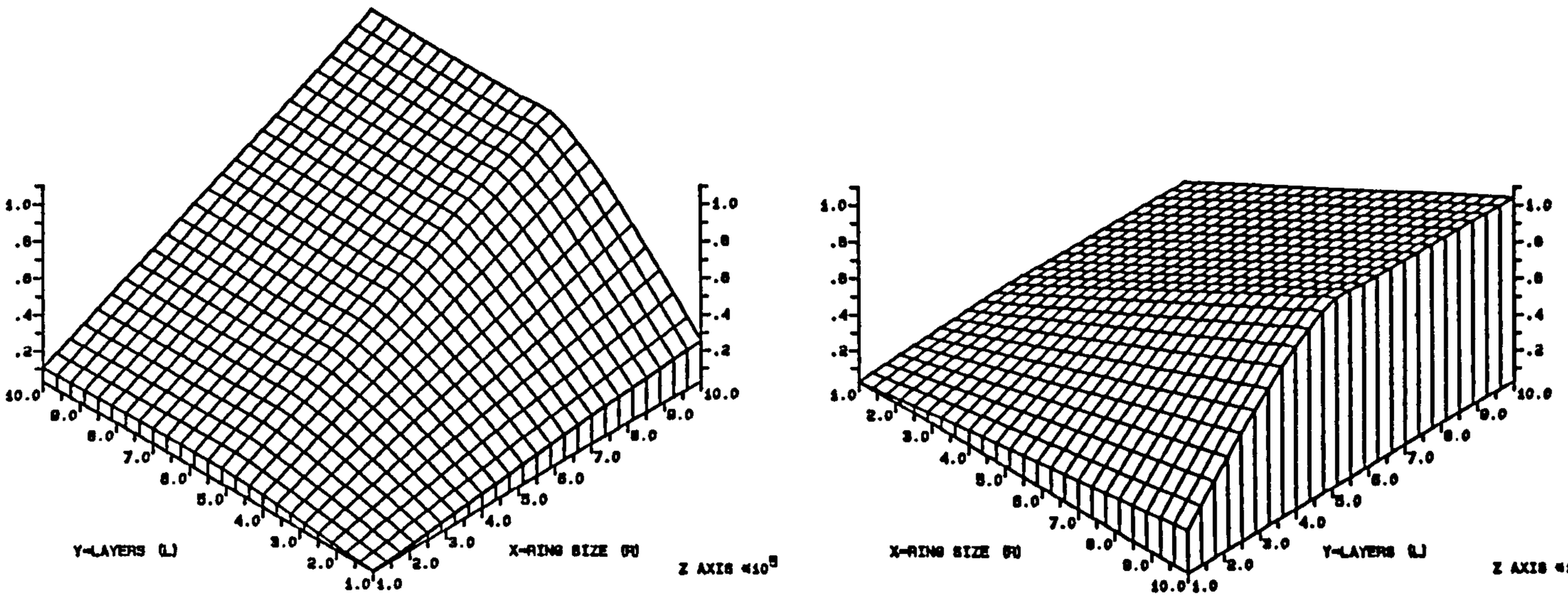
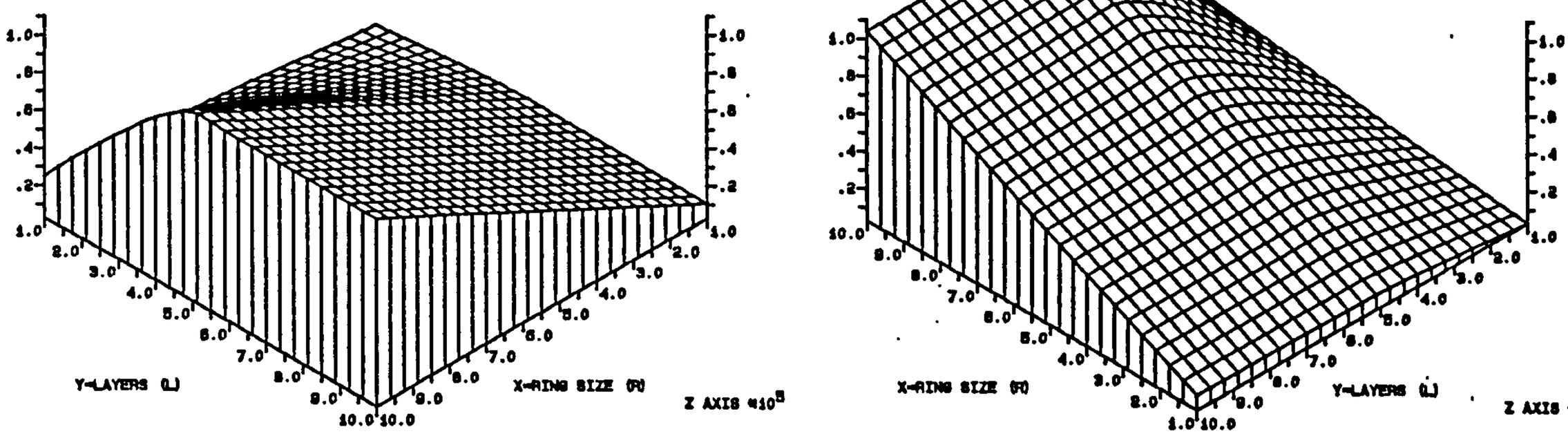
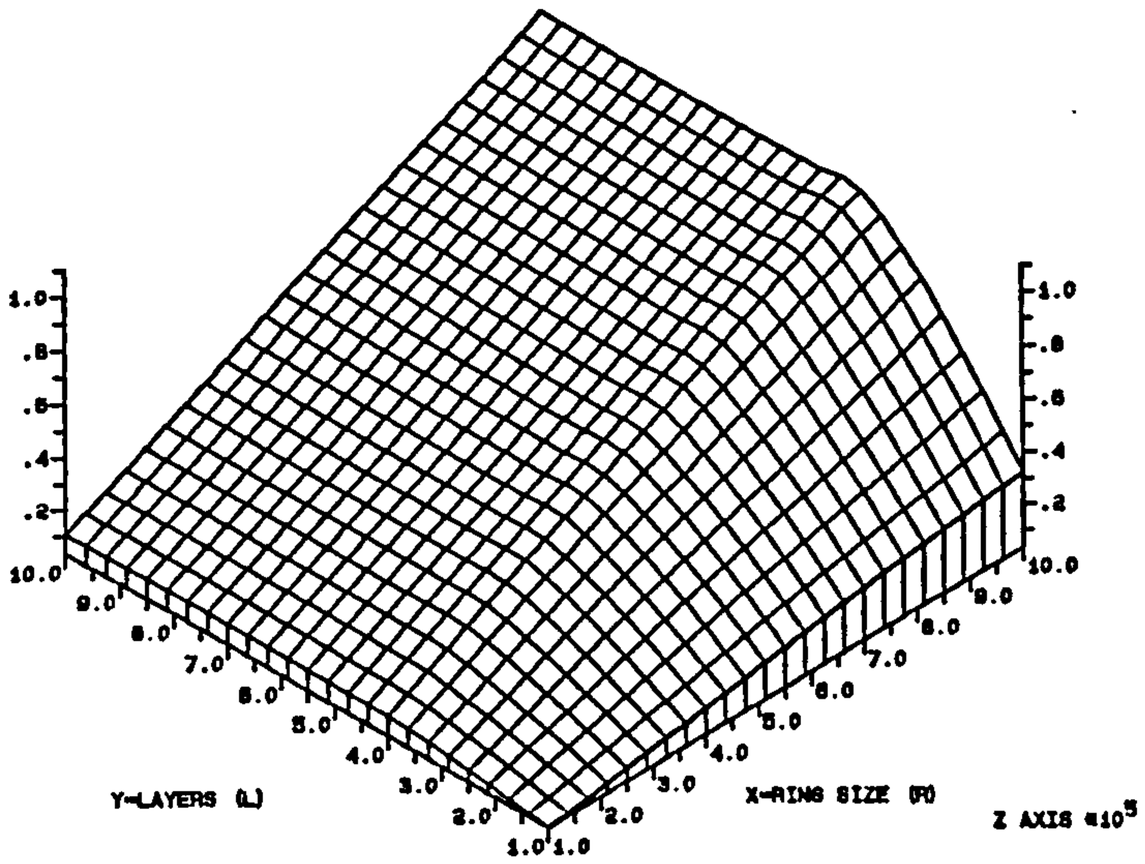


FIG. 5A2.2 HOMOGENEOUS COMMS2 FED AT ALL POINTS WITH DATA OF TYPE R20. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





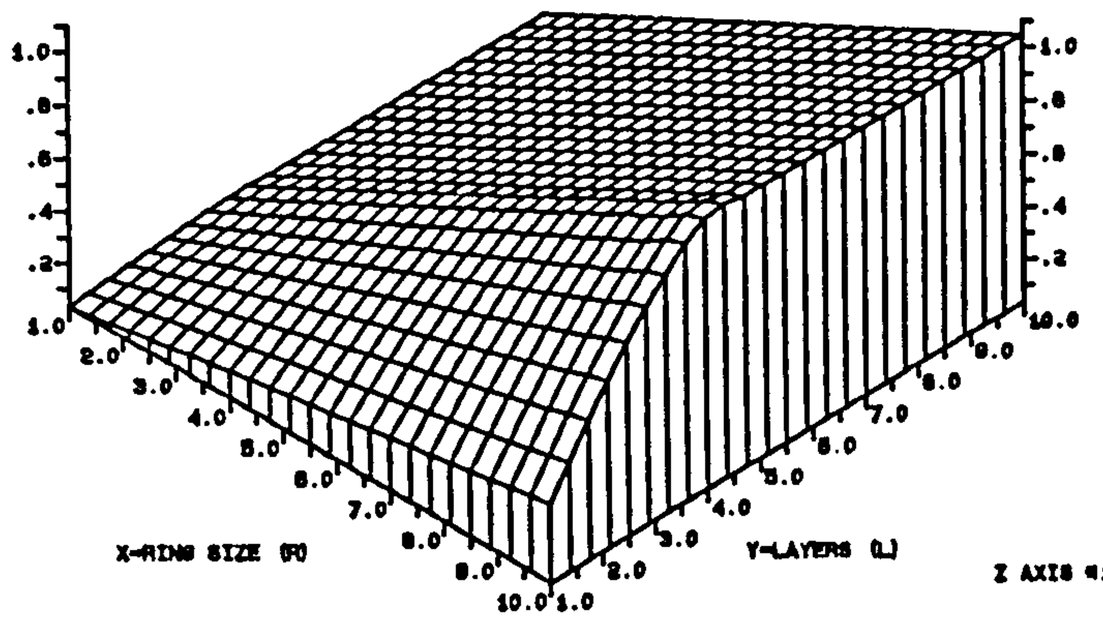
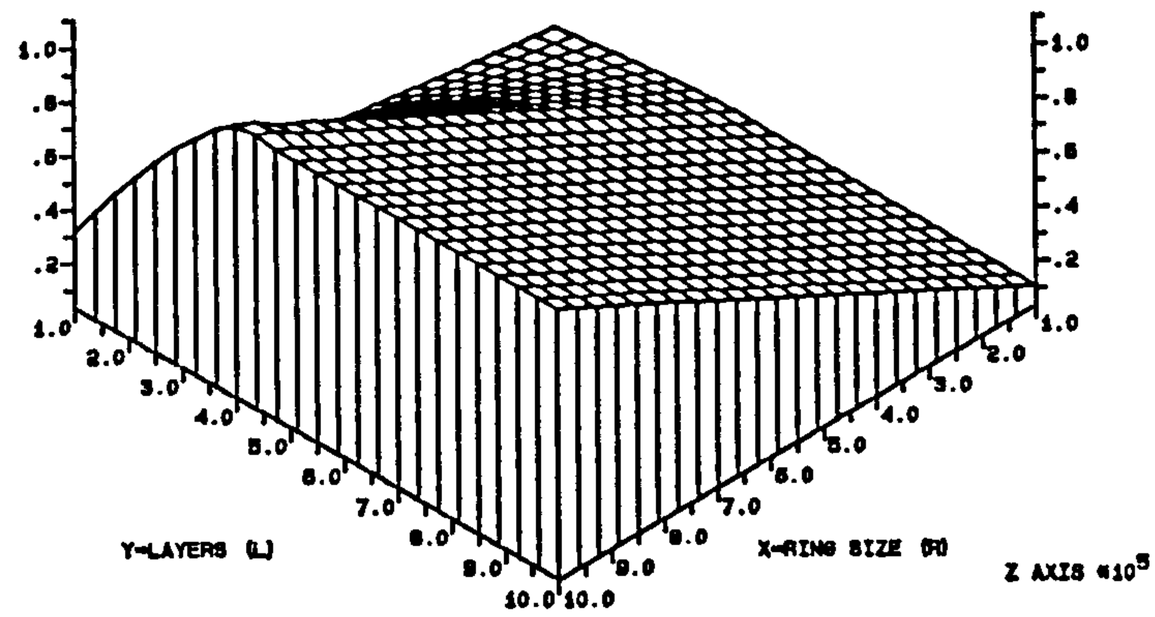
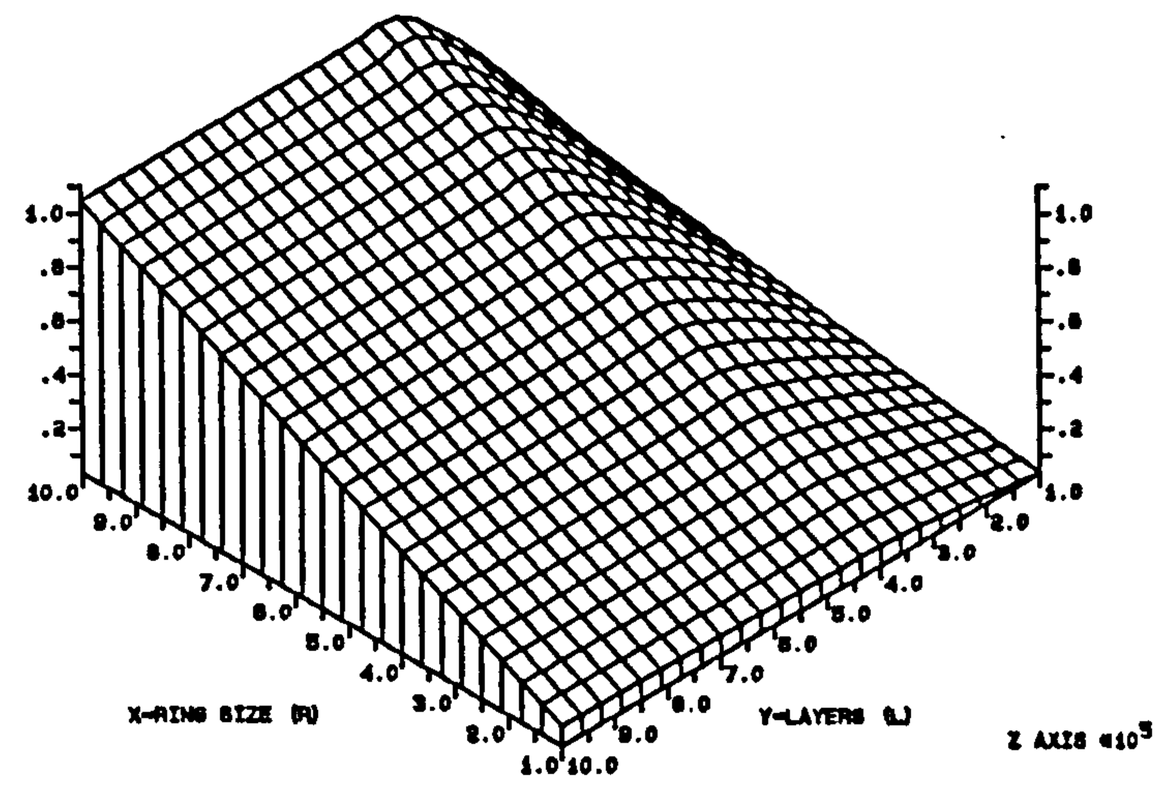


FIG. 5A2.3 HOMOGENEOUS COMMS3 FED AT ALL POINTS WITH DATA OF TYPE R20. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





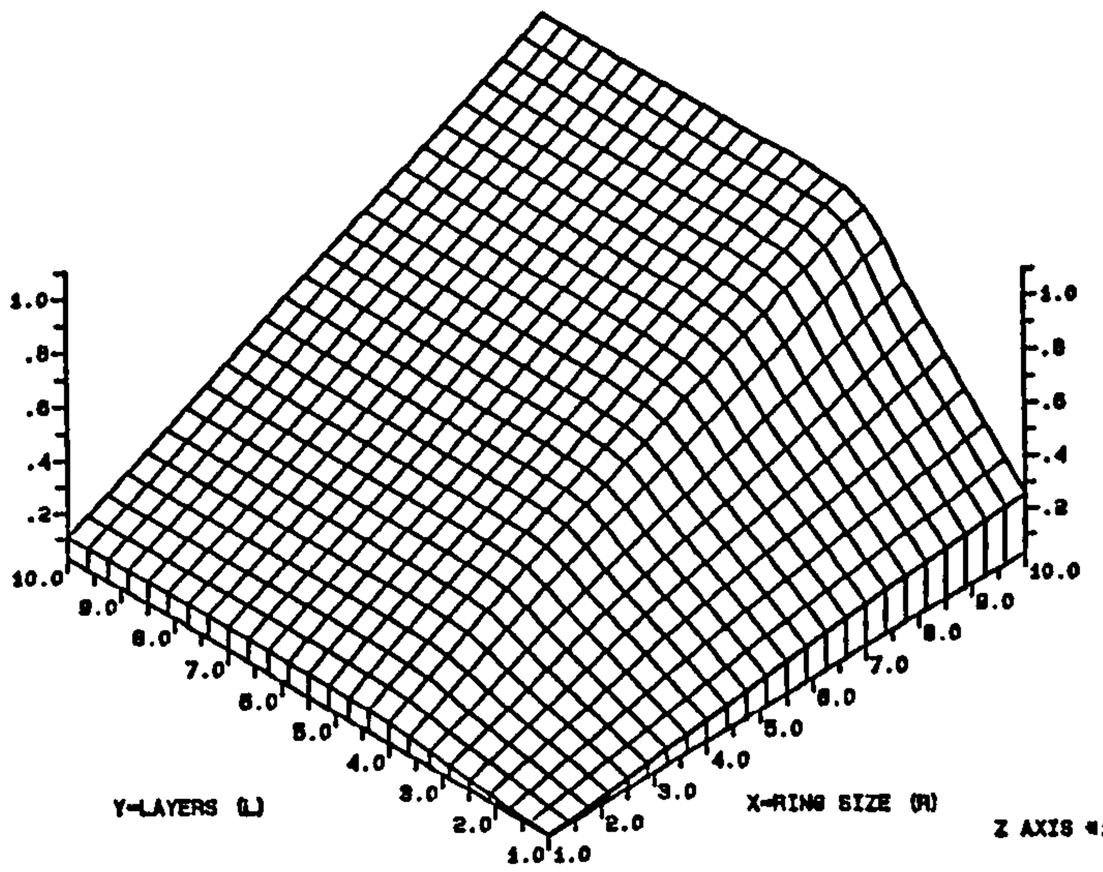
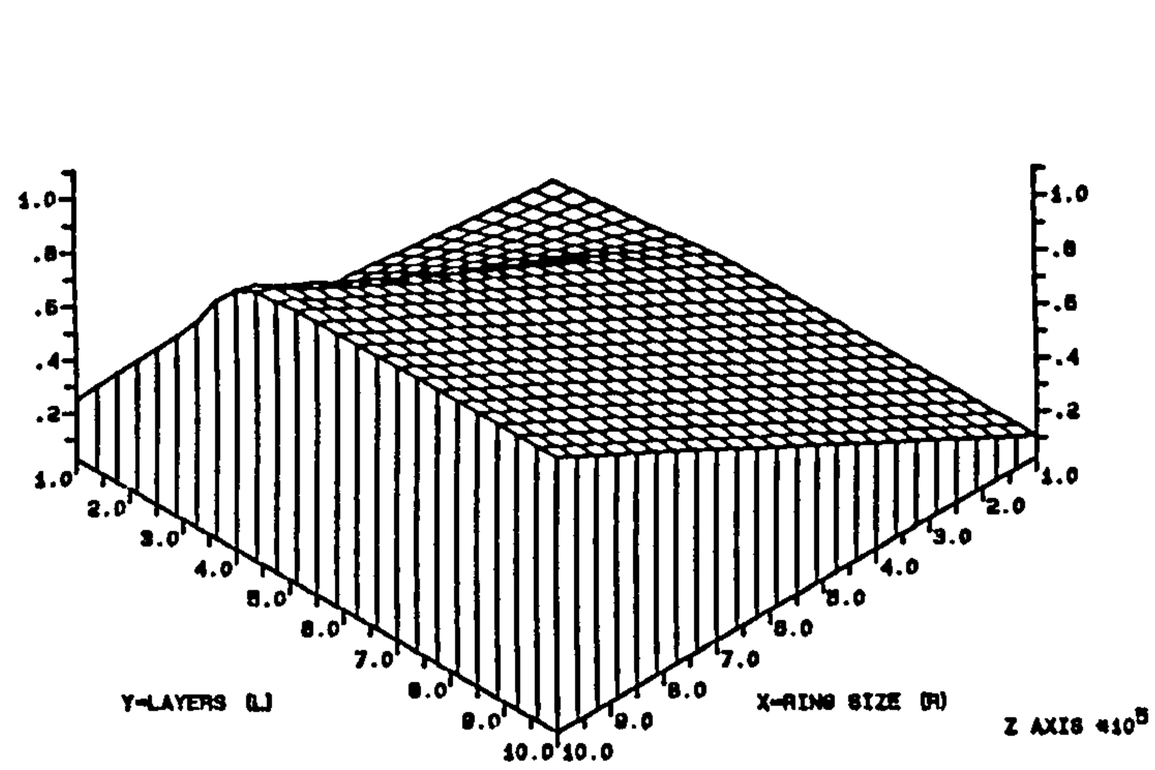
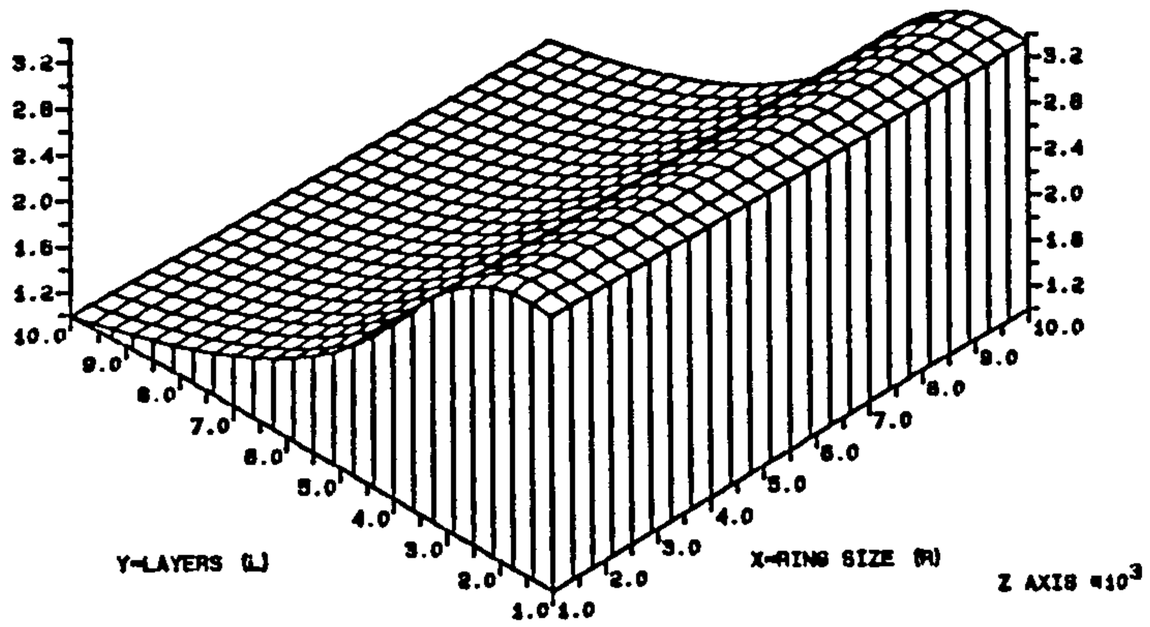


FIG. 5A2.4 HOMOGENEOUS COMMS4 FED AT ALL POINTS WITH DATA OF TYPE R20. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





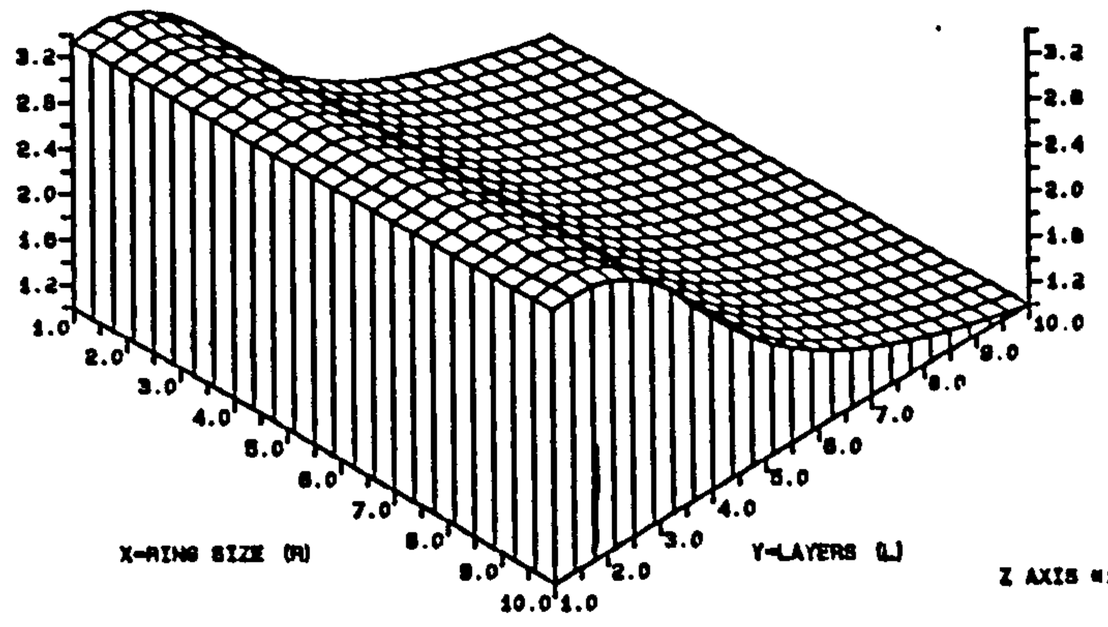
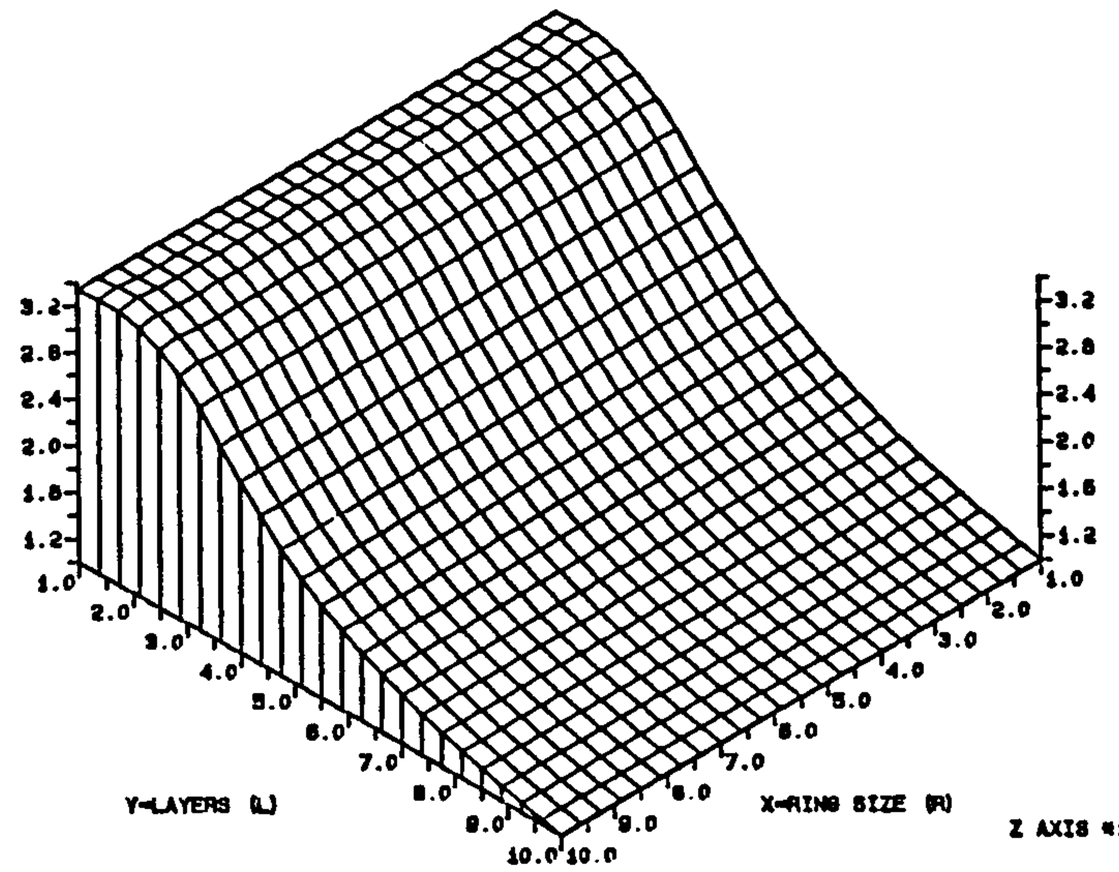
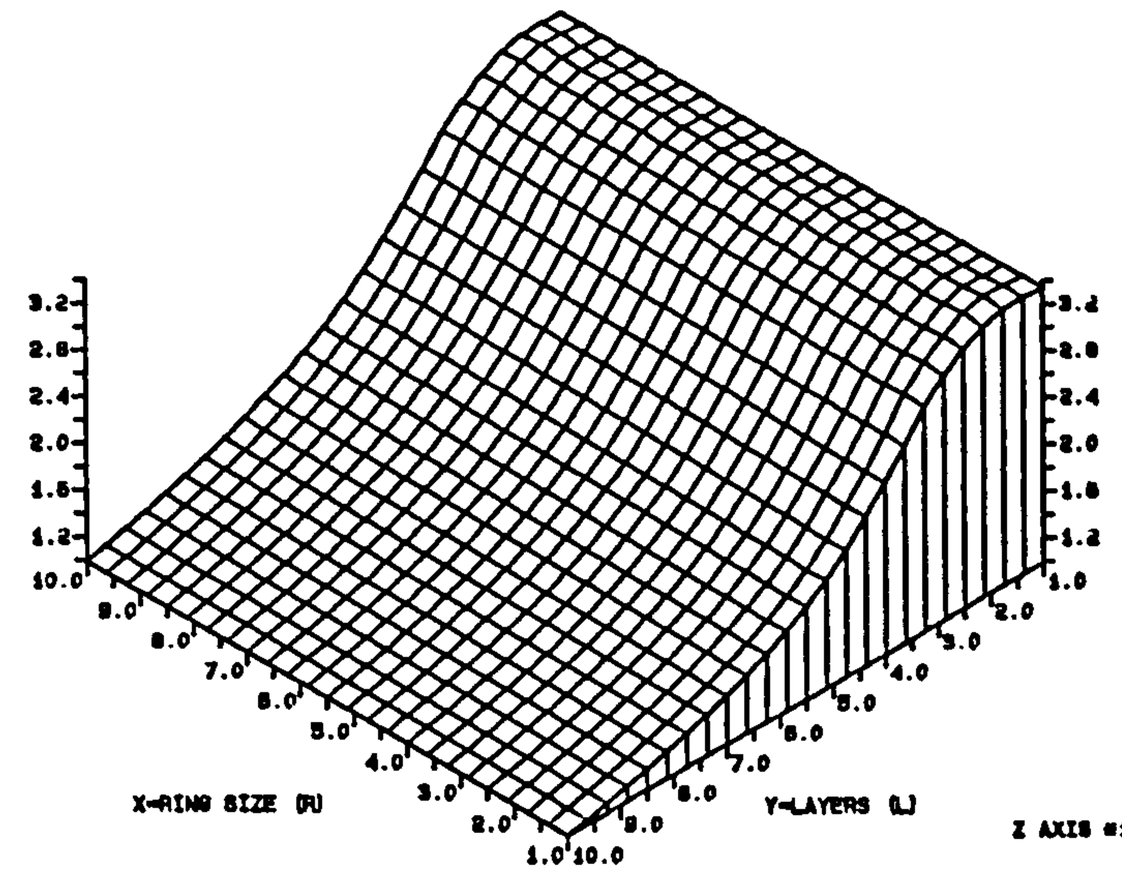


FIG. 5A3.1 HOMOGENEOUS COMMS1 FED AT ALL POINTS WITH DATA OF TYPE 10. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





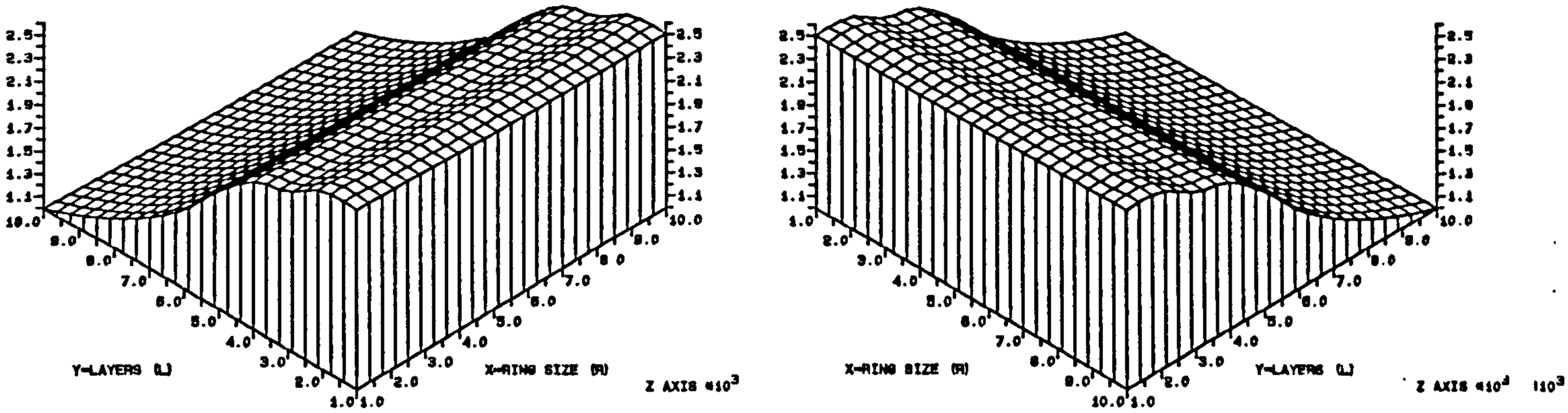
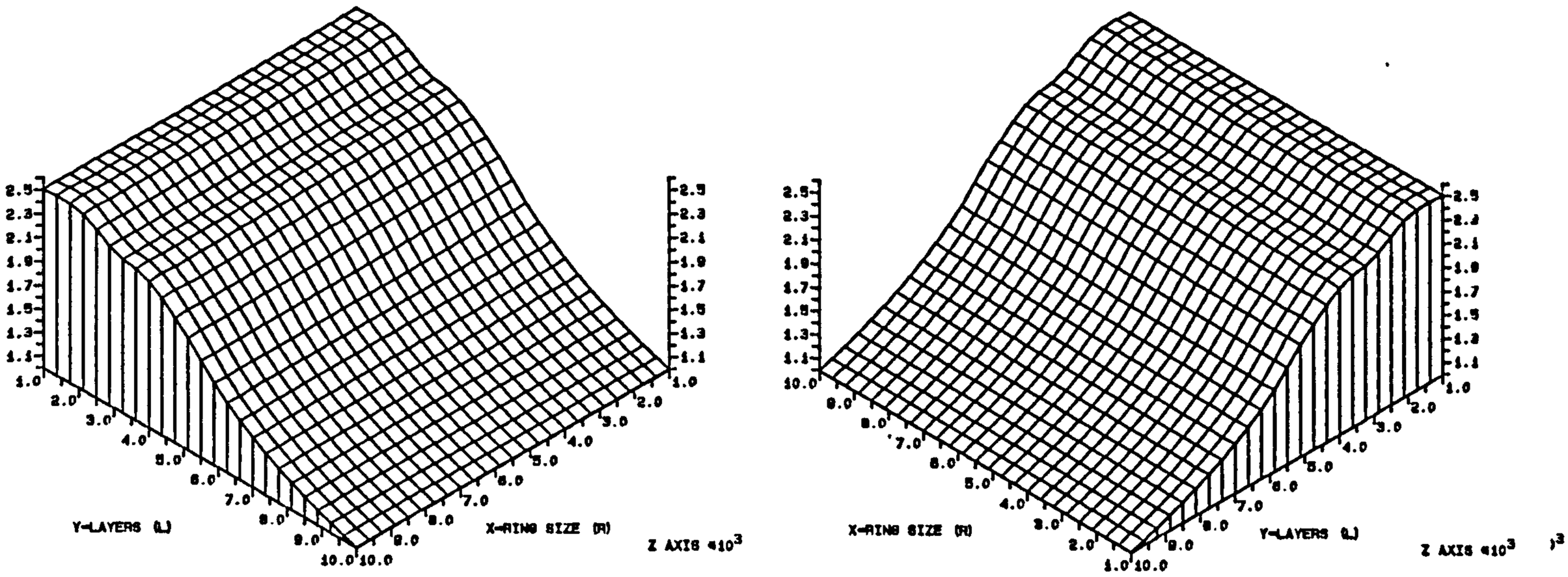
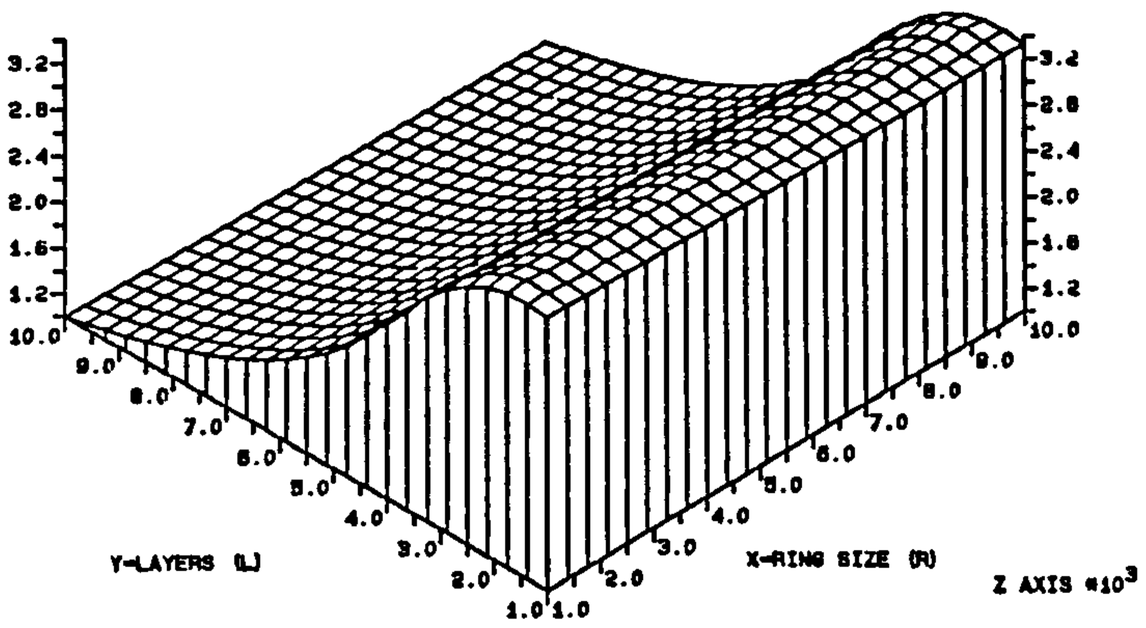


FIG. 5A3.2 HOMOGENEOUS COMMS2 FED AT ALL POINTS WITH DATA OF TYPE 10. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





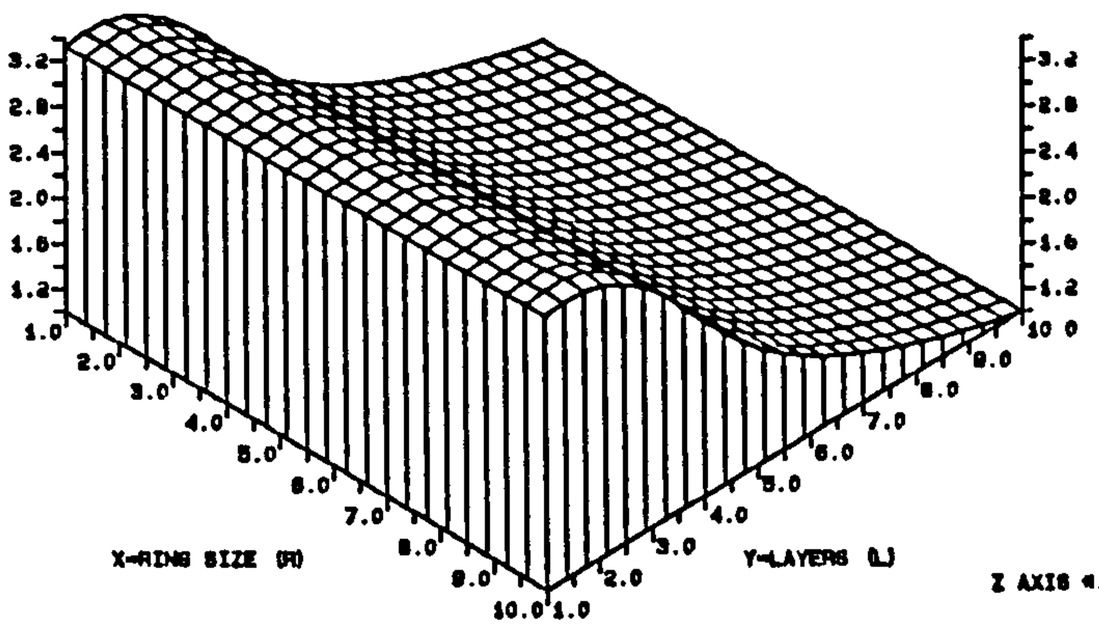
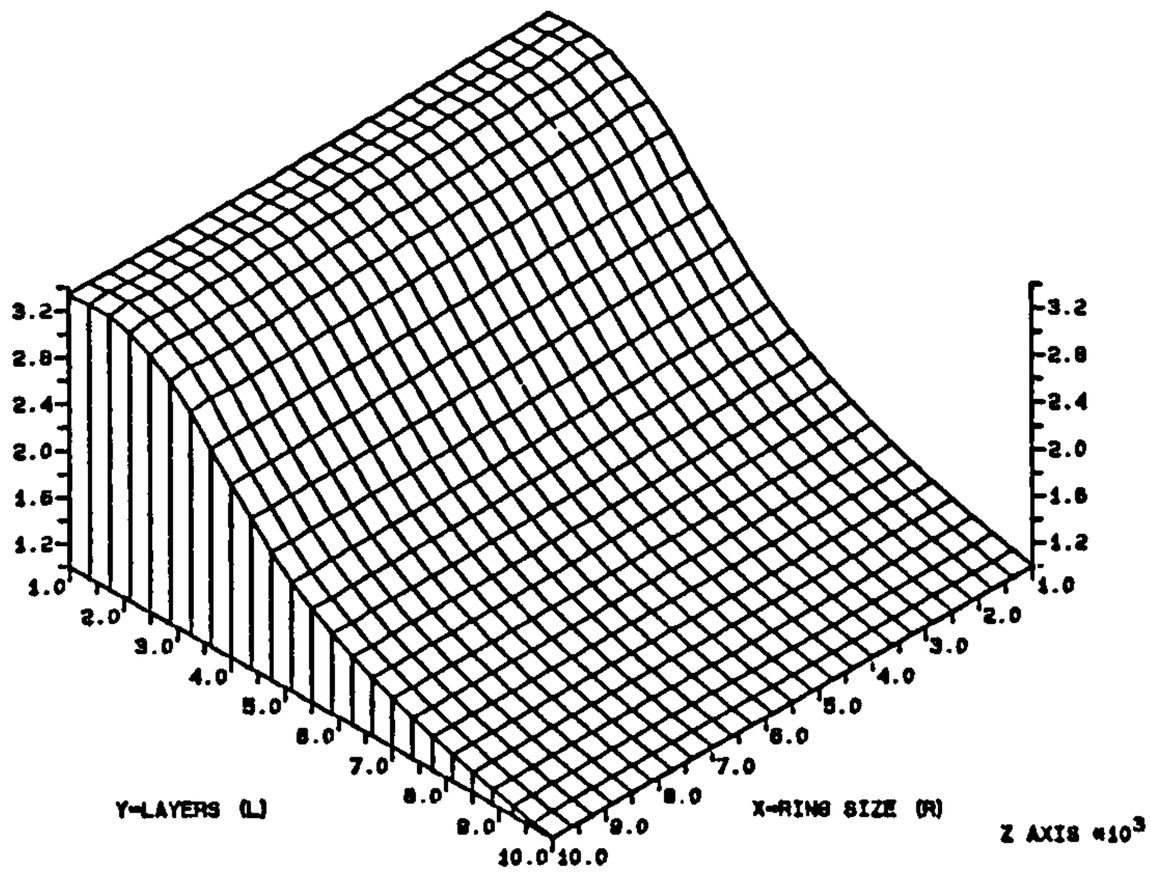
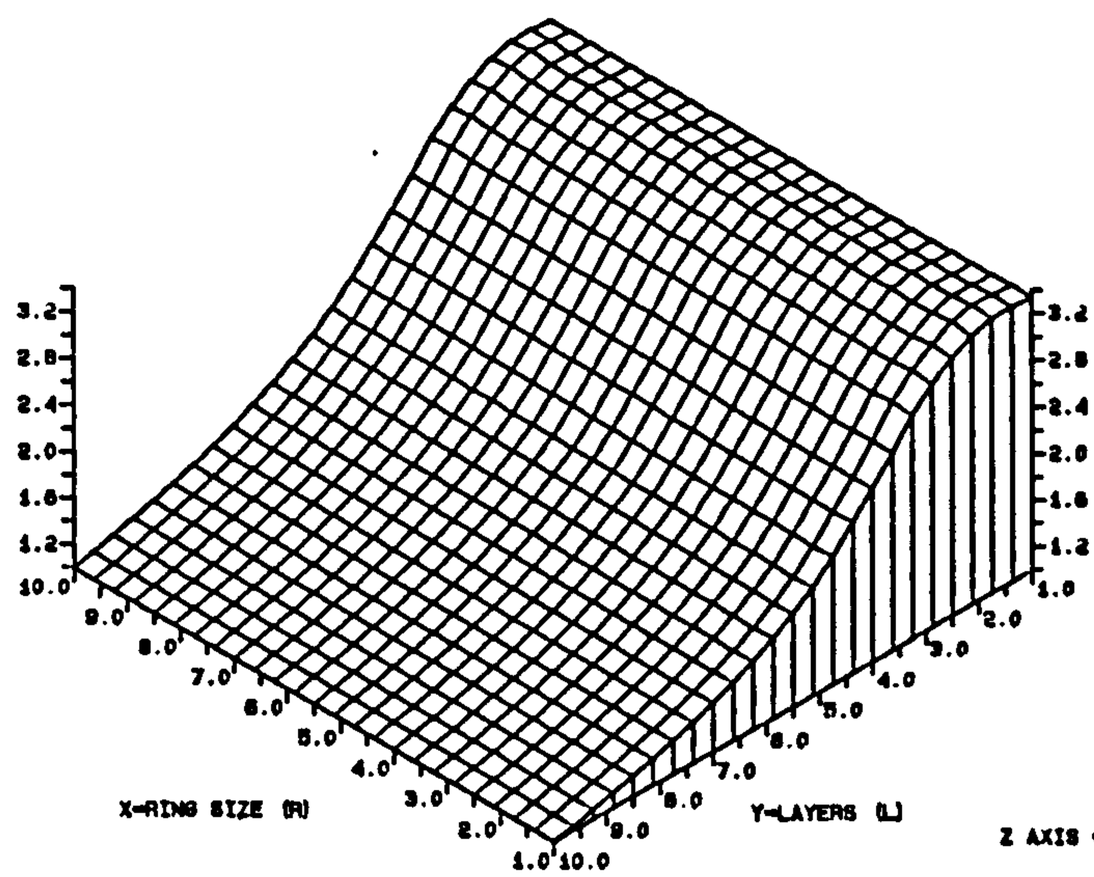


FIG. 5A3.3 HOMOGENEOUS COMMS3 FED AT ALL POINTS WITH DATA OF TYPE 10. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





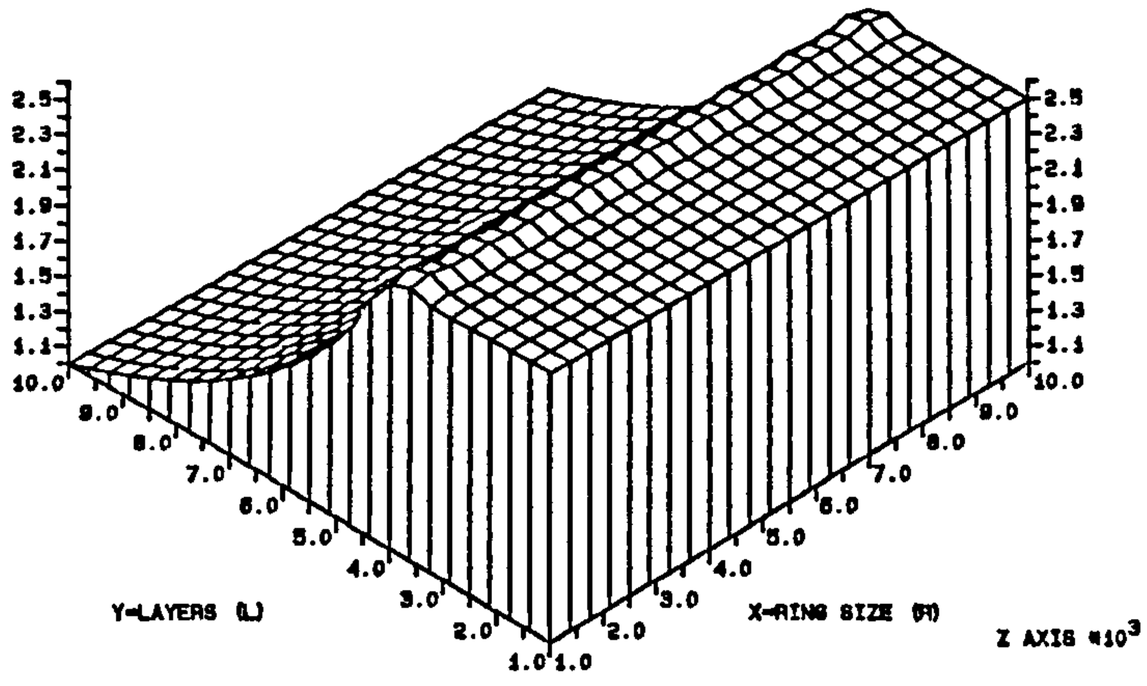
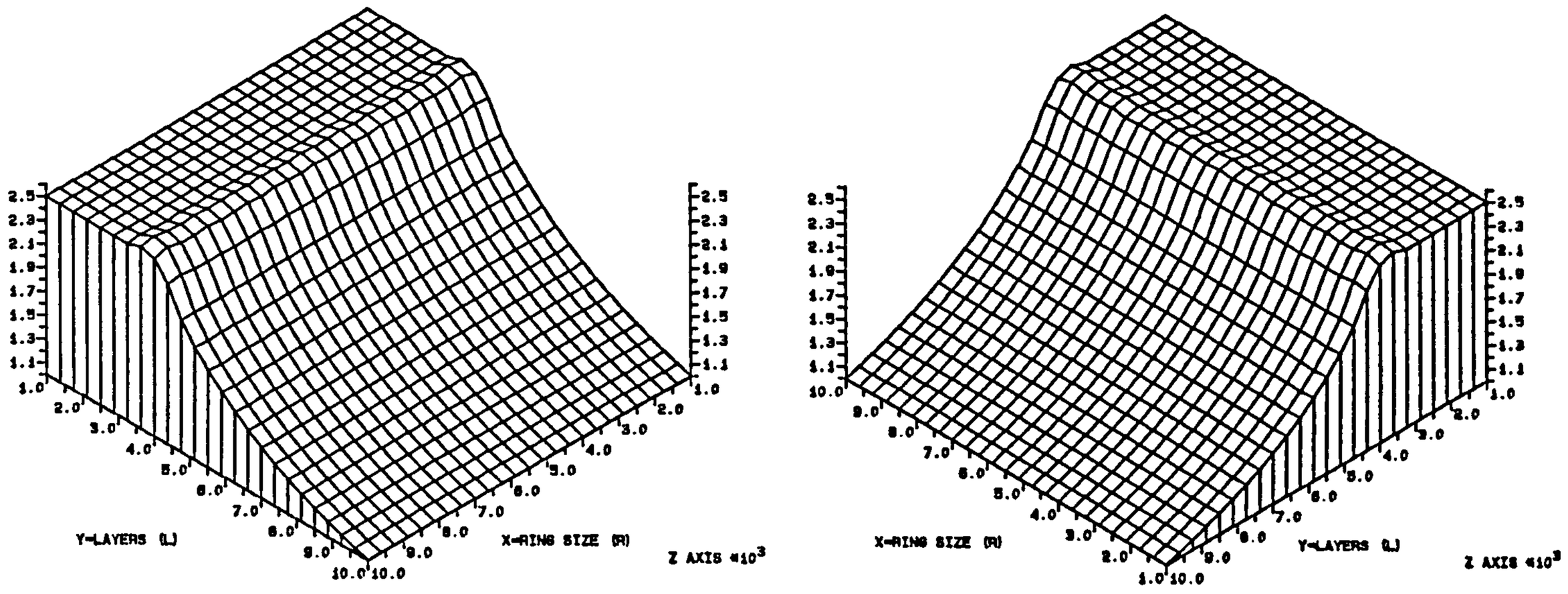
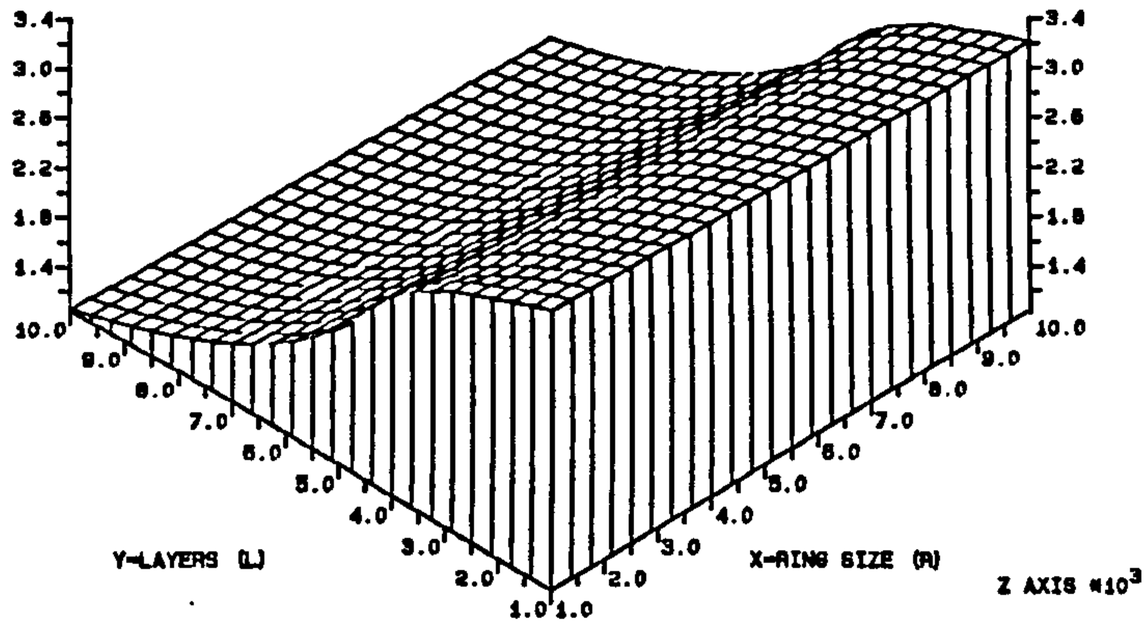


FIG. 5A3.4 HOMOGENEOUS COMMS4 FED AT ALL POINTS WITH DATA OF TYPE 10. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





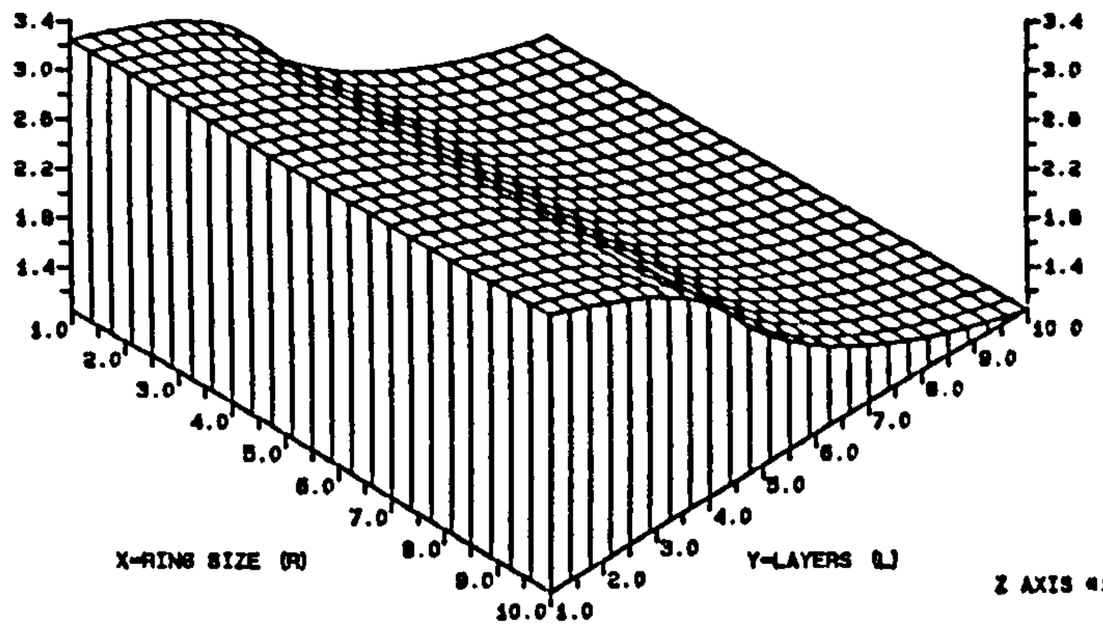
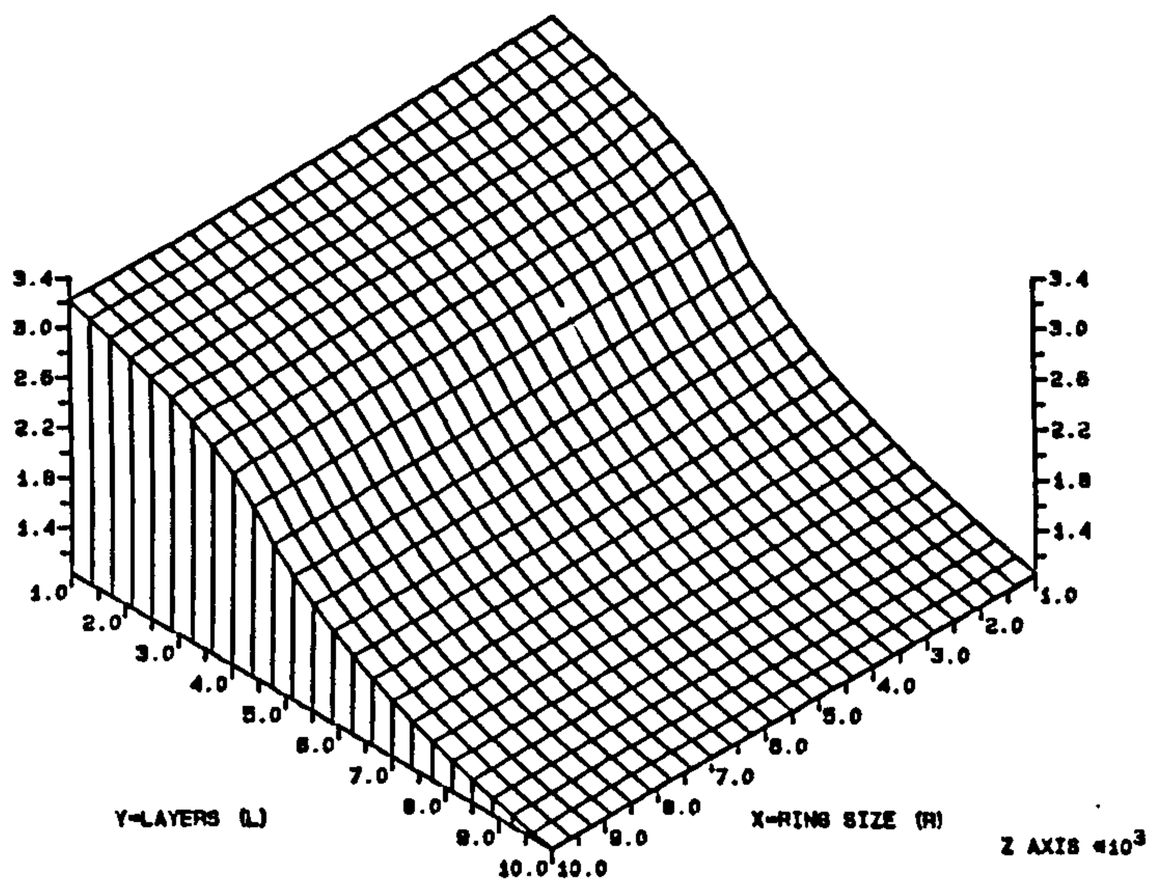
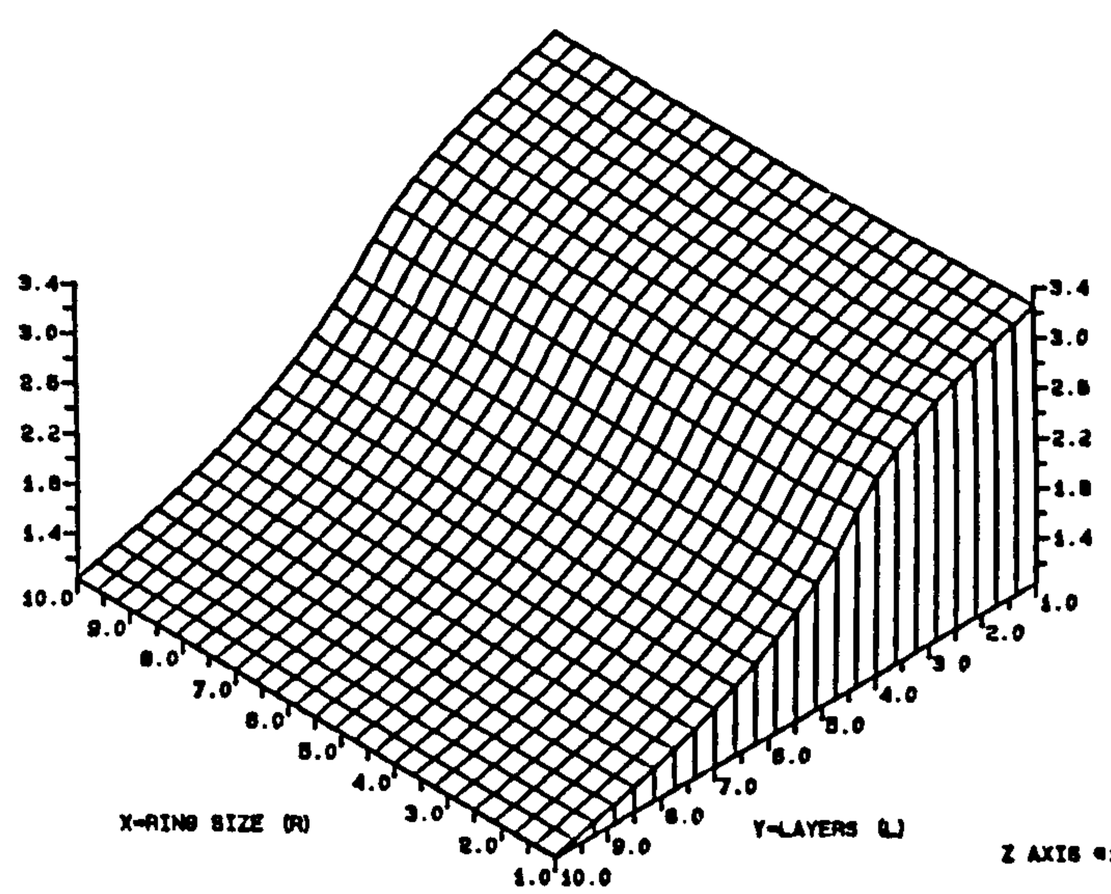


FIG. 5A4.1 HOMOGENEOUS COMMS1 FED AT ALL POINTS WITH DATA OF TYPE R20. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





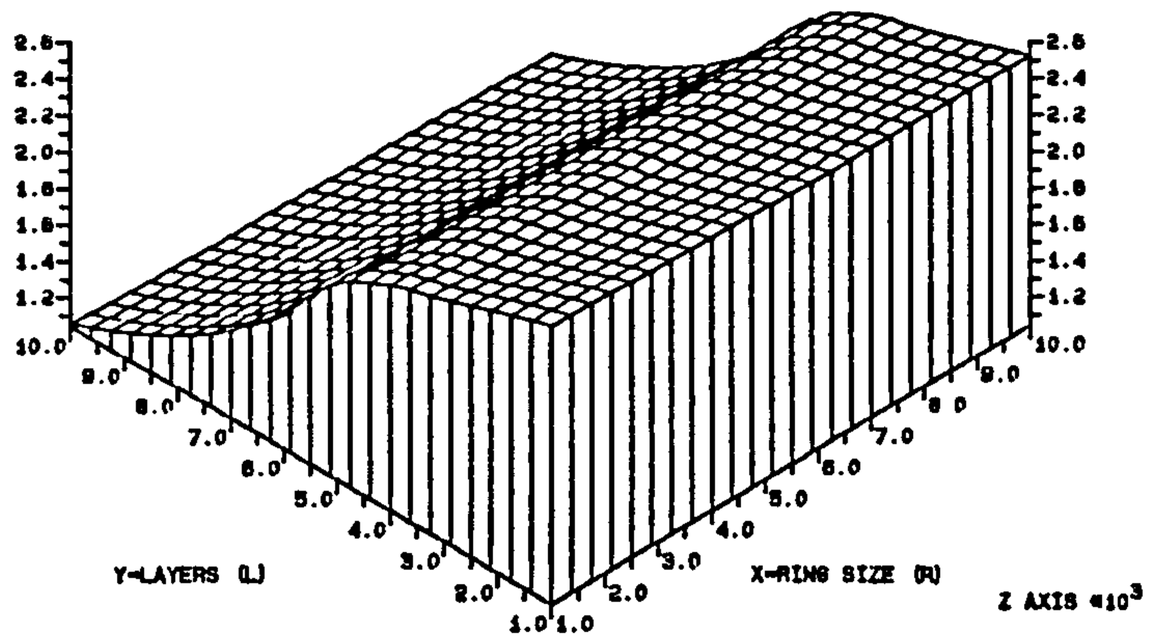
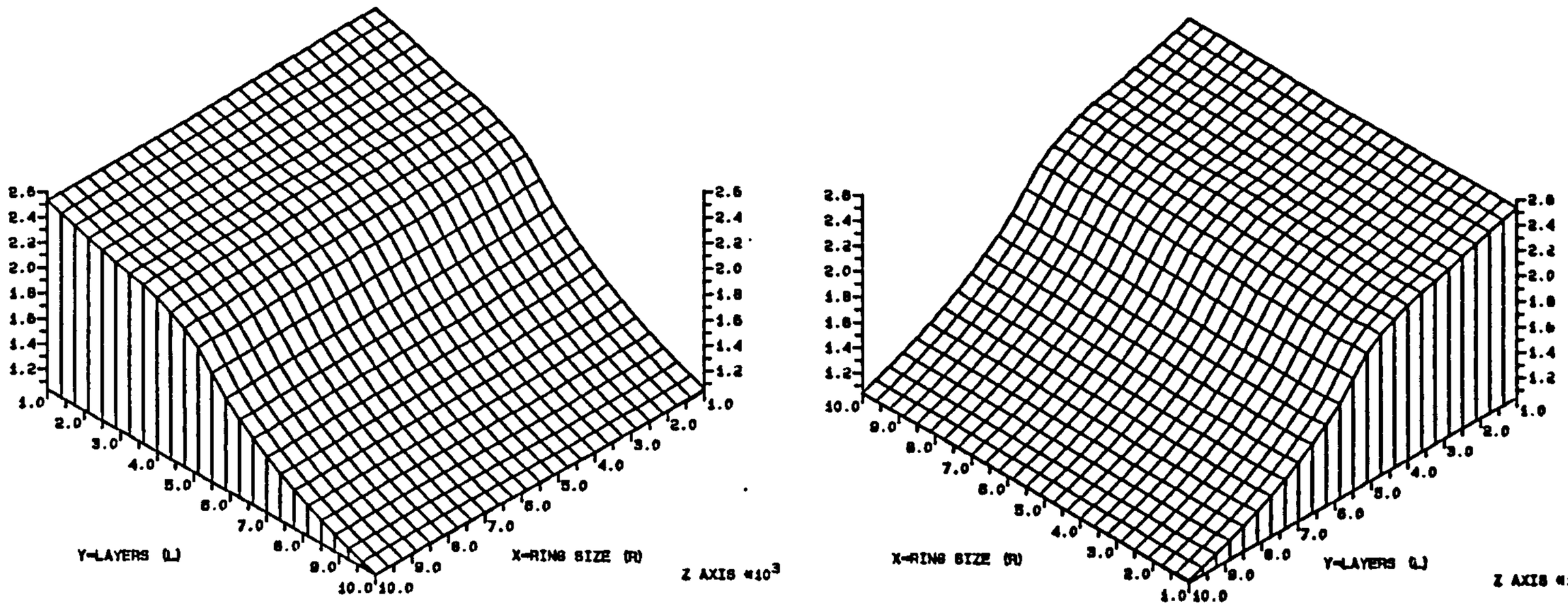
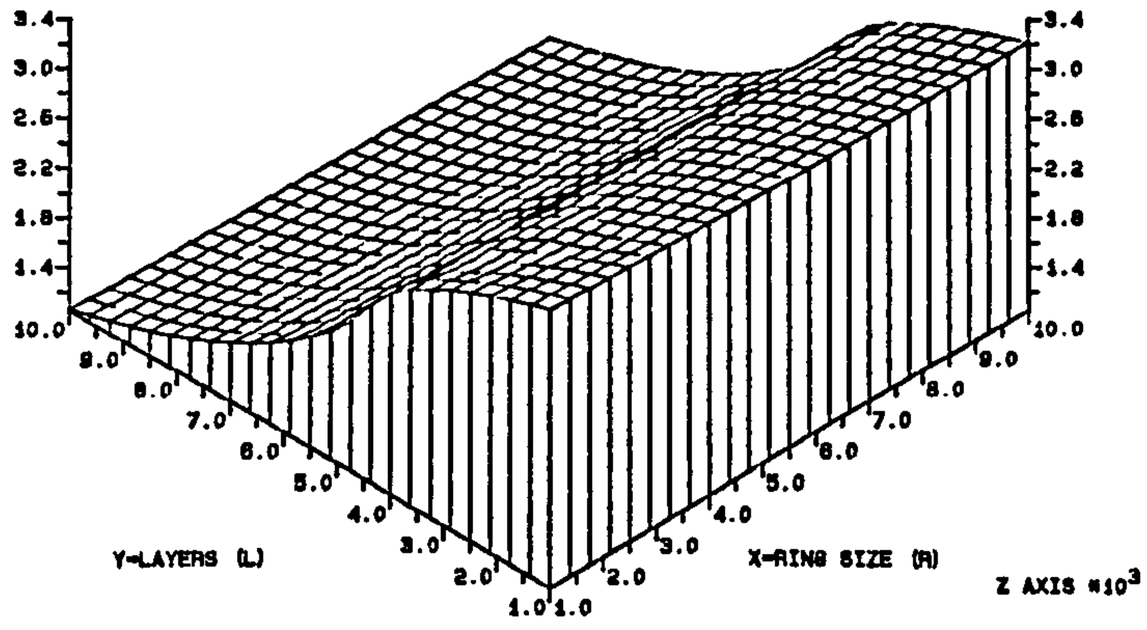


FIG. 5A4.2 HOMOGENEOUS COMMS2 FED AT ALL POINTS WITH DATA OF TYPE R20. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





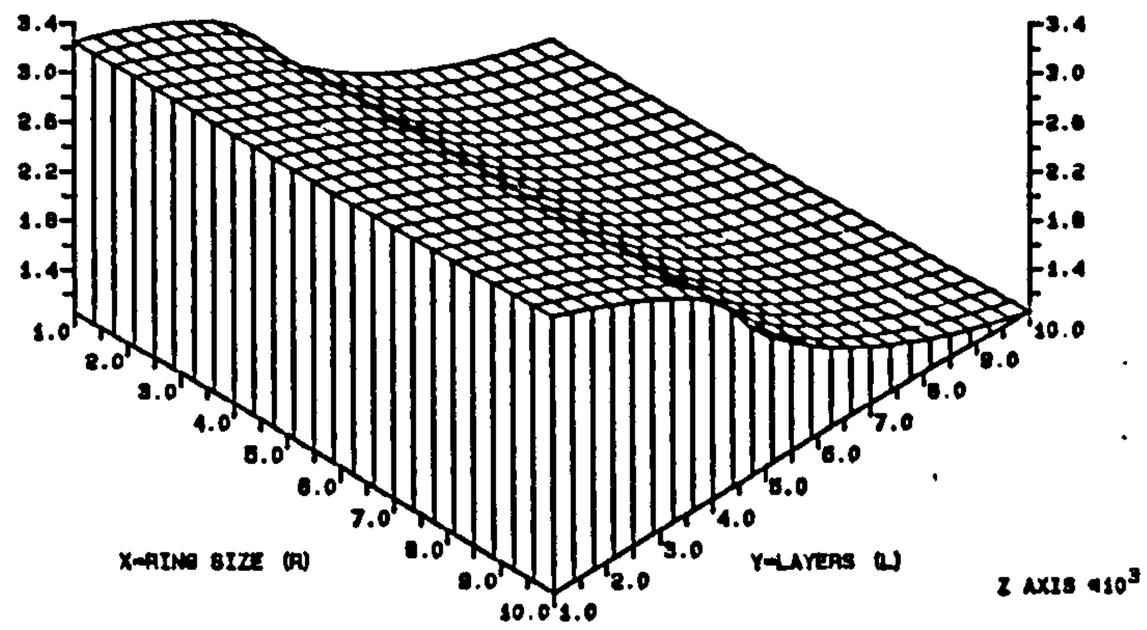
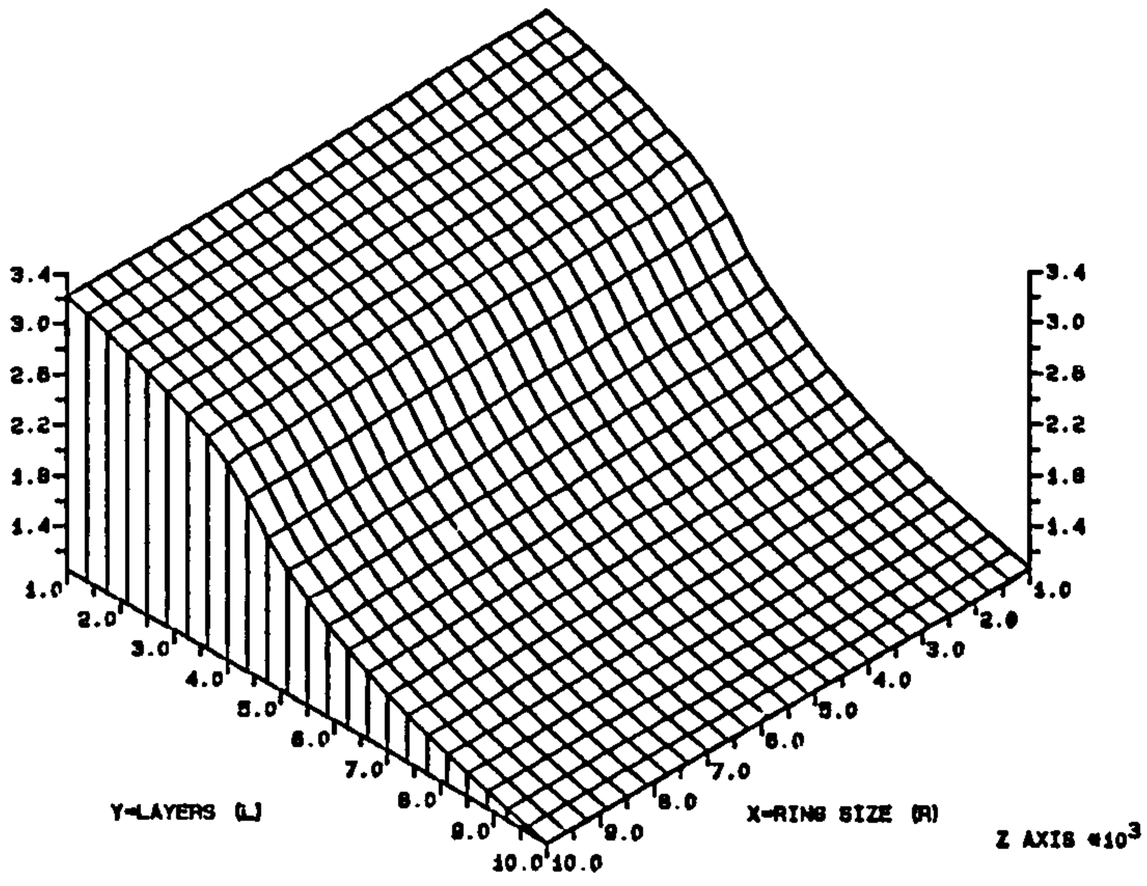
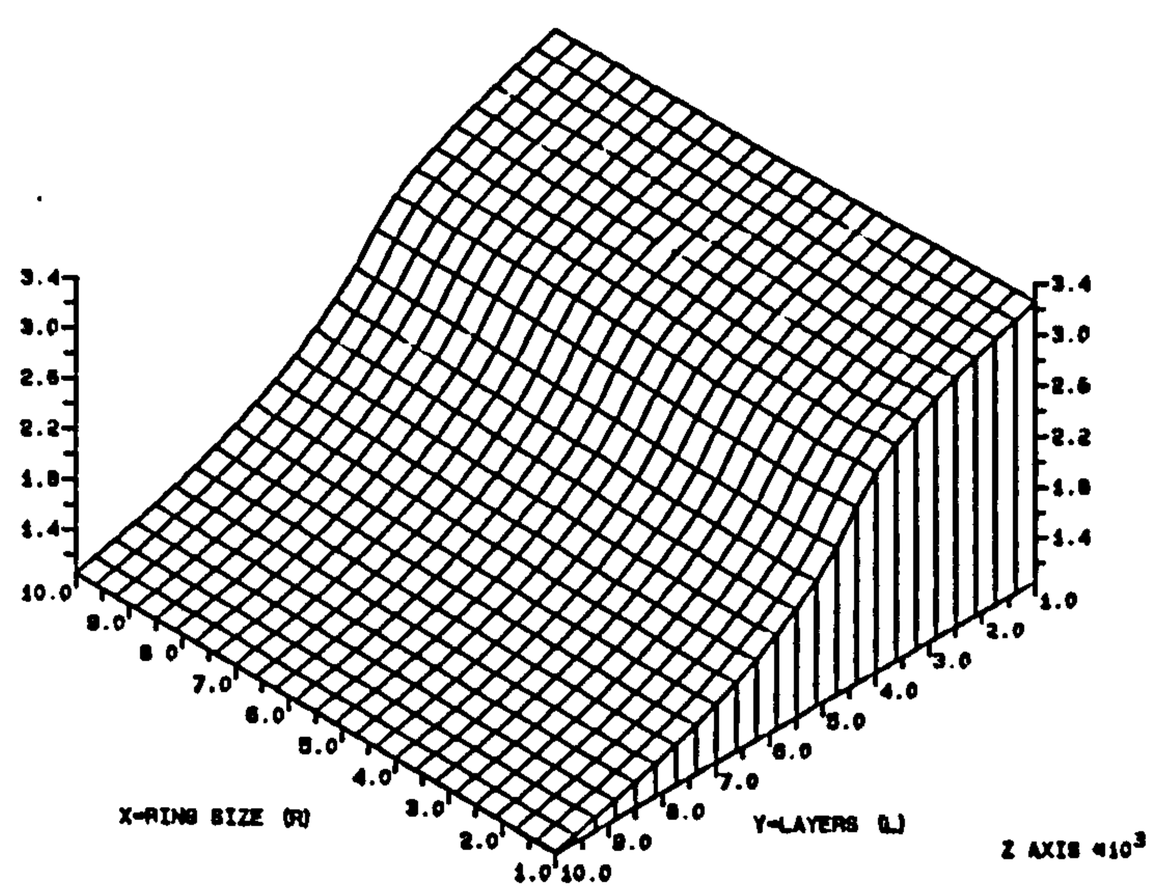


FIG. 5A4.3 HOMOGENEOUS COMMS3 FED AT ALL POINTS WITH DATA OF TYPE R20. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





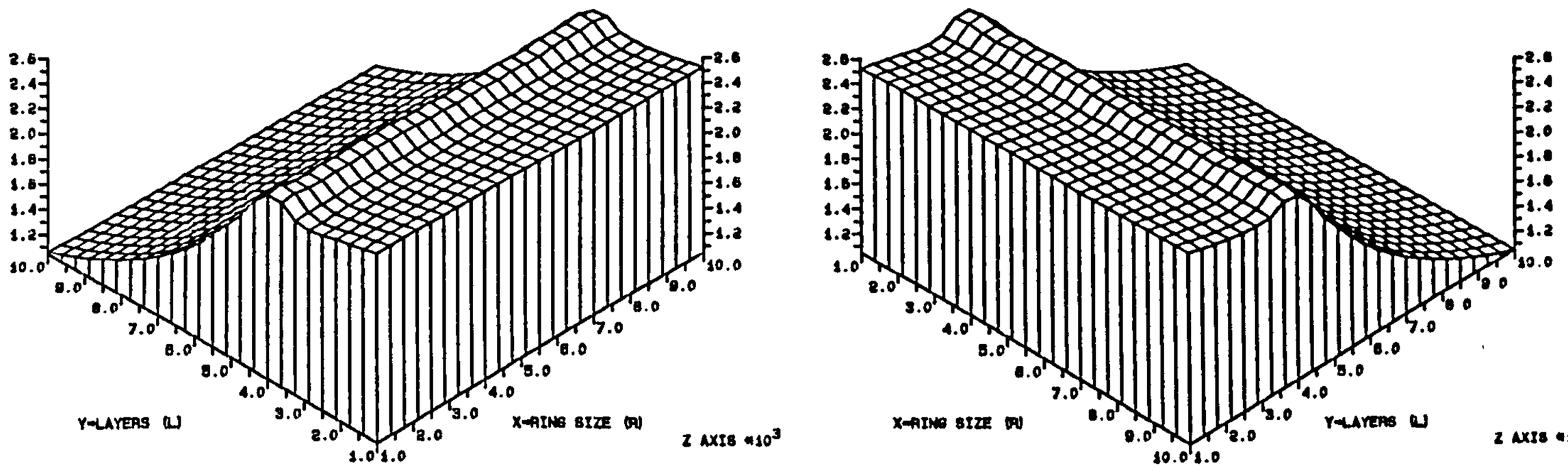
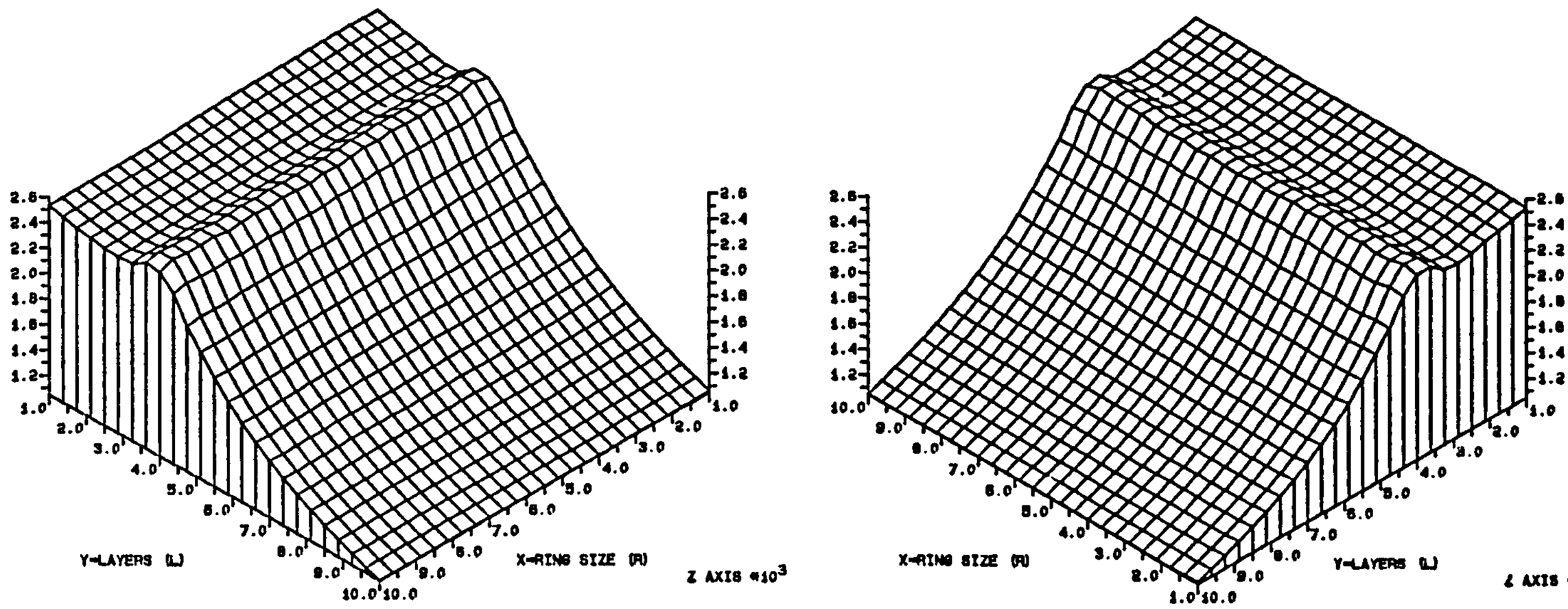
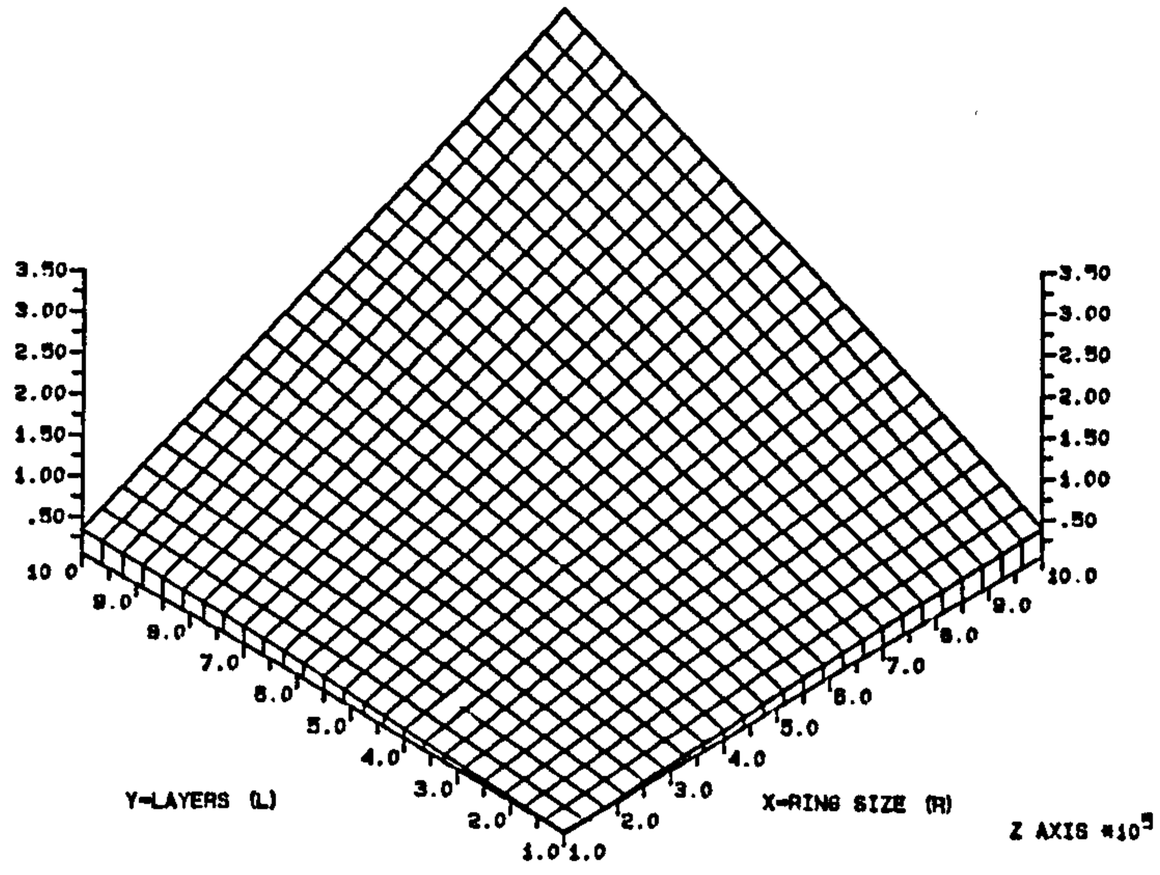


FIG. 5A4.4 HOMOGENEOUS COMMS4 FED AT ALL POINTS WITH DATA OF TYPE R20. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





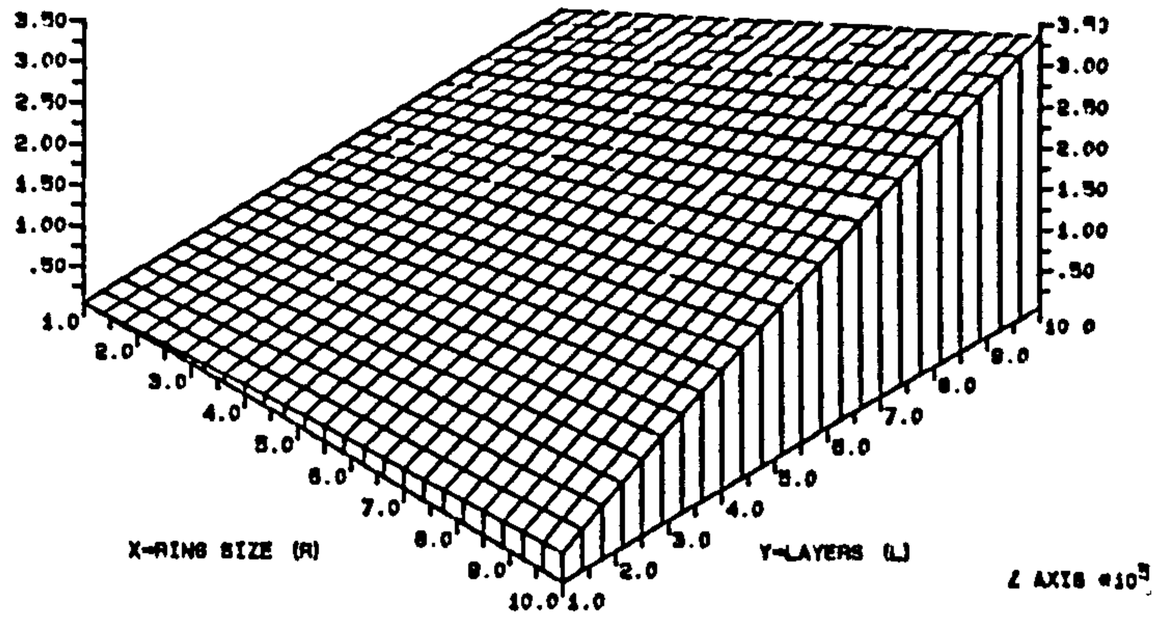
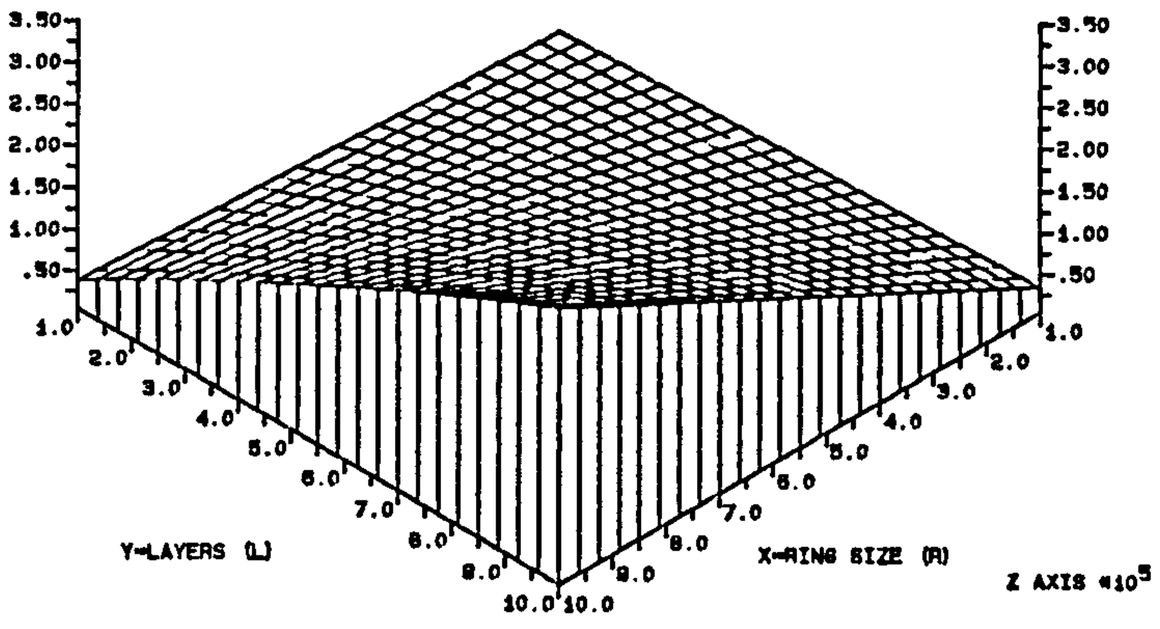
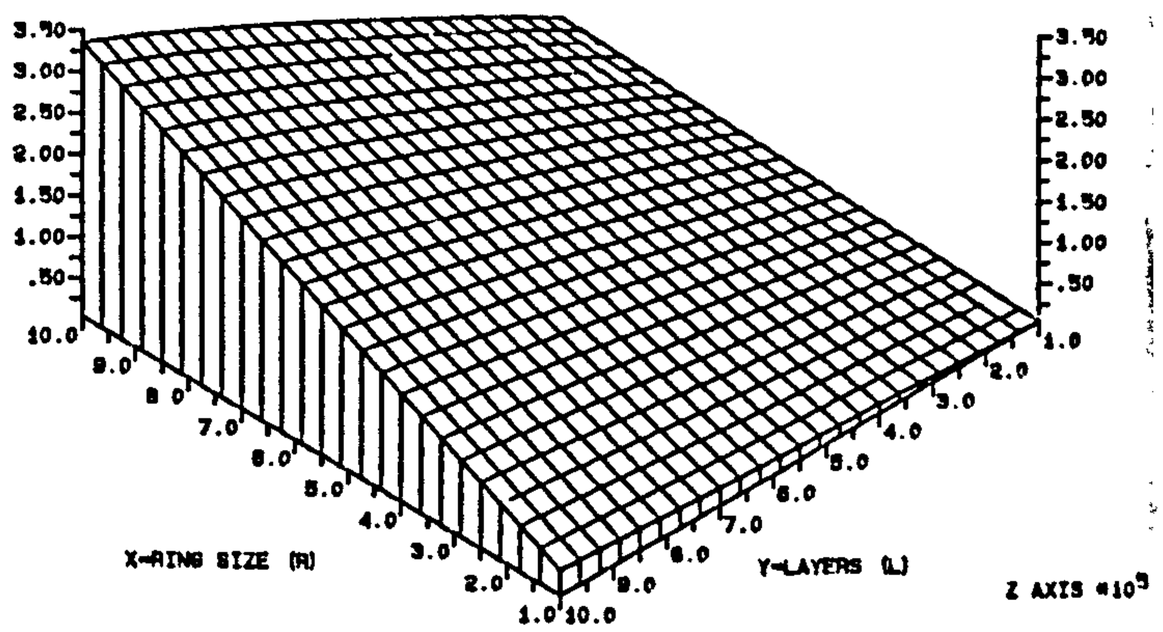


FIG. 5A5.1 HOMOGENEOUS COMMS1 FED AT ALL POINTS WITH DATA OF TYPE 50. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





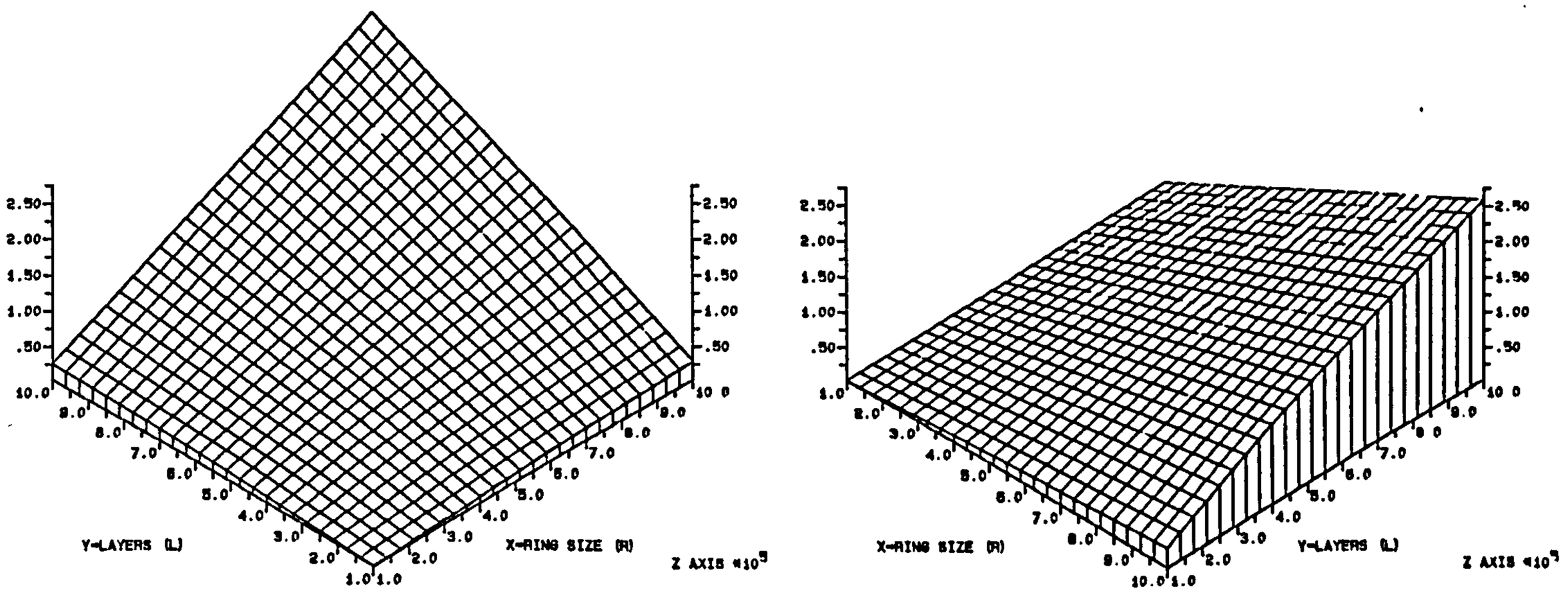
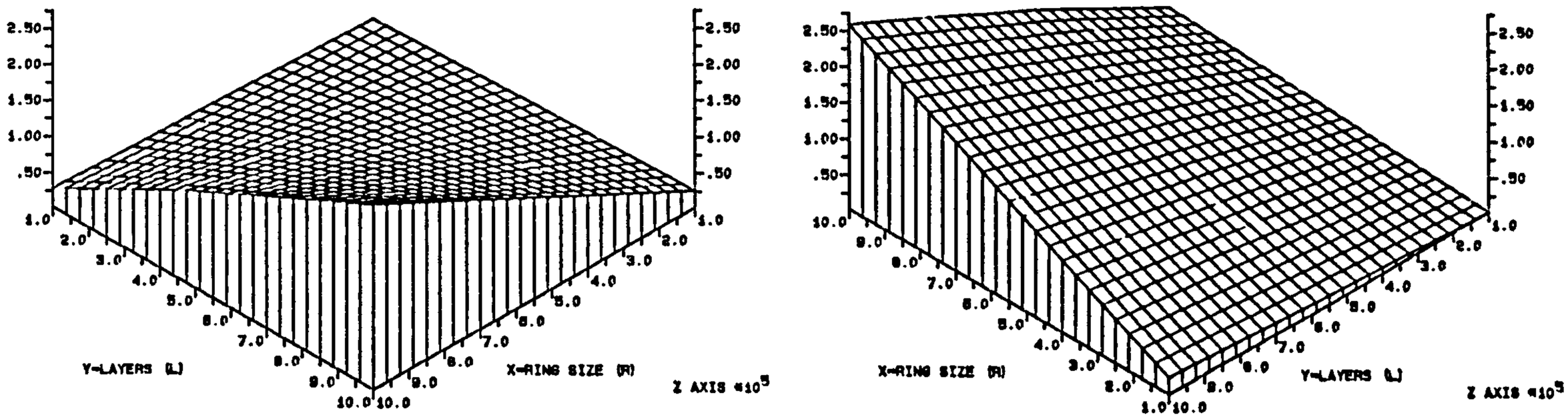


FIG. 5A5.2 HOMOGENEOUS COMMS2 FED AT ALL POINTS WITH DATA OF TYPE 50. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.



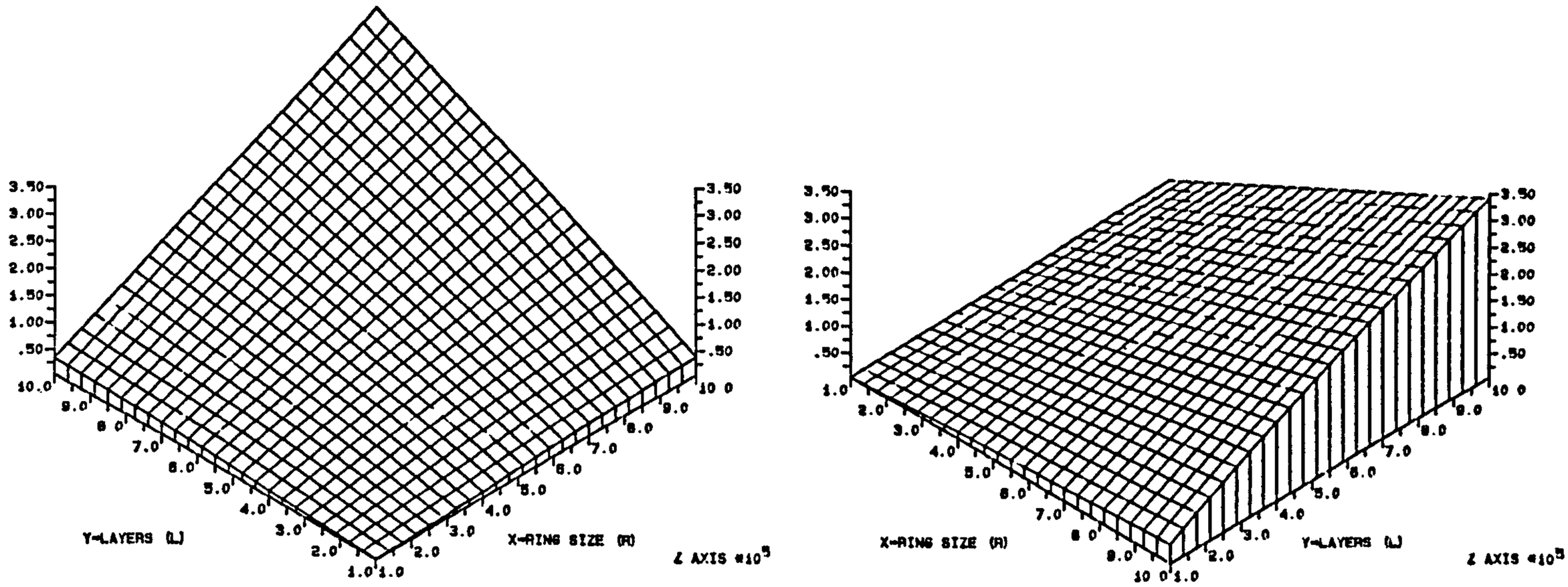
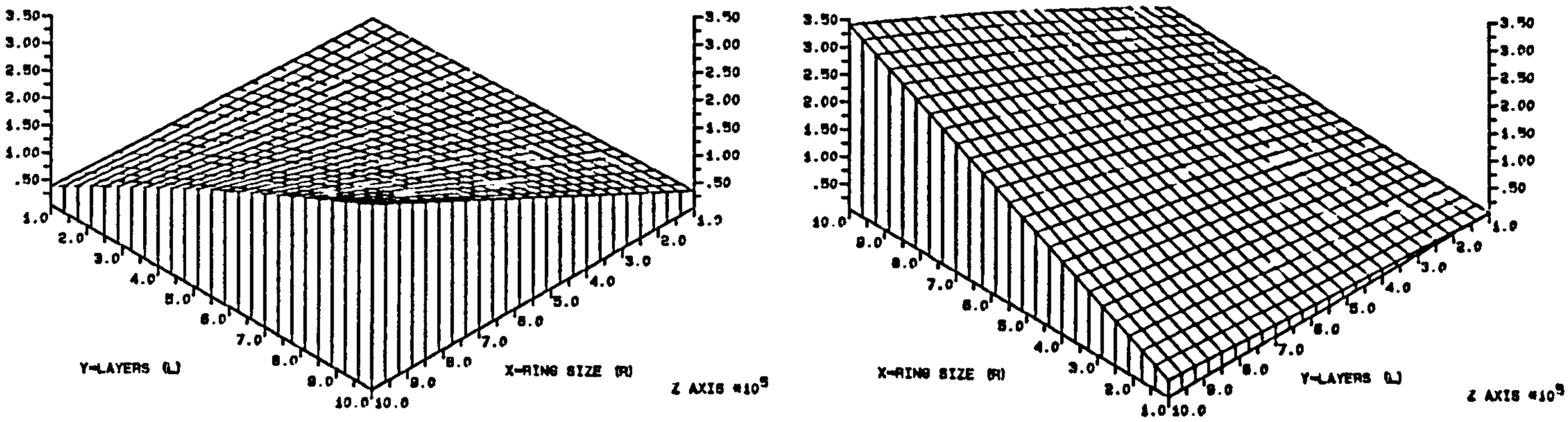


FIG. 5A5.3 HOMOGENEOUS COMMS3 FED AT ALL POINTS WITH DATA OF TYPE 50. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.



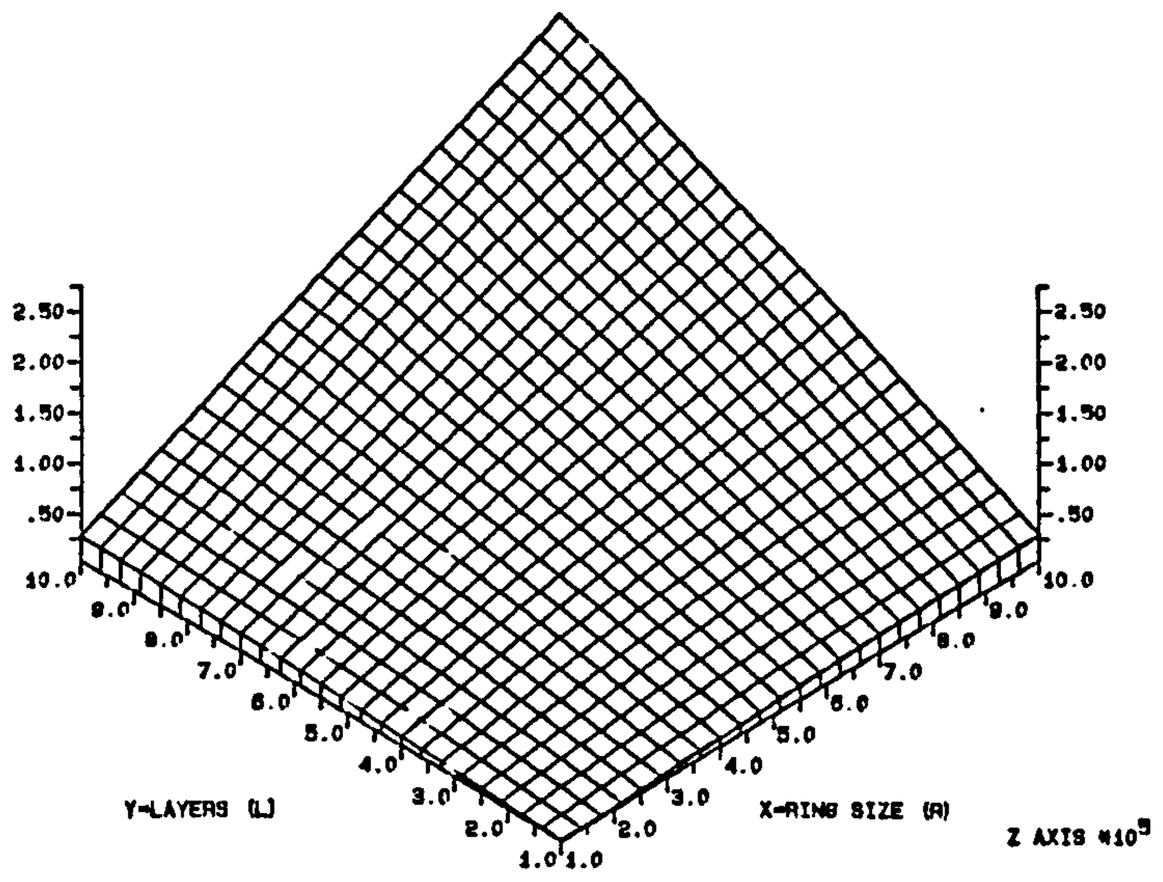
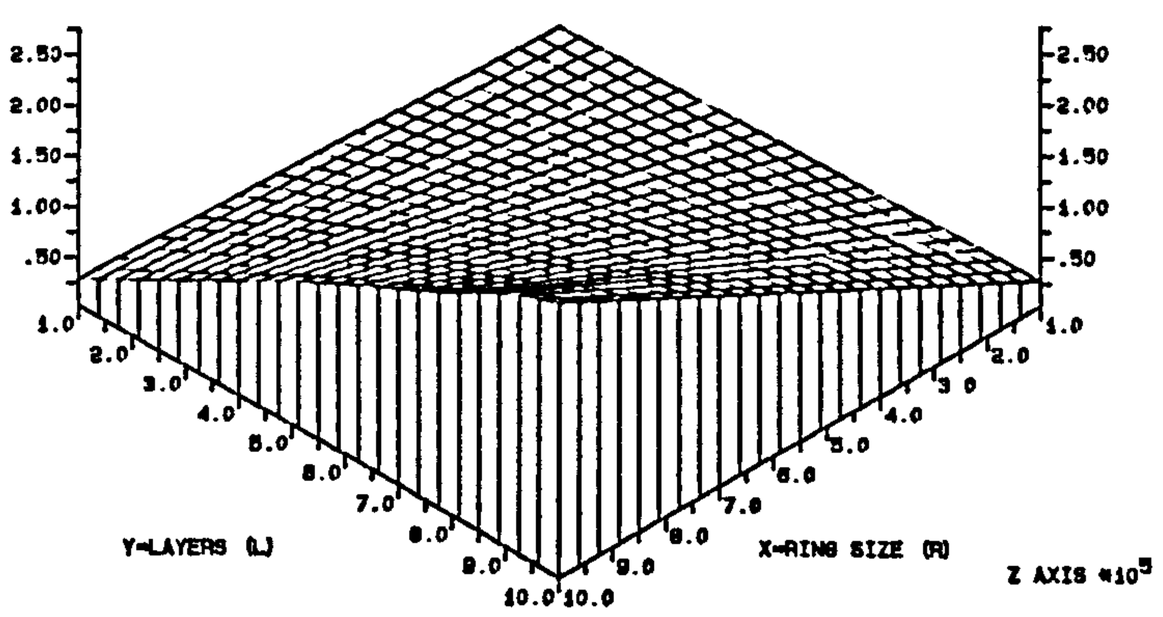
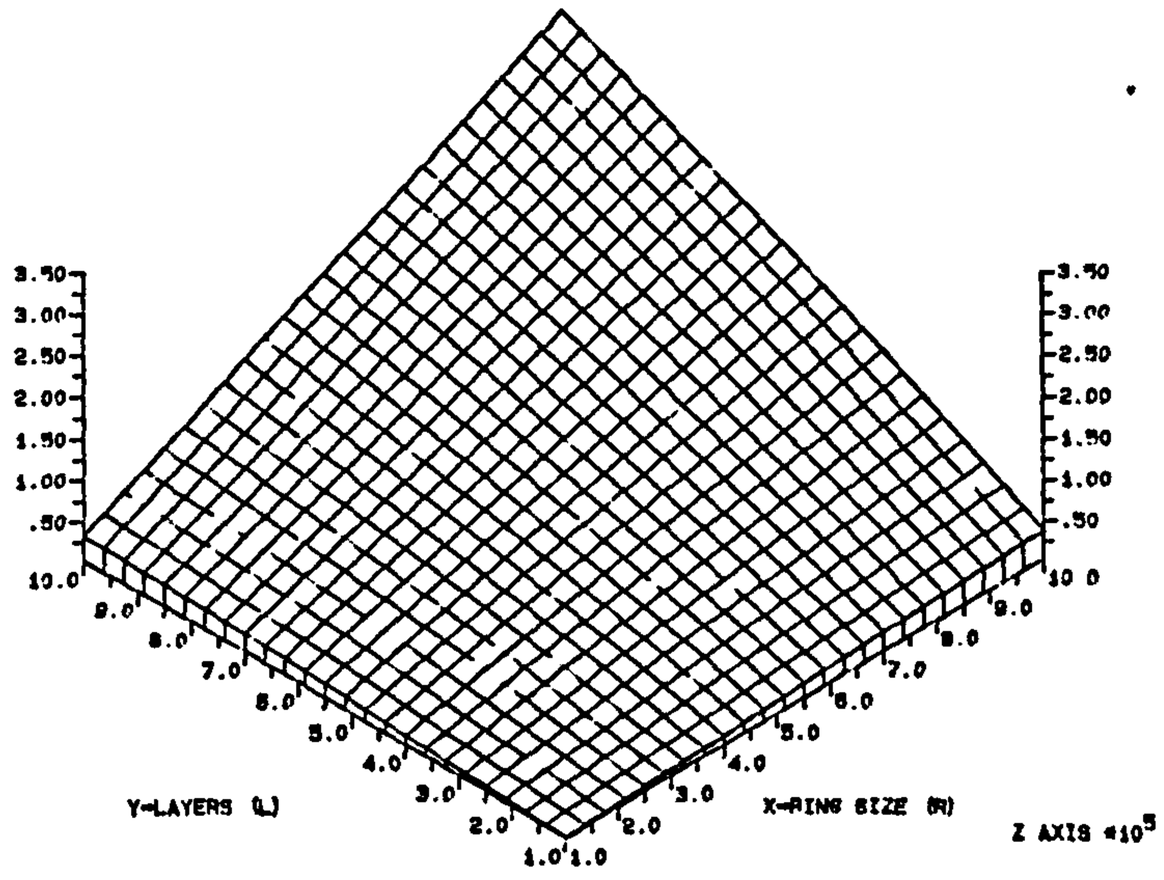


FIG. 5A5.4 HOMOGENEOUS COMMS4 FED AT ALL POINTS WITH DATA OF TYPE 50. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





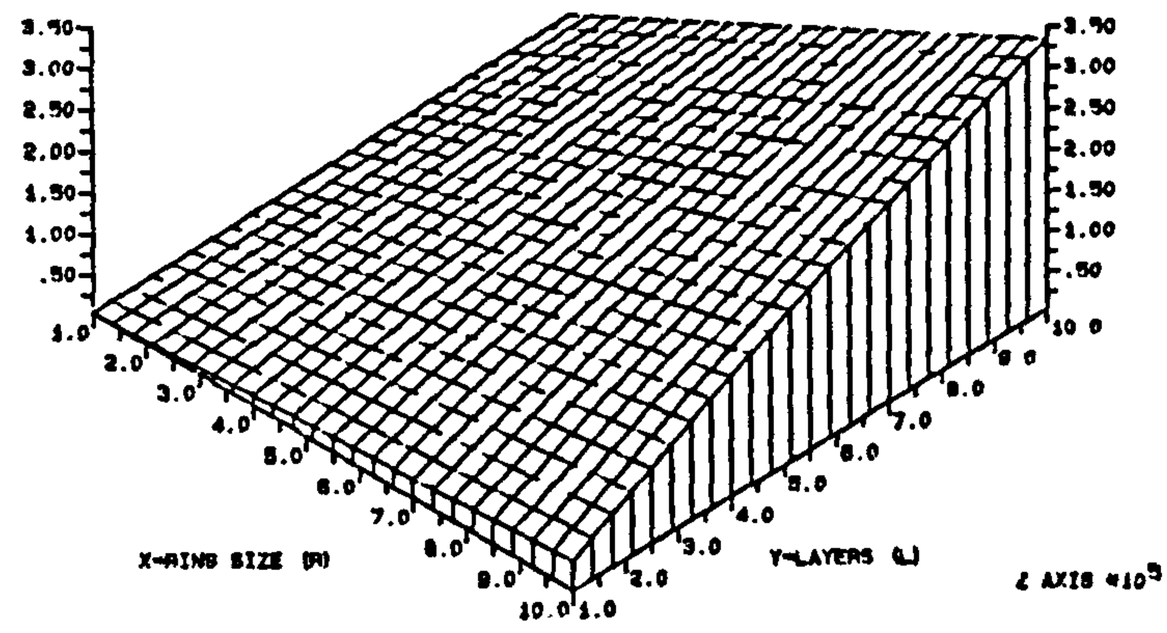
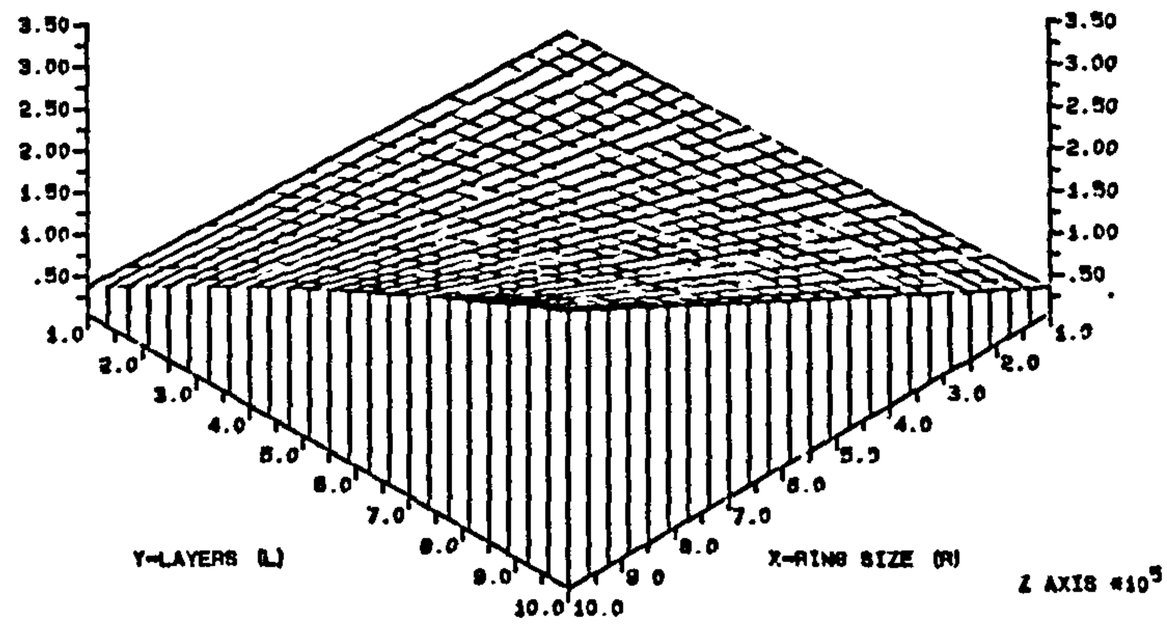
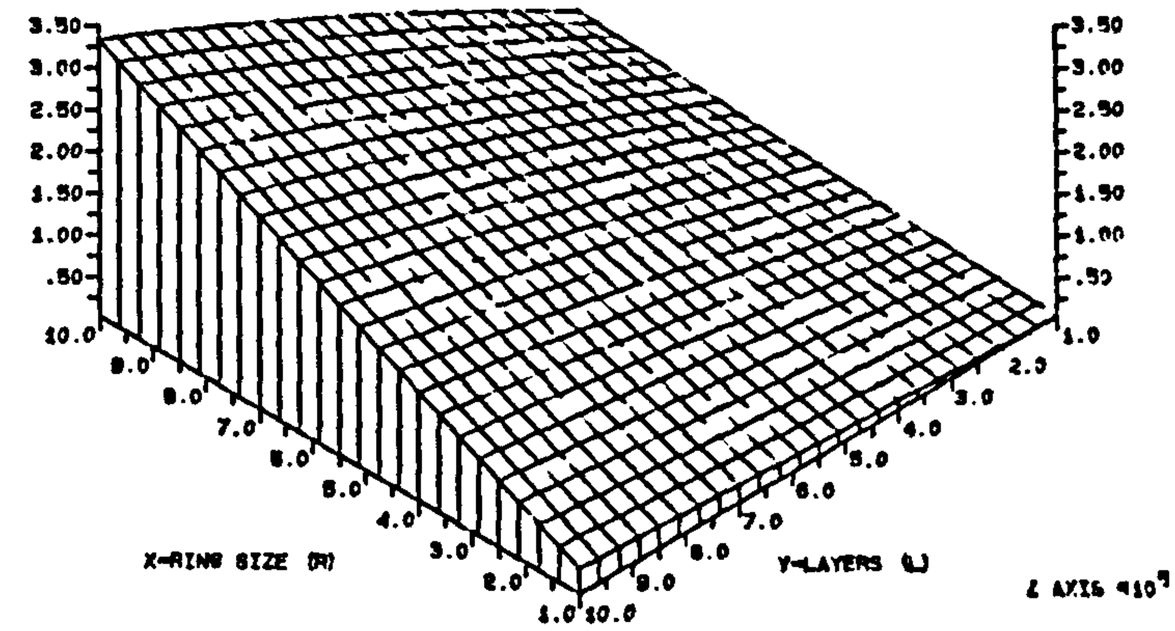
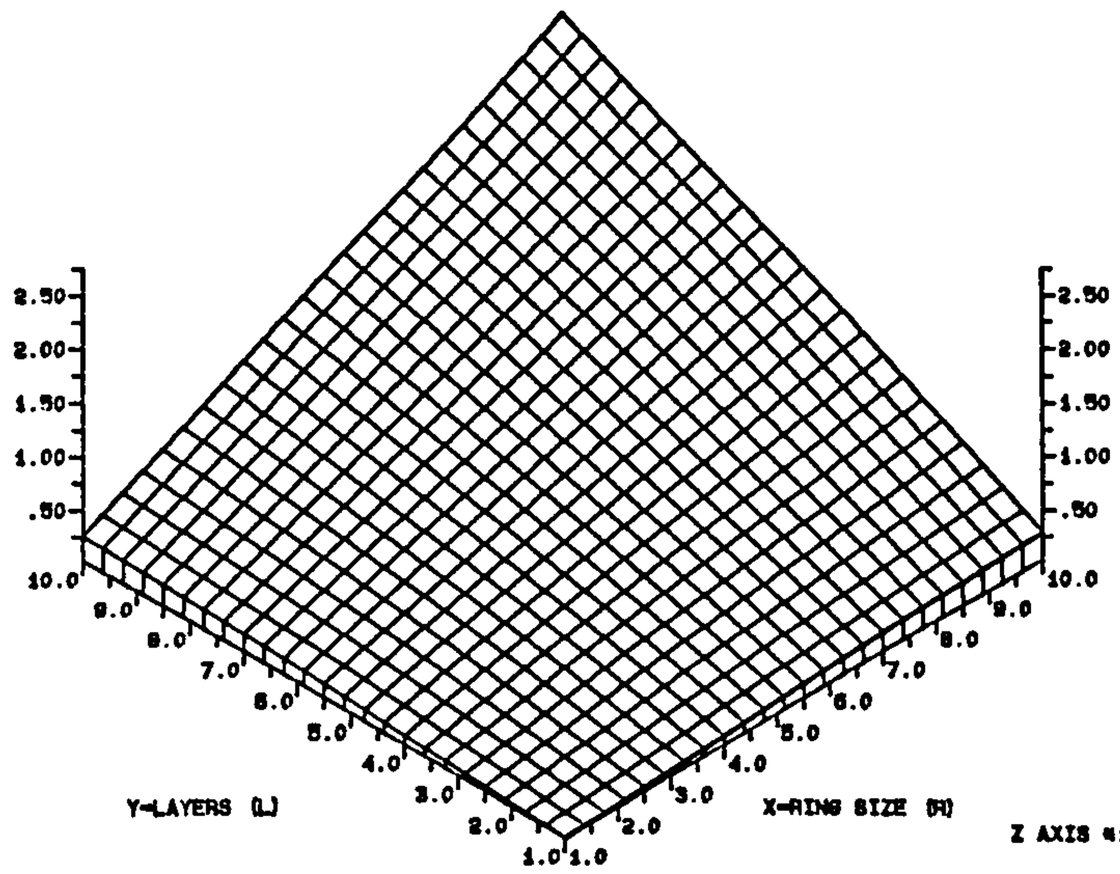


FIG. 5A6.1 HOMOGENEOUS COMMS1 FED AT ALL POINTS WITH DATA OF TYPE R100. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.







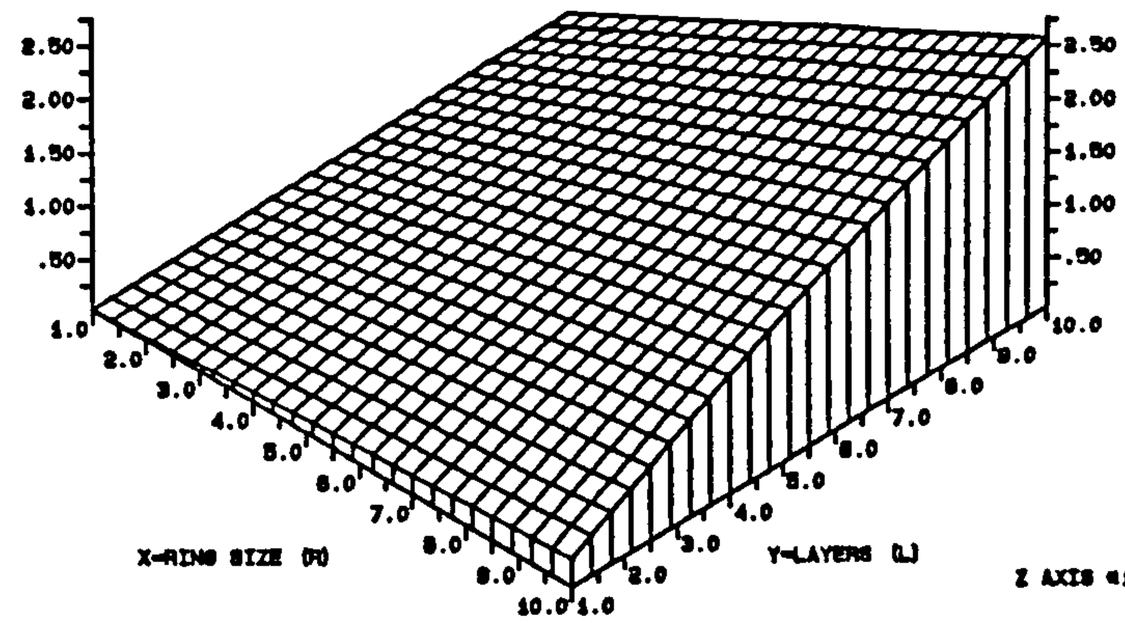
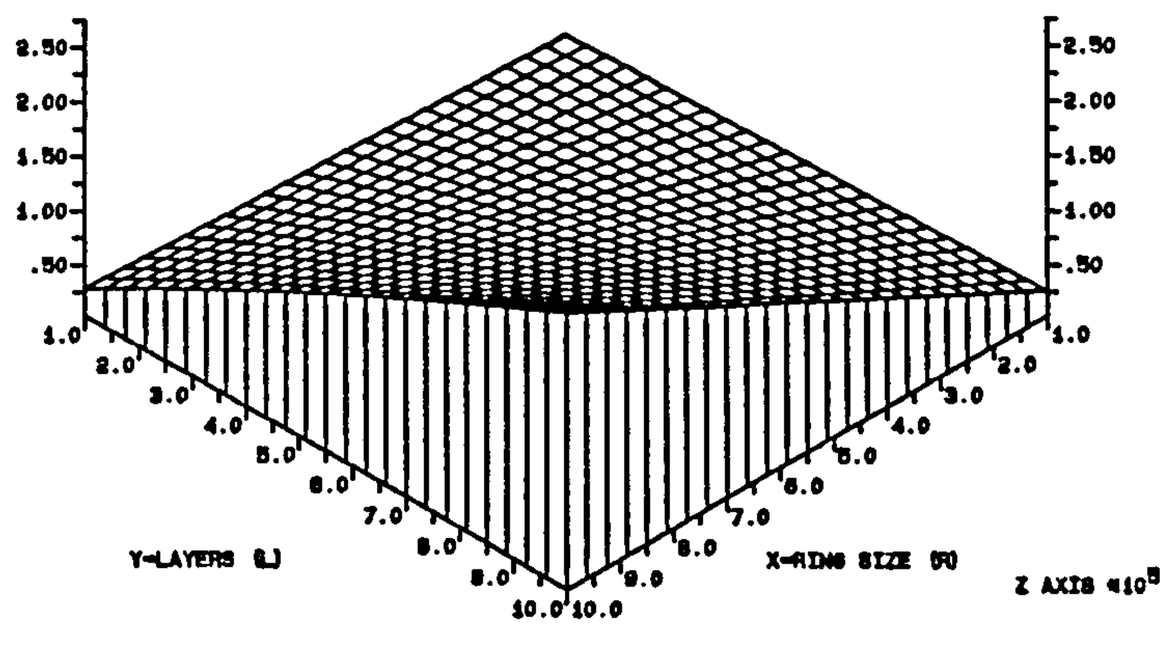
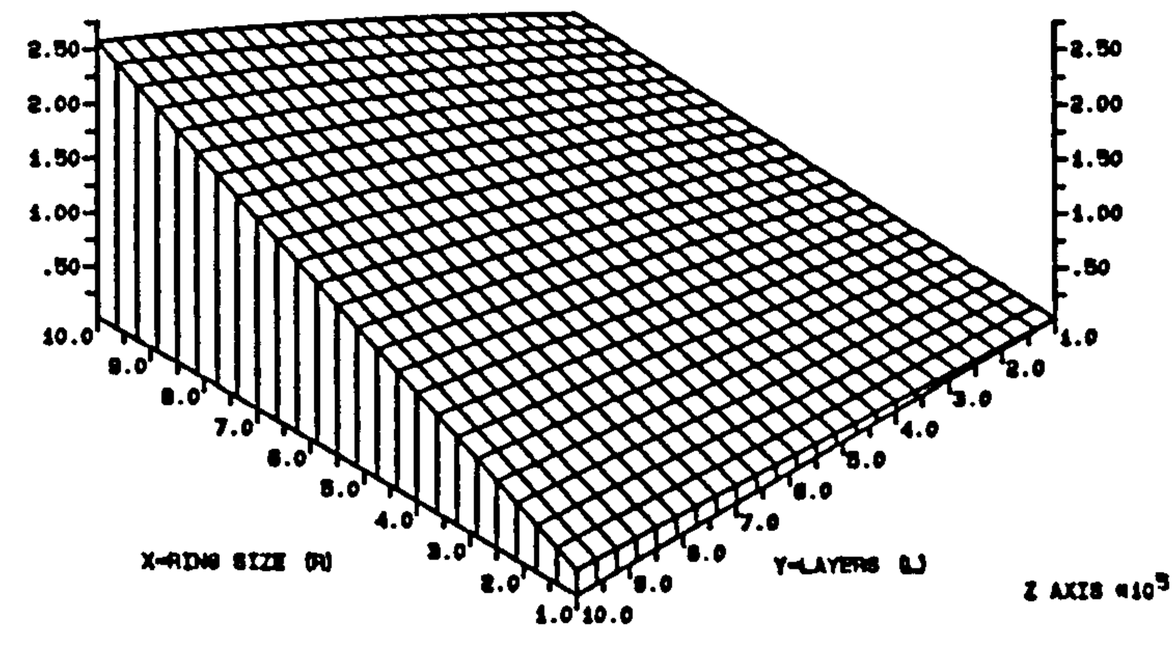


FIG. 5A6.2 HOMOGENEOUS COMMS2 FED AT ALL POINTS WITH DATA OF TYPE R100. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





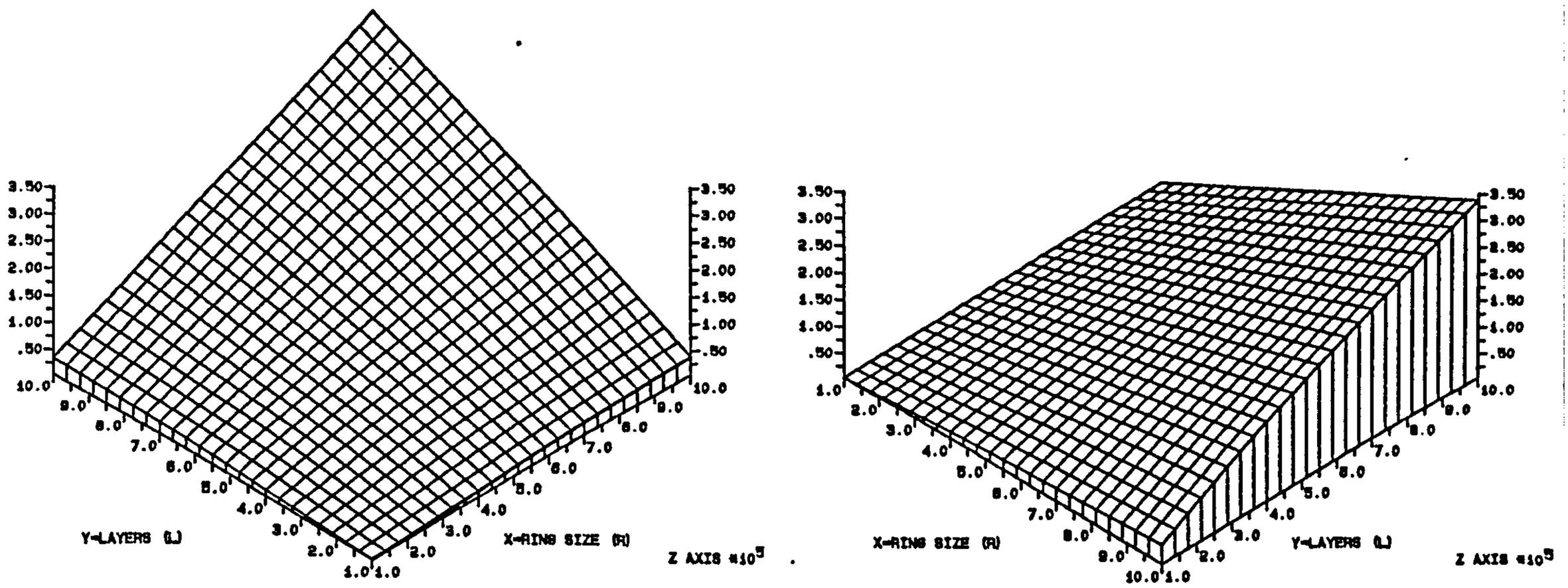
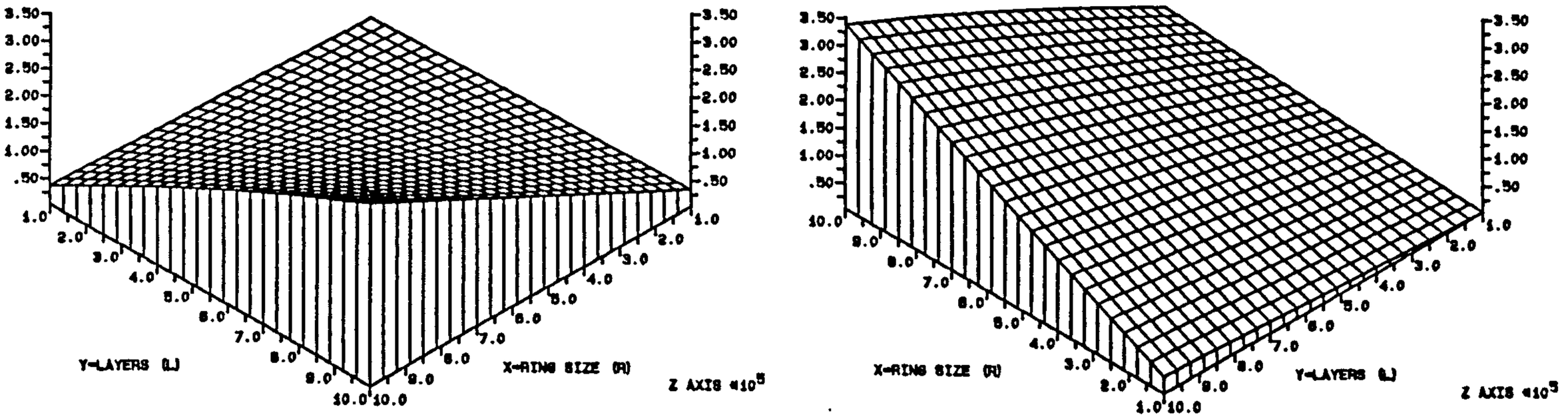


FIG. 5A6.3 HOMOGENEOUS COMMS3 FED AT ALL POINTS WITH DATA OF TYPE R100. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.



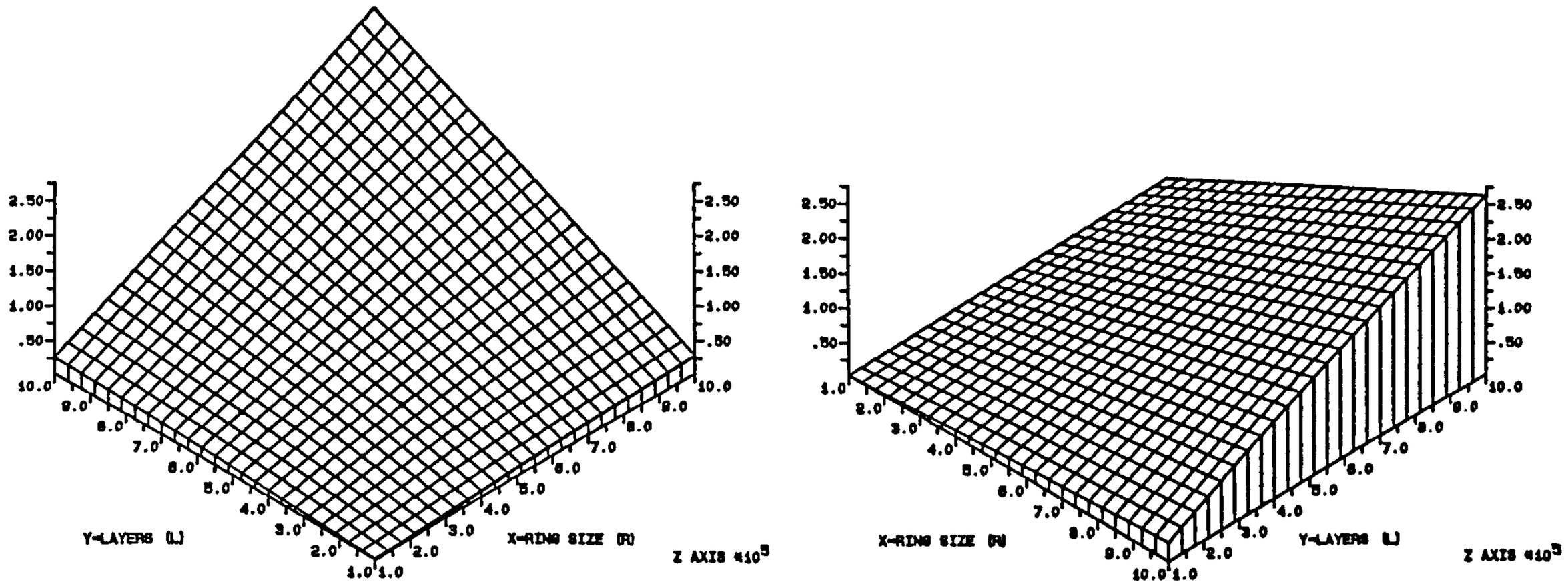
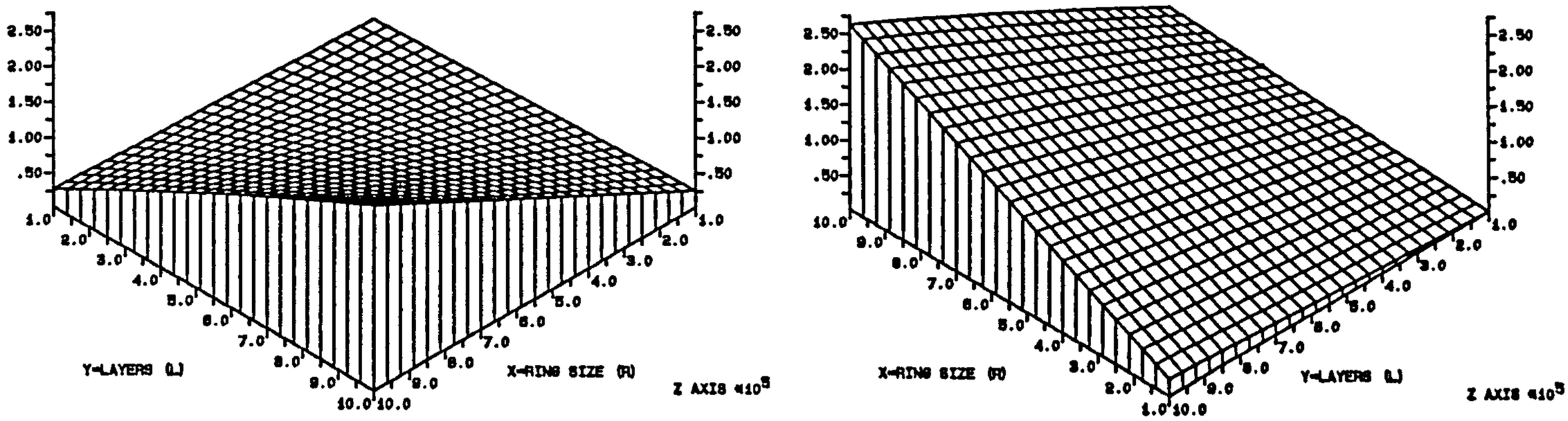


FIG. 5A6.4 HOMOGENEOUS COMMS4 FED AT ALL POINTS WITH DATA OF TYPE R100. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.



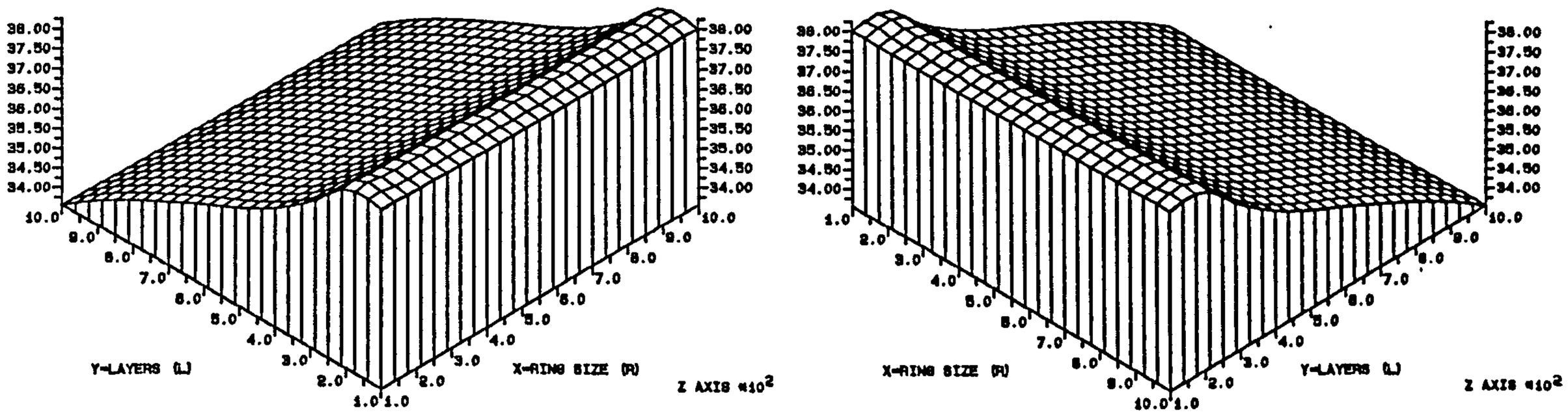
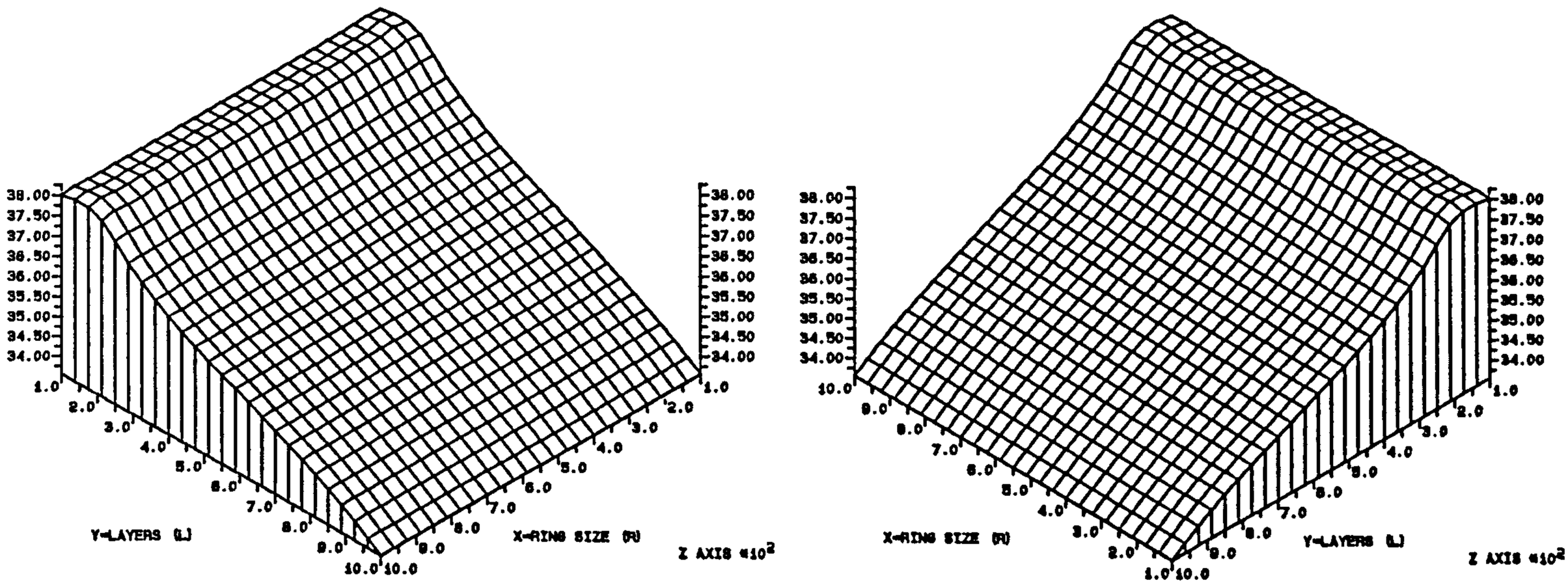


FIG. 5A7.1 HOMOGENEOUS COMMS1 FED AT ALL POINTS WITH DATA OF TYPE 50. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



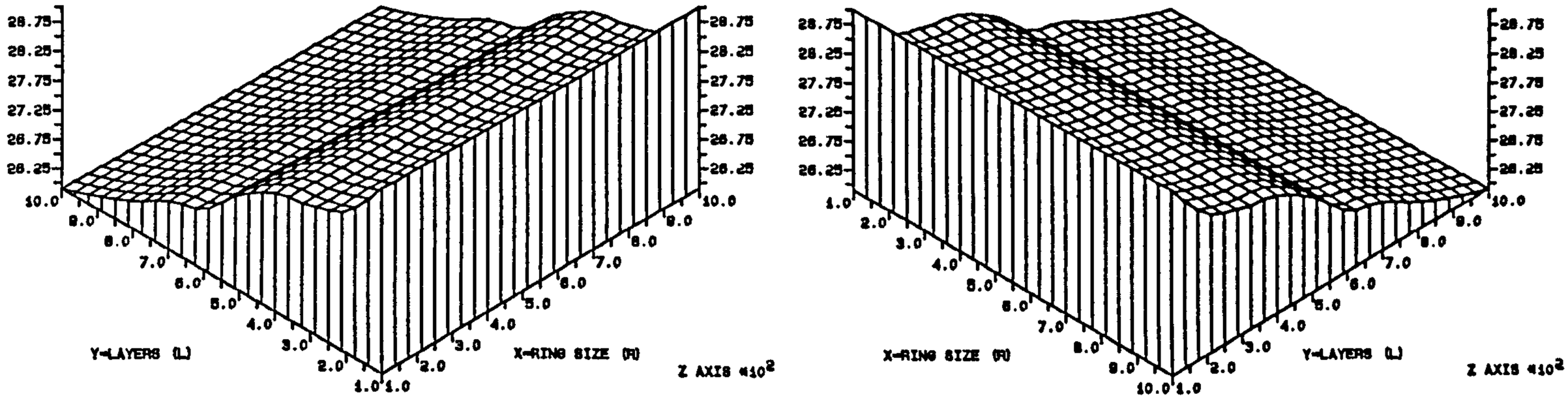
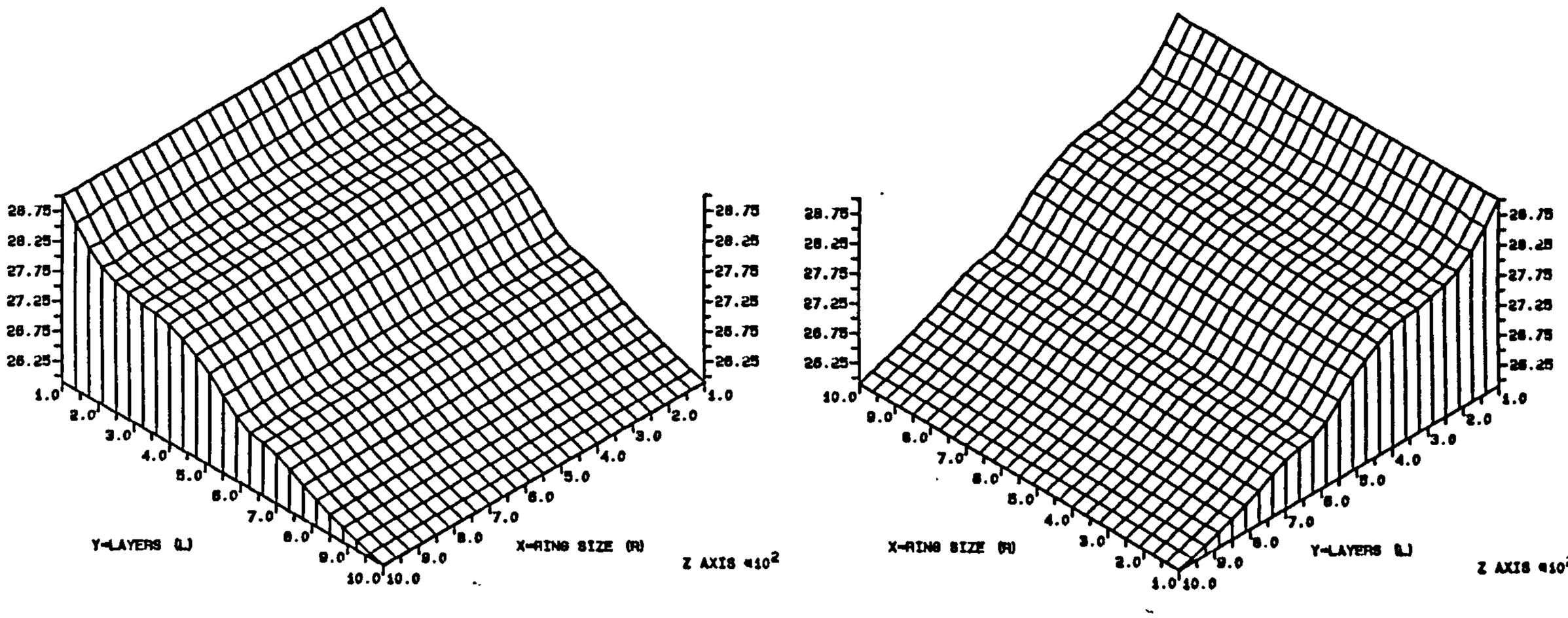
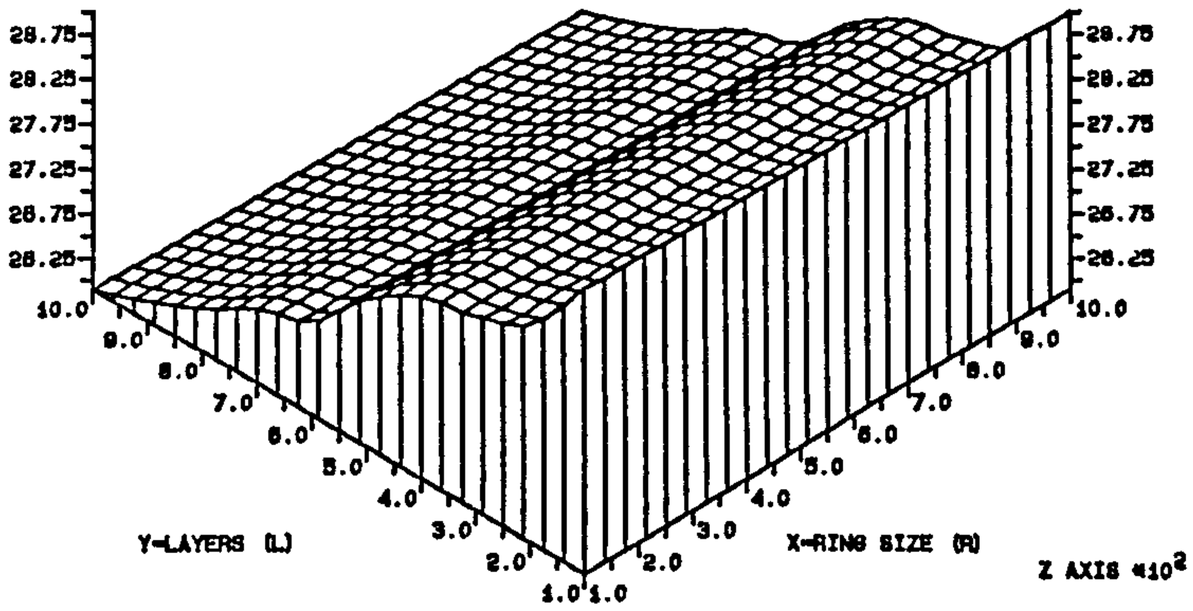


FIG. 5A7.2 HOMOGENEOUS COMMS2 FED AT ALL POINTS WITH DATA OF TYPE 50. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





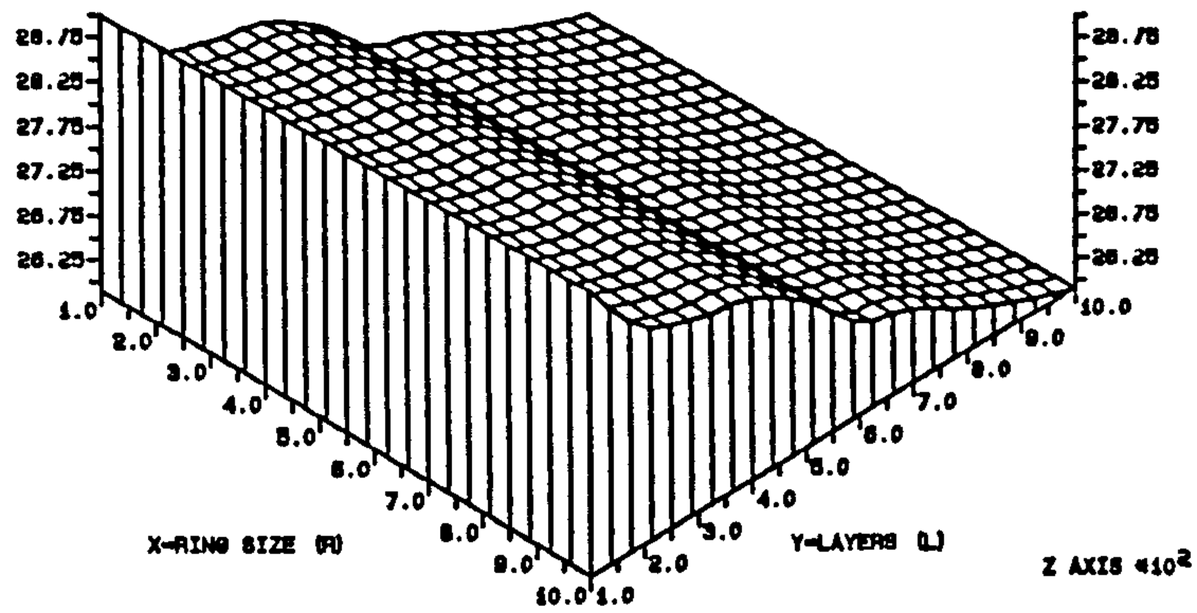
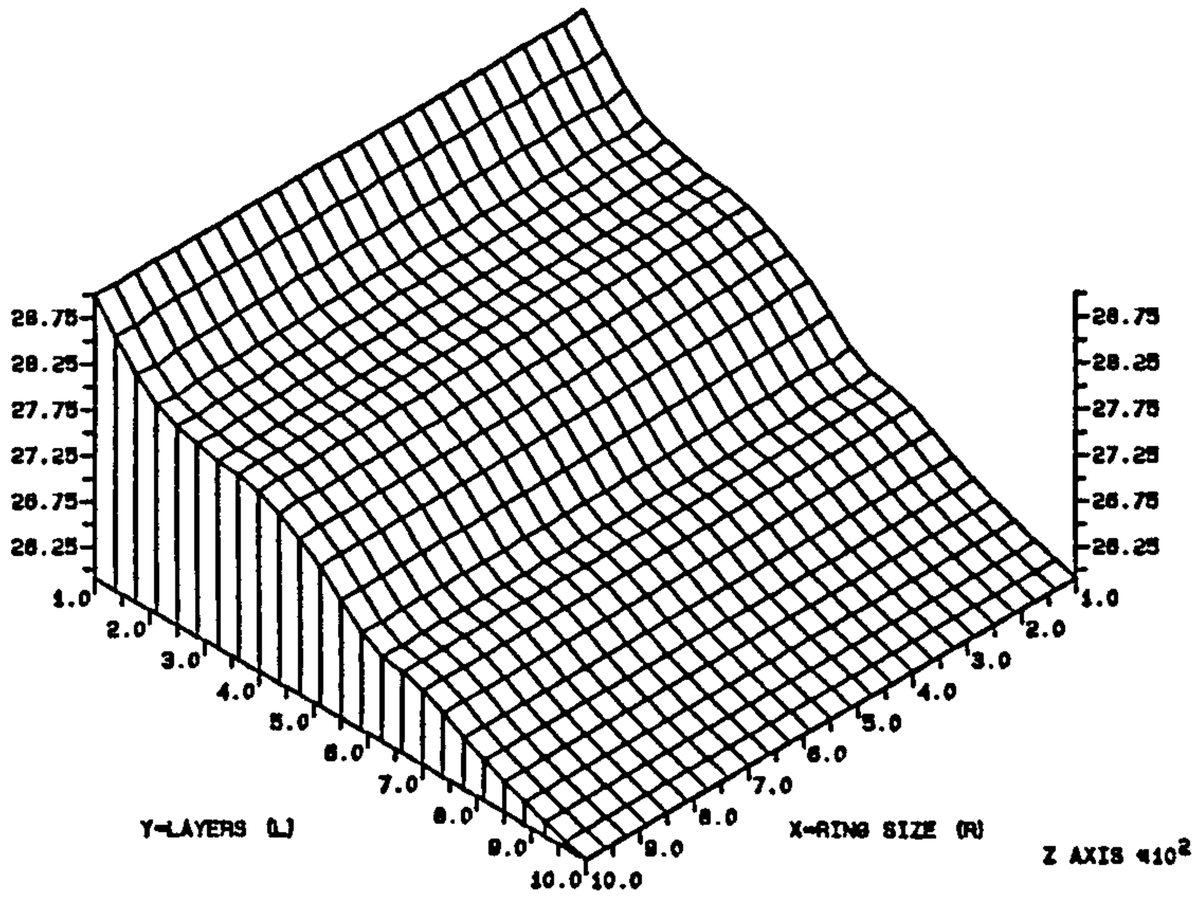
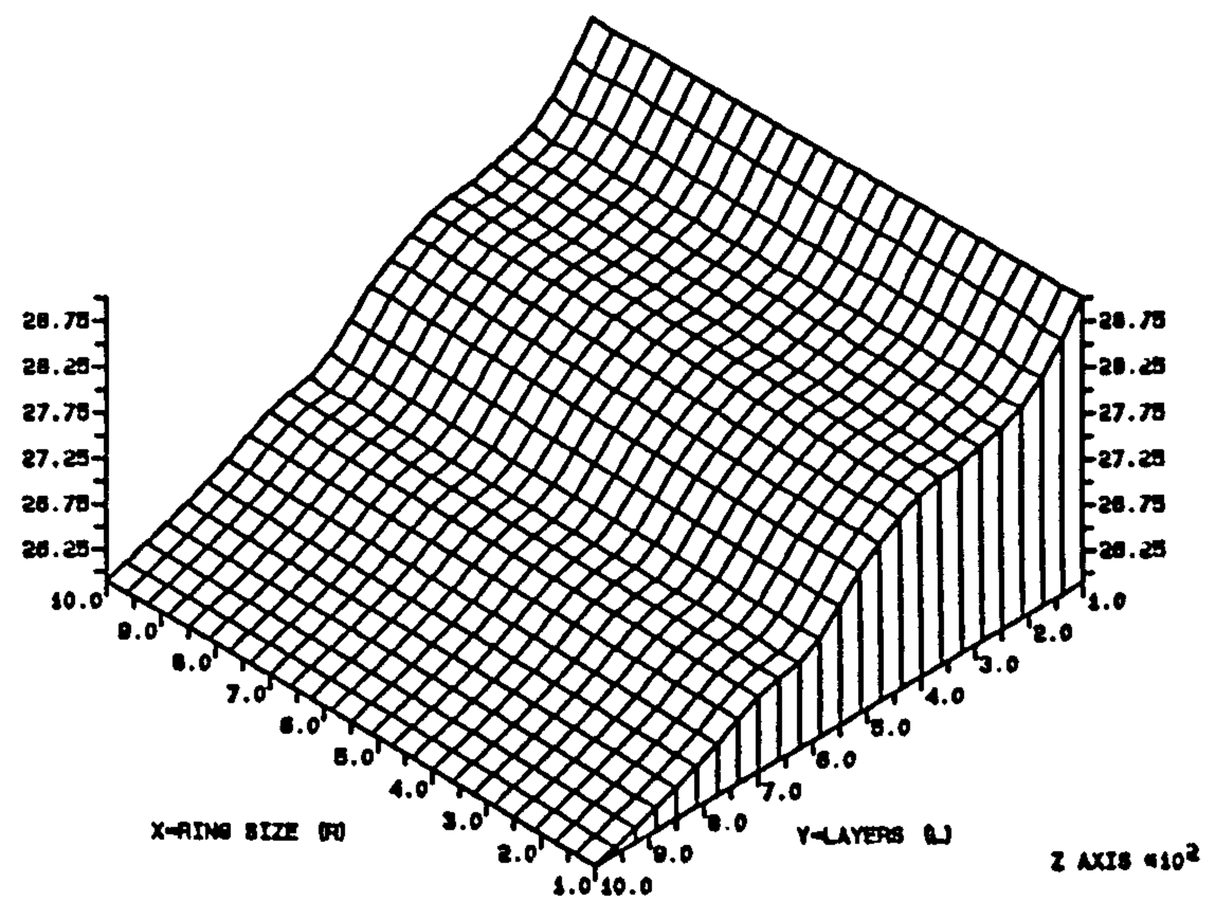


FIG. 5A7.2 HOMOGENEOUS COMMS2 FED AT ALL POINTS WITH DATA . OF TYPE 50. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





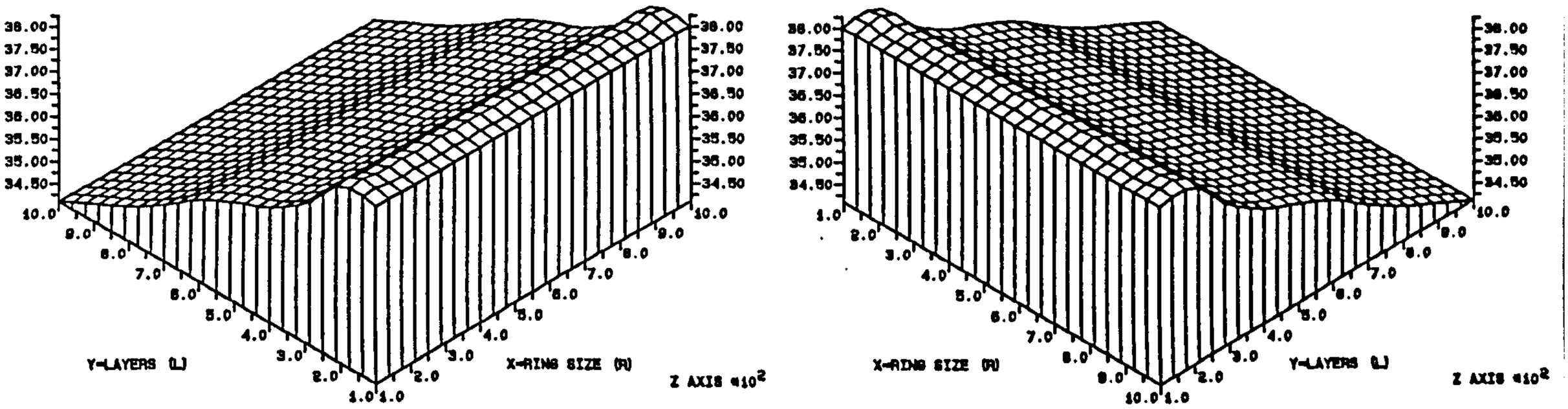
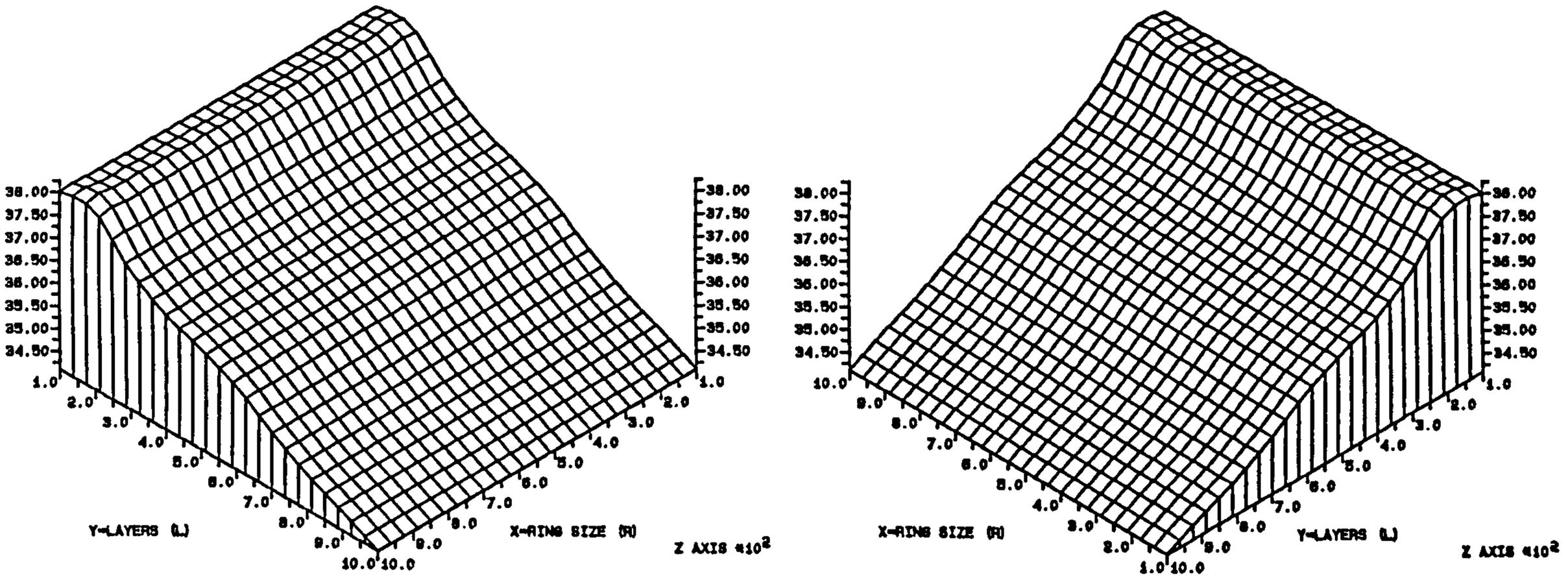


FIG. 5A7.3 HOMOGENEOUS COMMS3 FED AT ALL POINTS WITH DATA OF TYPE 50. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



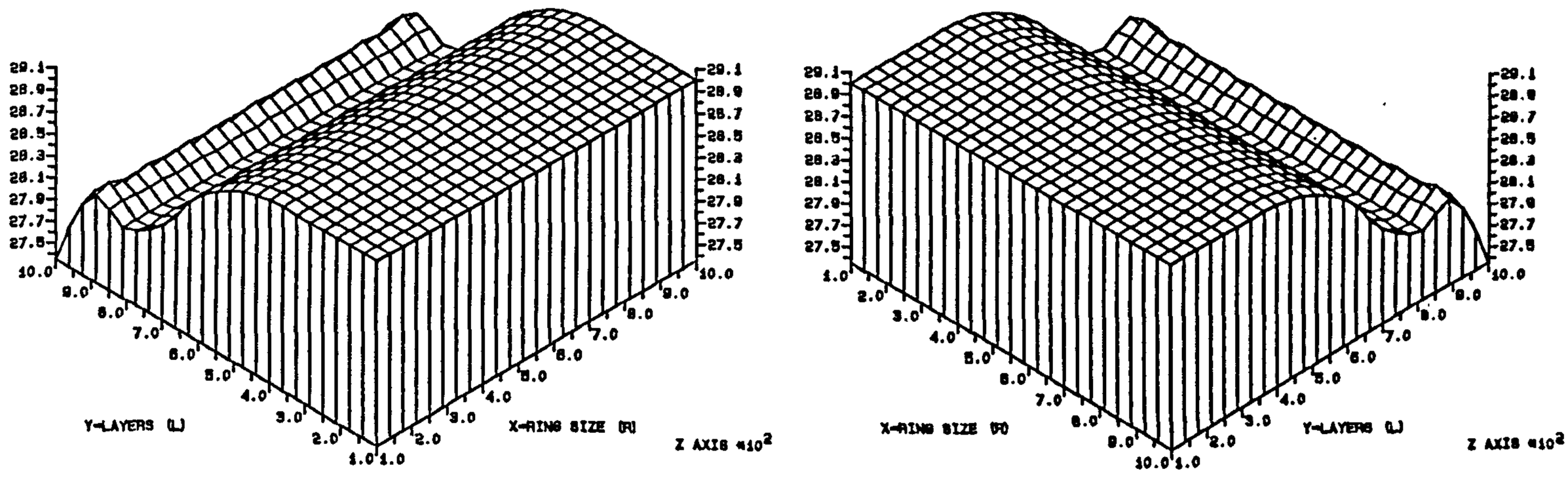
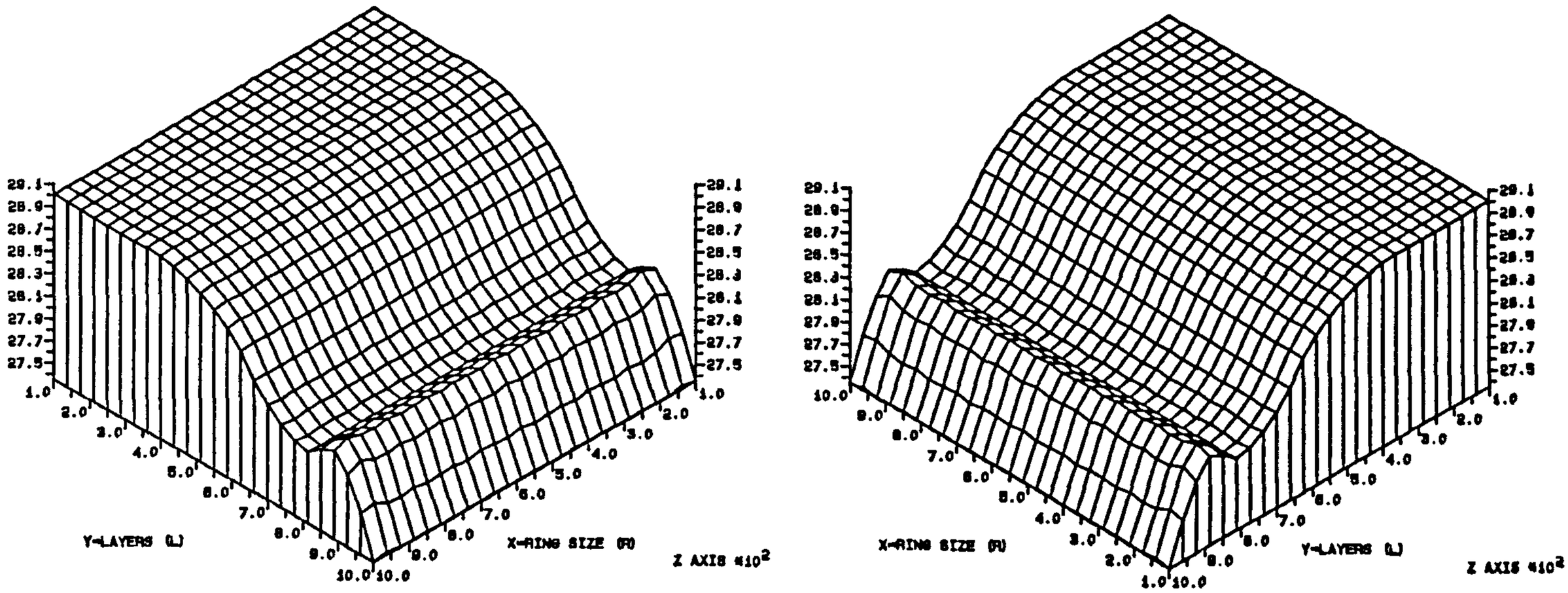


FIG. 5A7.4 HOMOGENEOUS COMMS4 FED AT ALL POINTS WITH DATA OF TYPE 50. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



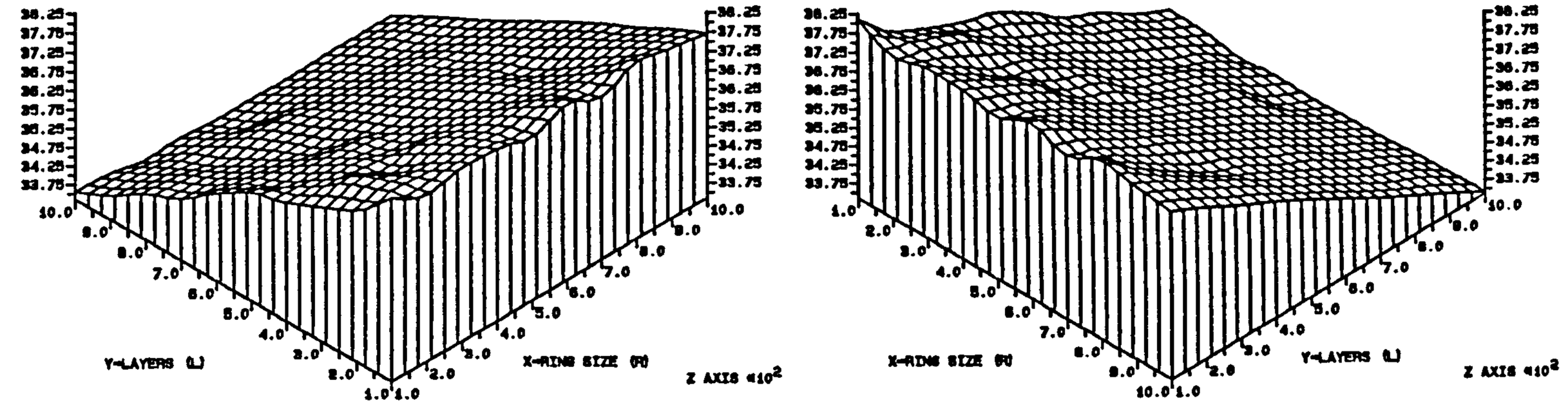
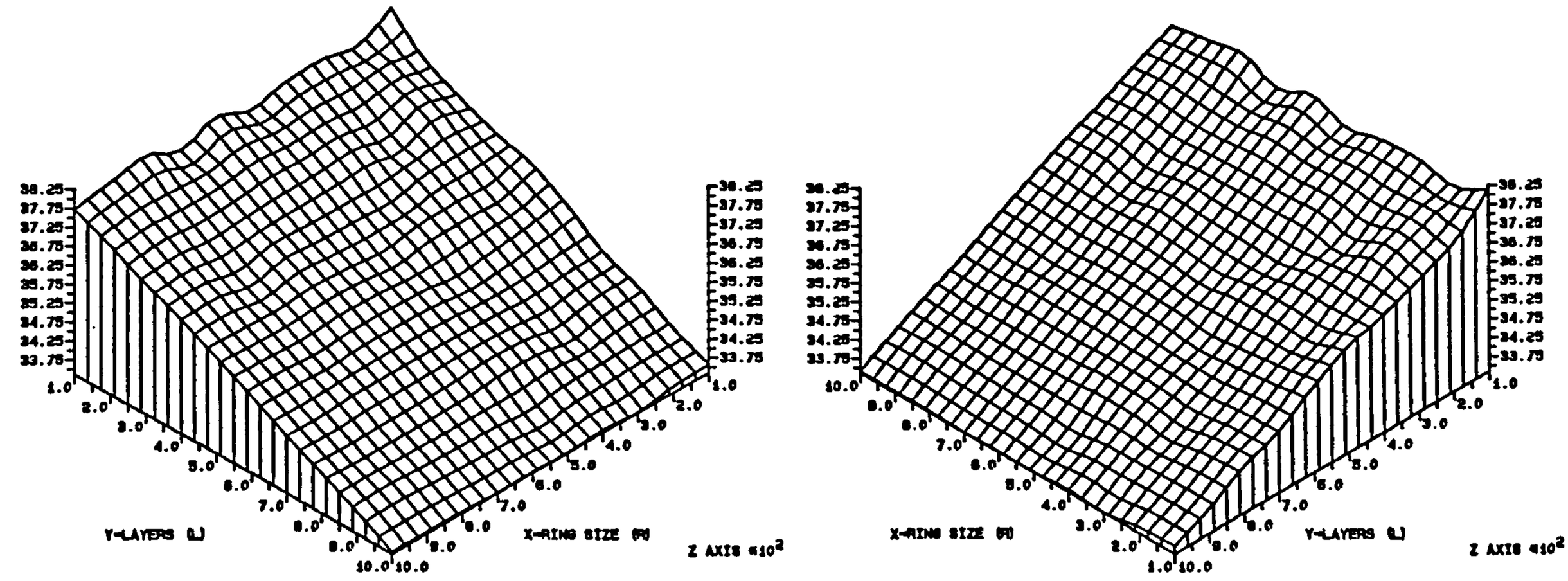


FIG. 5A8.1 HOMOGENEOUS COMMS1 FED AT ALL POINTS WITH DATA OF TYPE R100. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



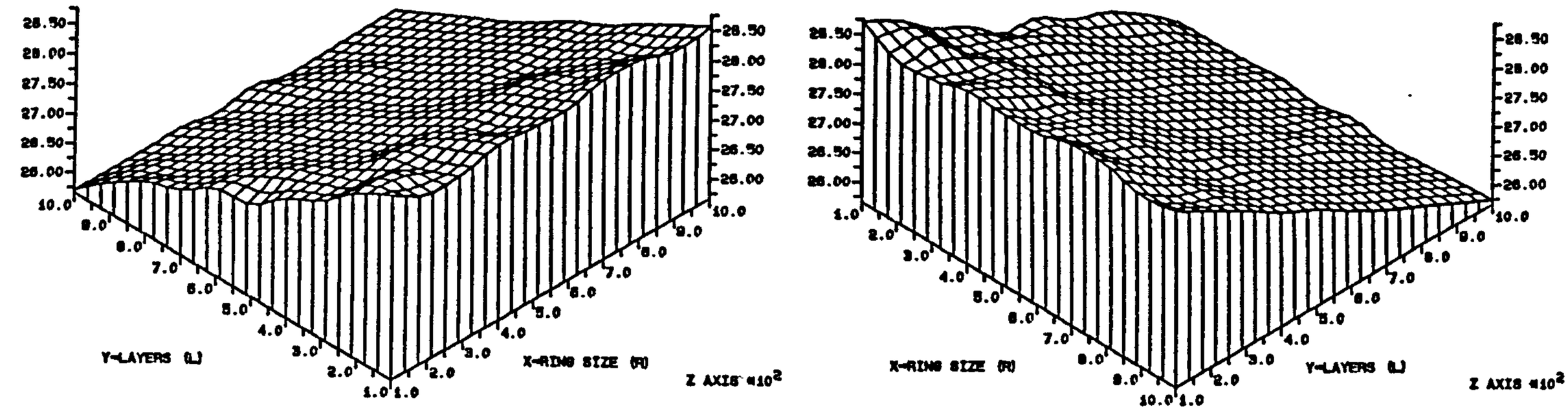
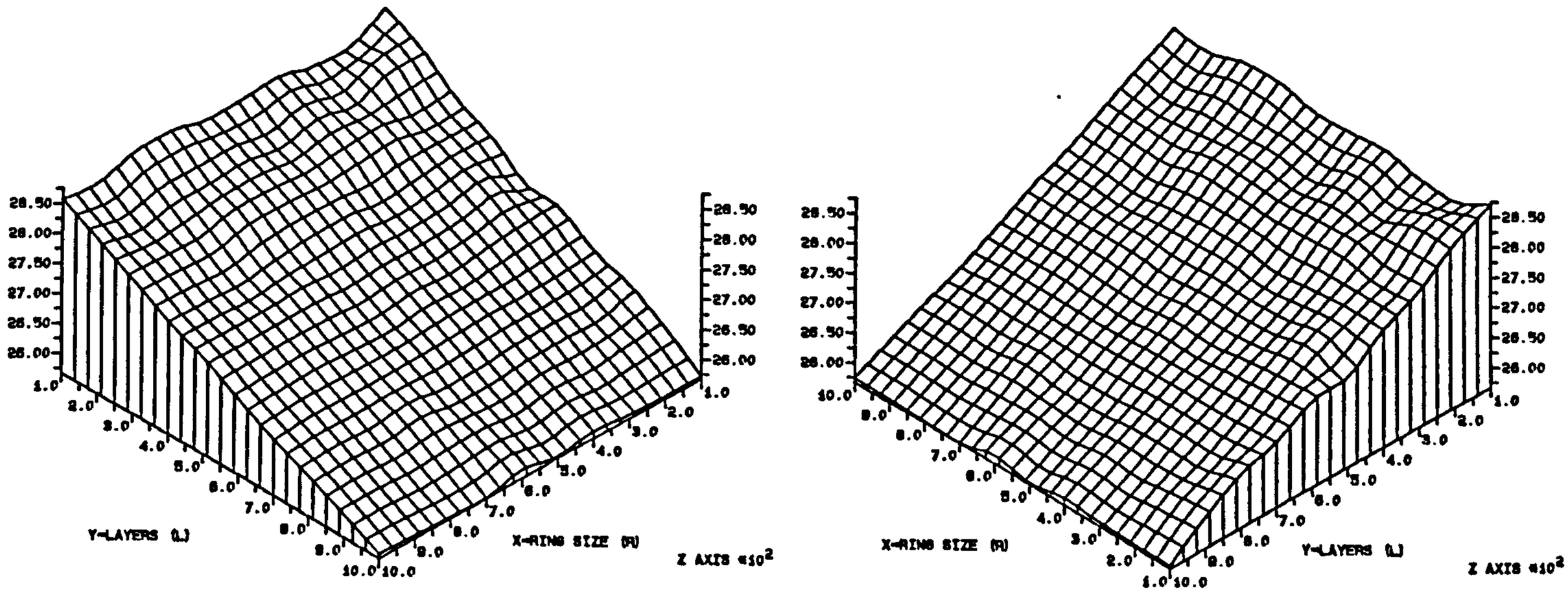


FIG. 5A8.2 HOMOGENEOUS COMMS2 FED AT ALL POINTS WITH DATA OF TYPE R100. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



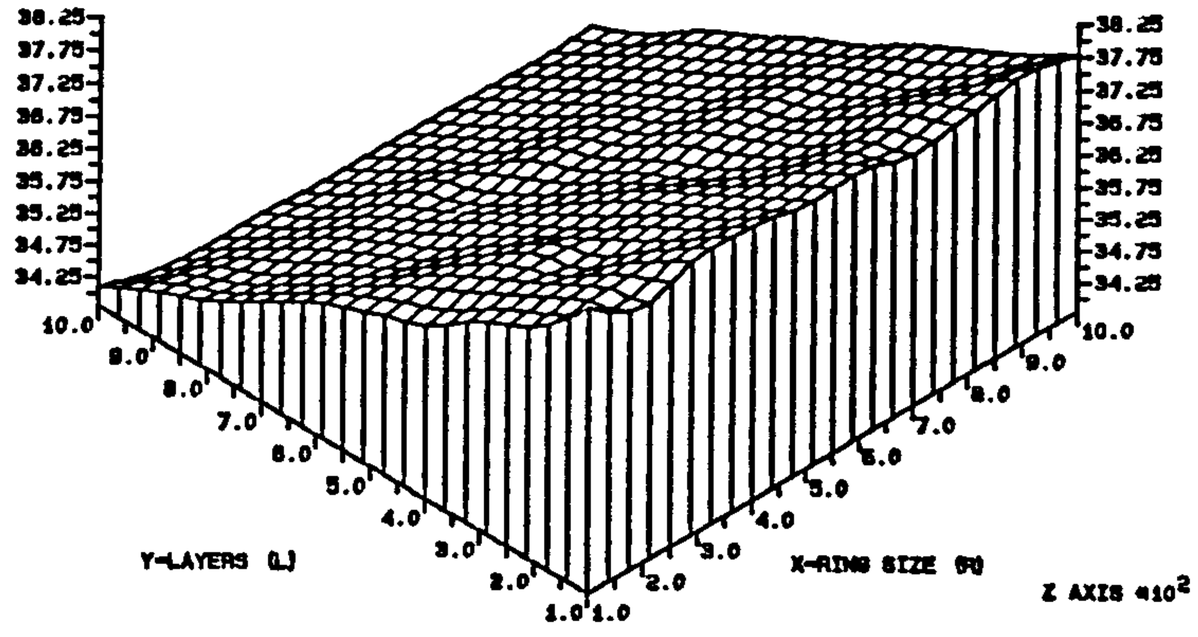
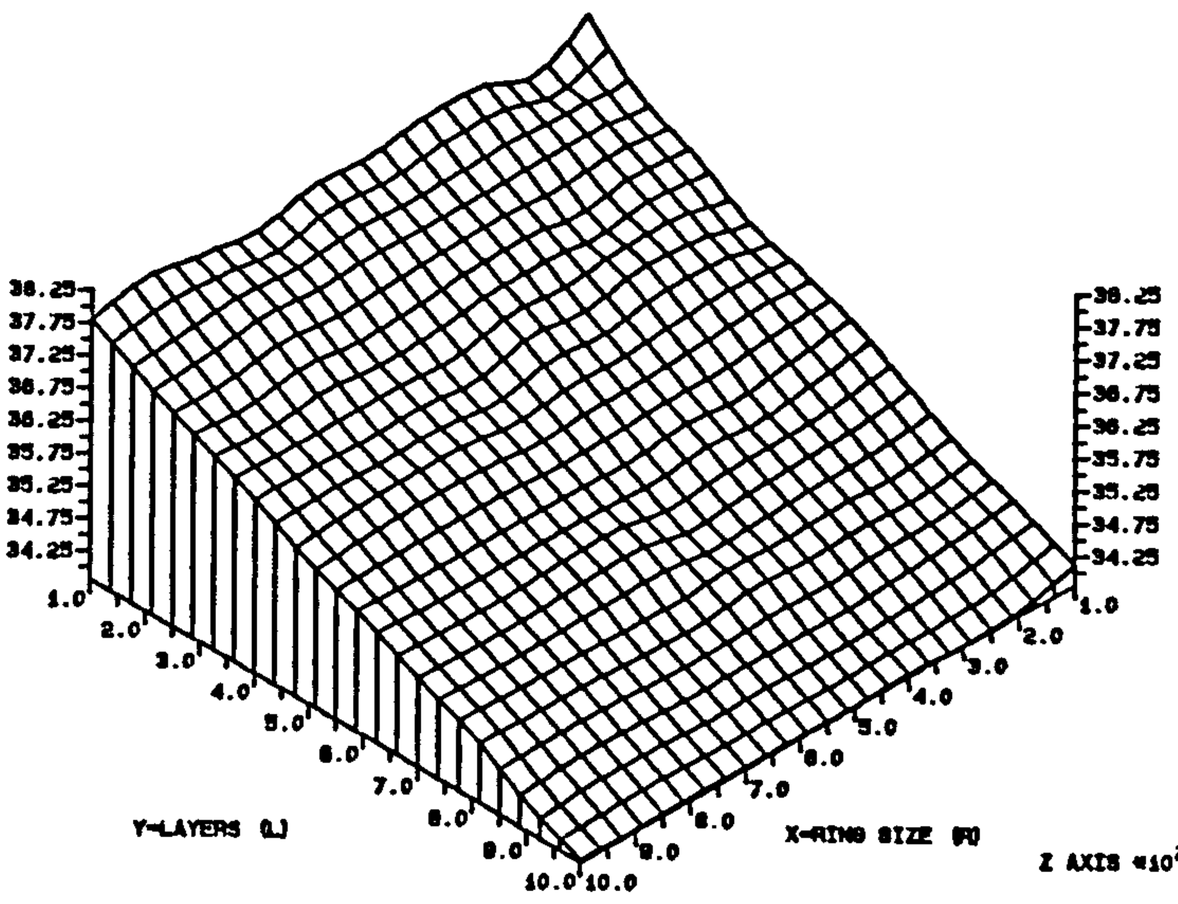


FIG. 5A8.3 HOMOGENEOUS COMMS3 FED AT ALL POINTS WITH DATA OF TYPE R100. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



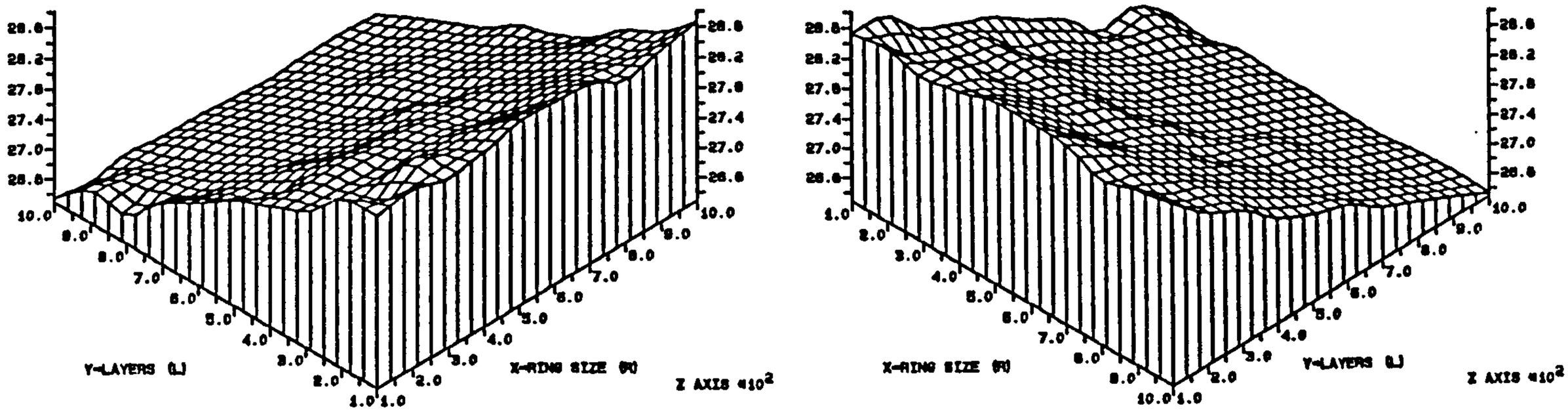
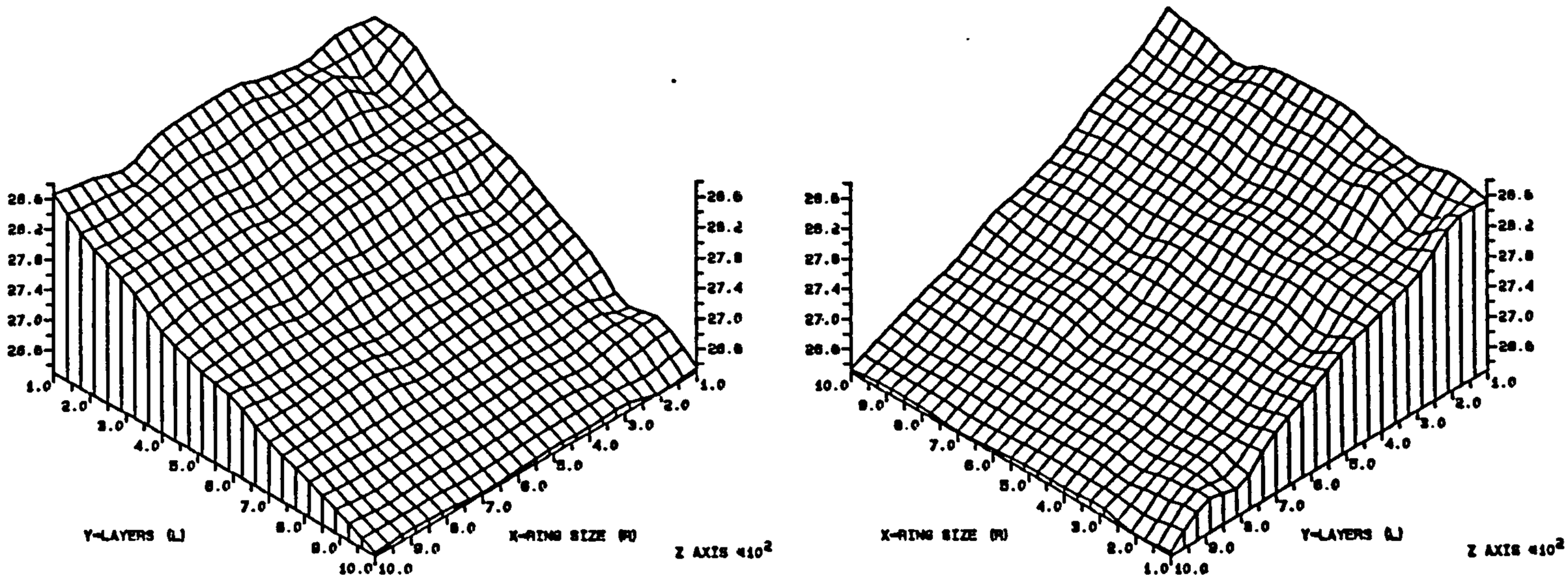


FIG. 5A8.4 HOMOGENEOUS COMMS4 FED AT ALL POINTS WITH DATA OF TYPE R100. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



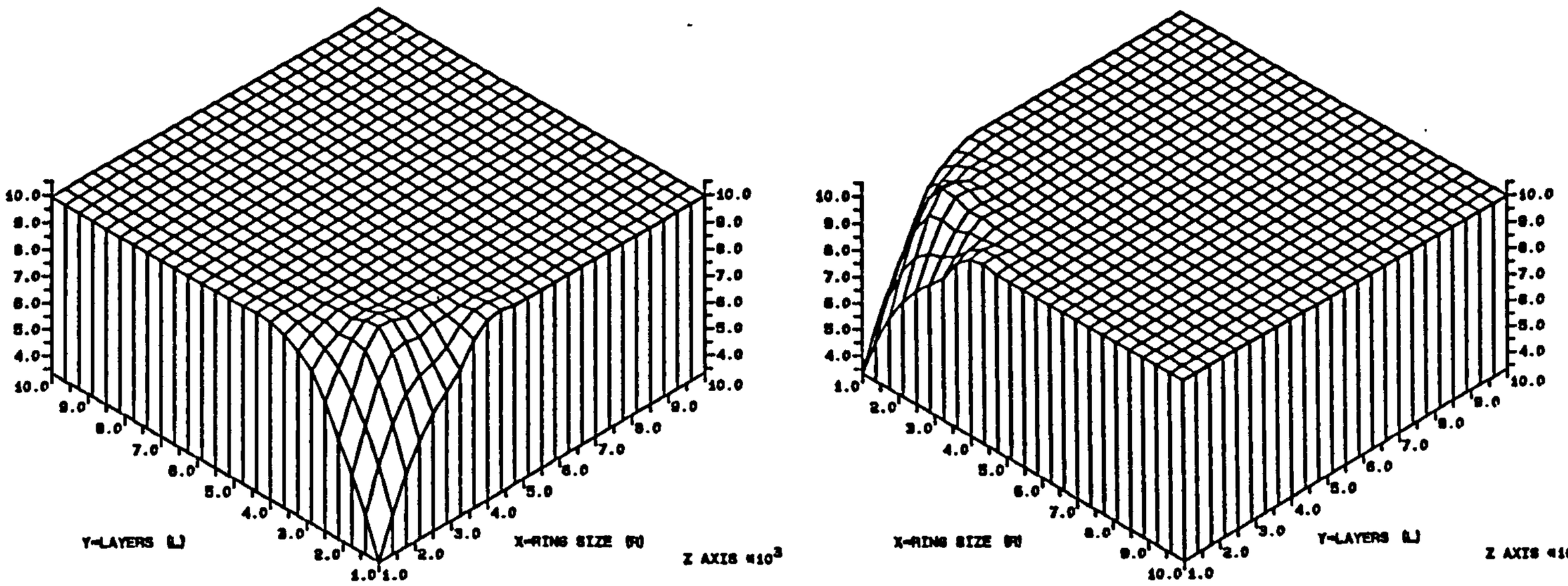
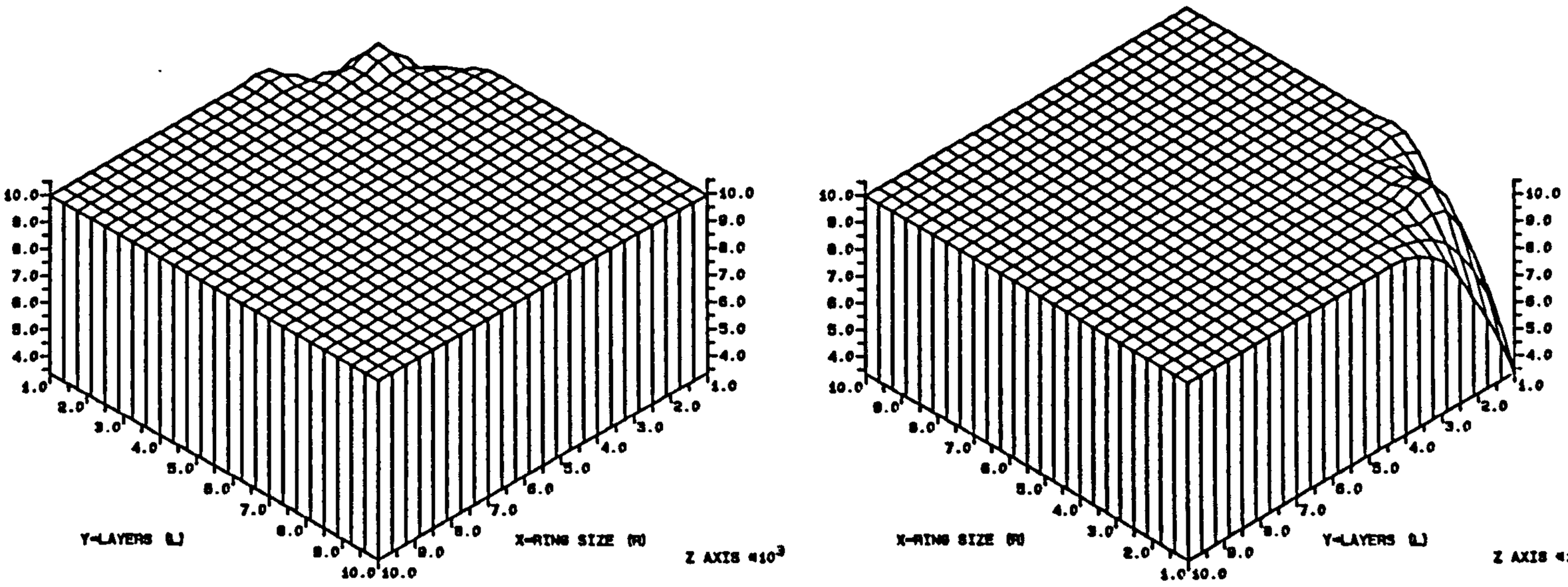
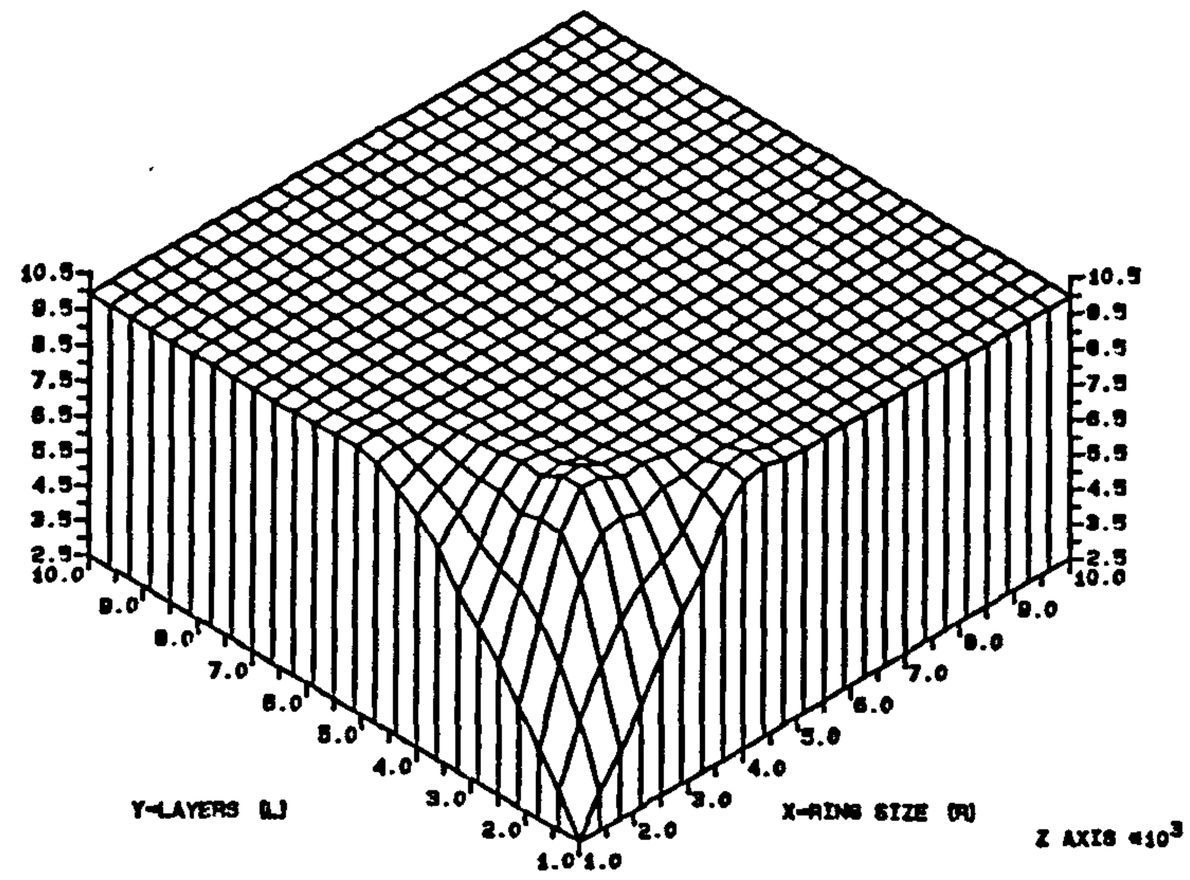


FIG. 5A9.1 HOMOGENEOUS COMMS1 FED AT ONE POINT WITH DATA OF TYPE 10. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





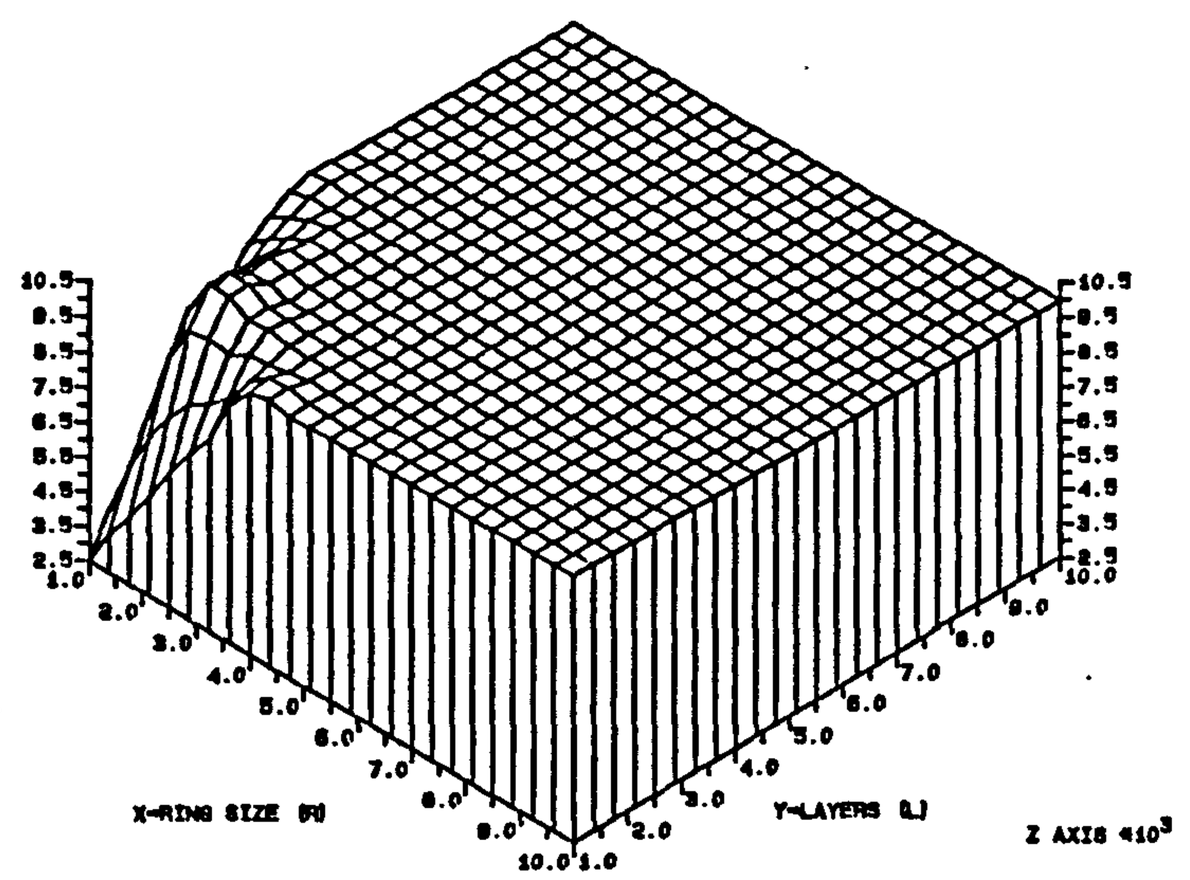
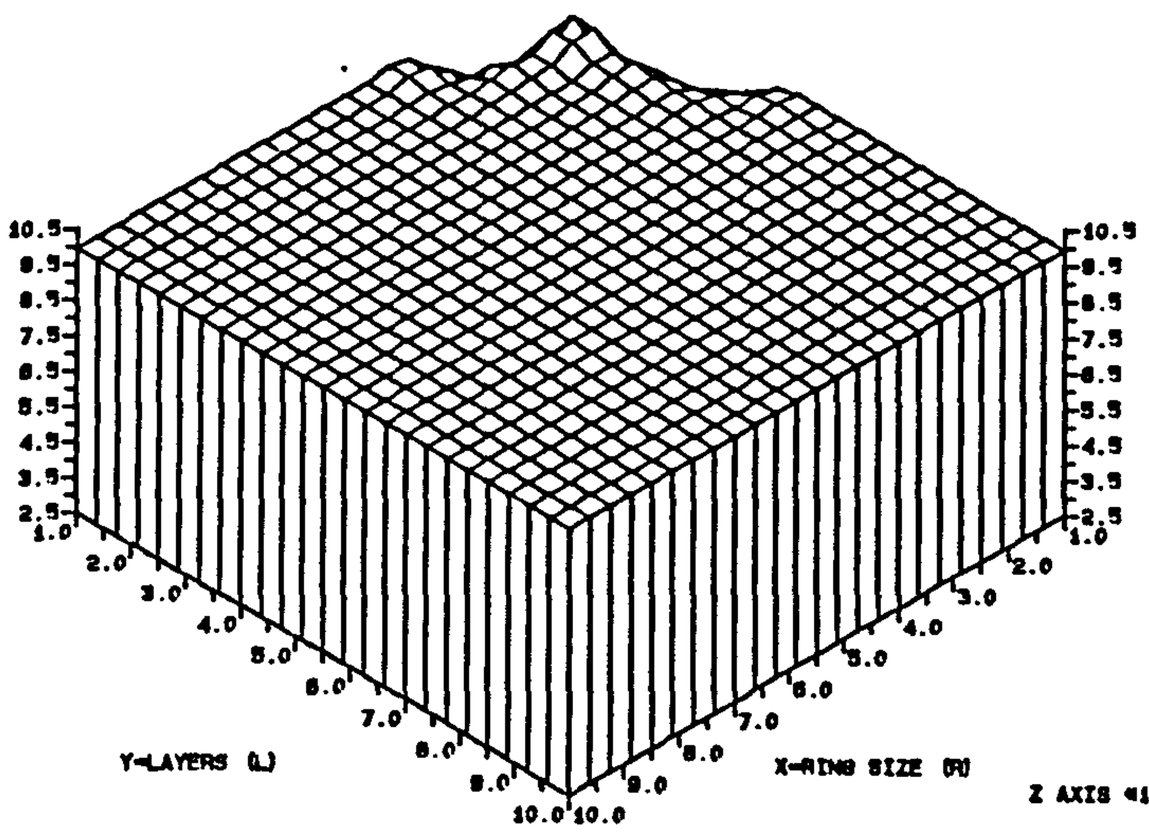
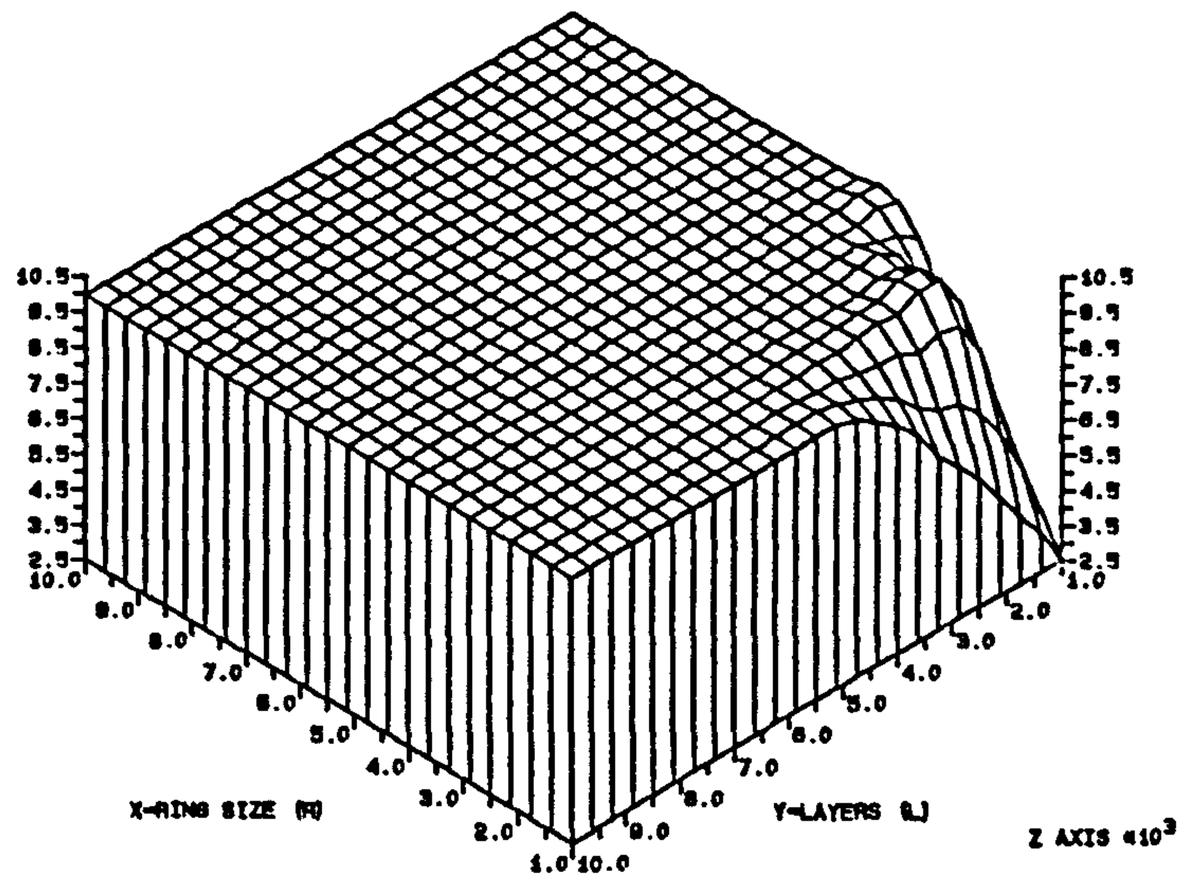


FIG. 5A9.2 HOMOGENEOUS COMMS2 FED AT ONE POINT WITH DATA OF TYPE 10. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





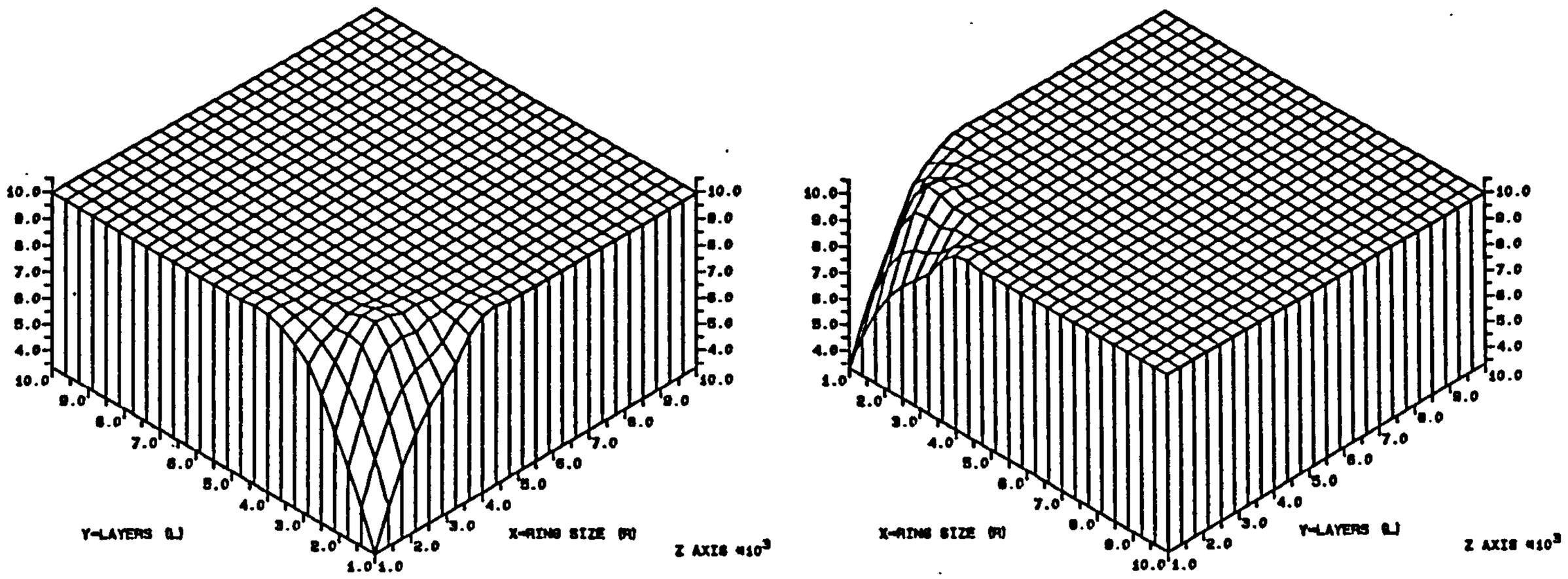
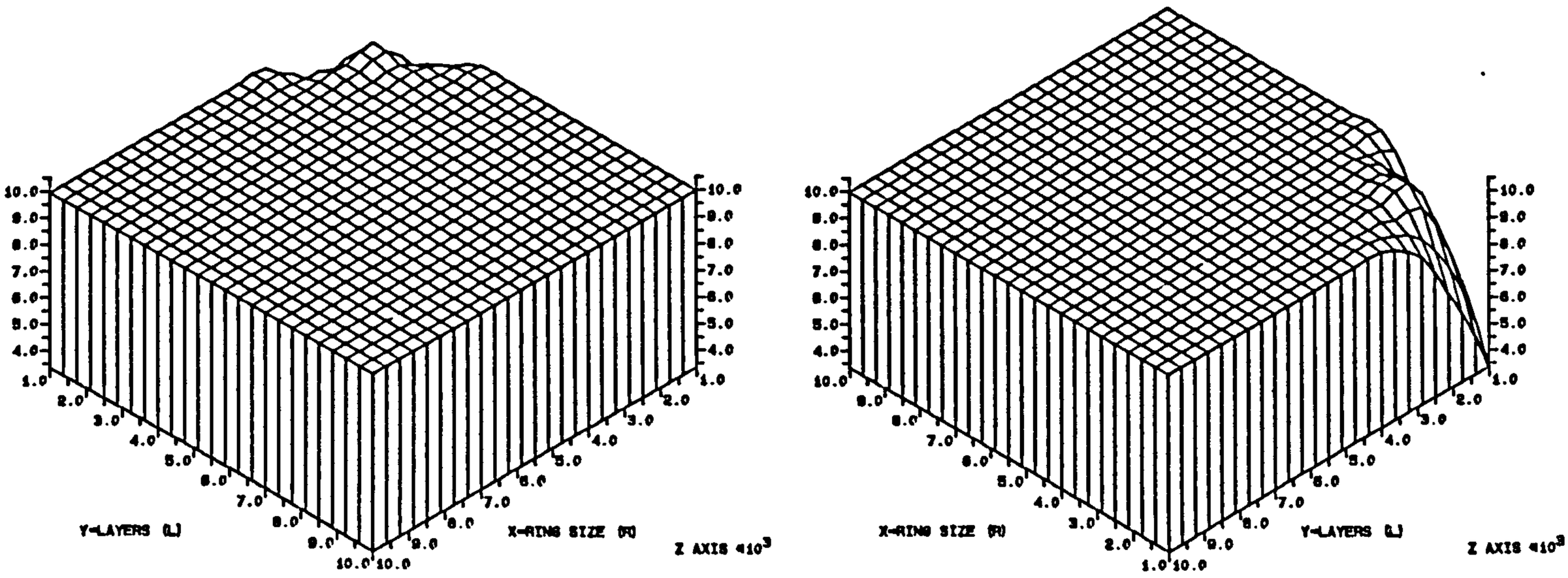
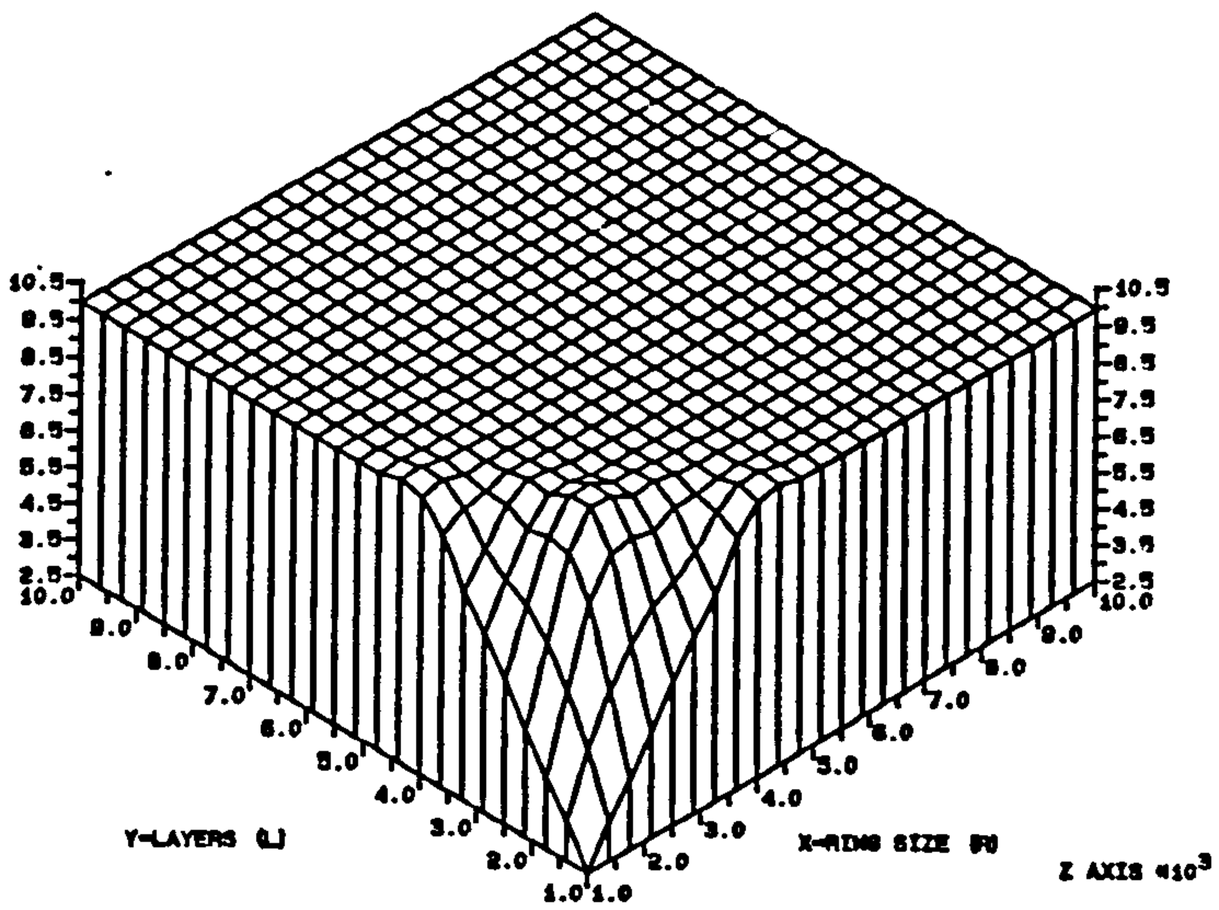


FIG. 5A9.3 HOMOGENEOUS COMMS3 FED AT ONE POINT WITH DATA OF TYPE 10. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





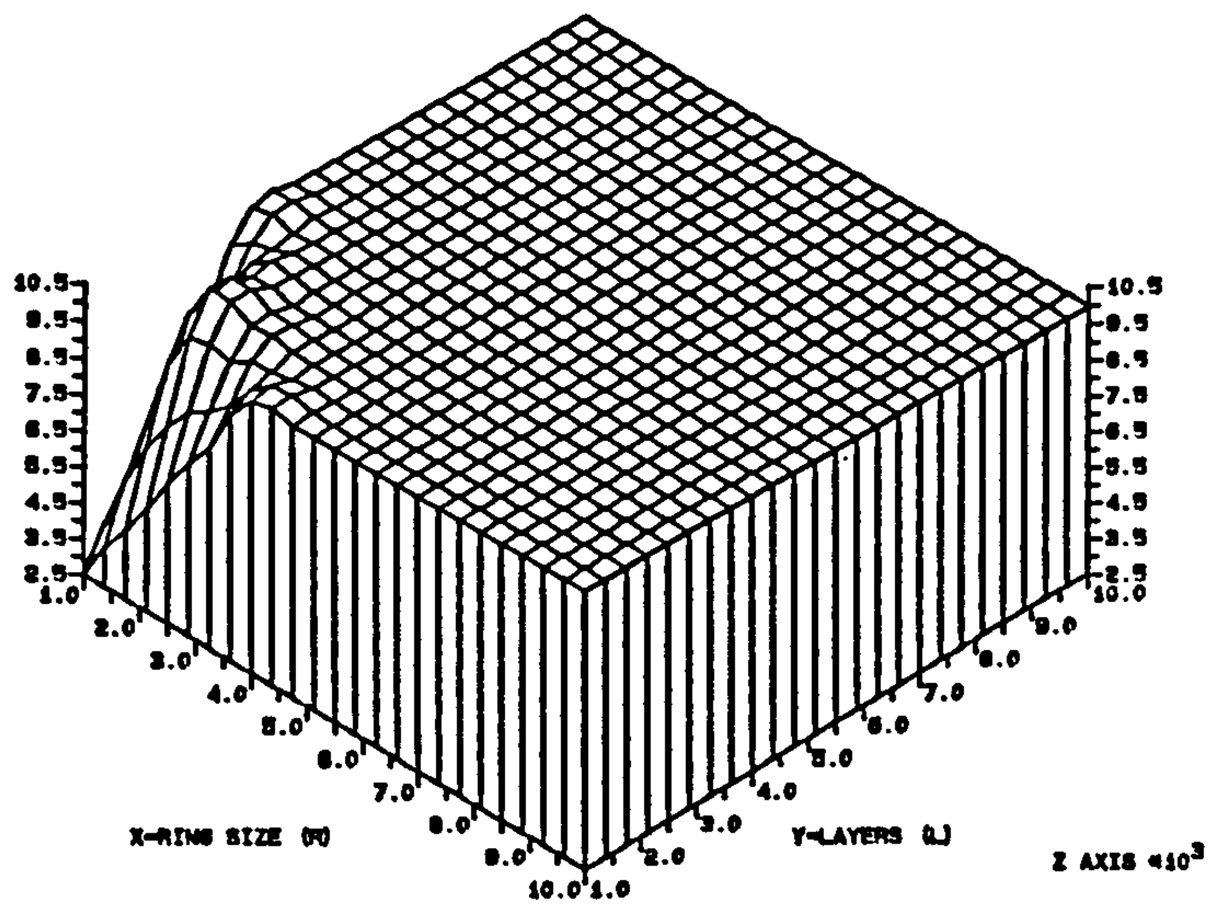
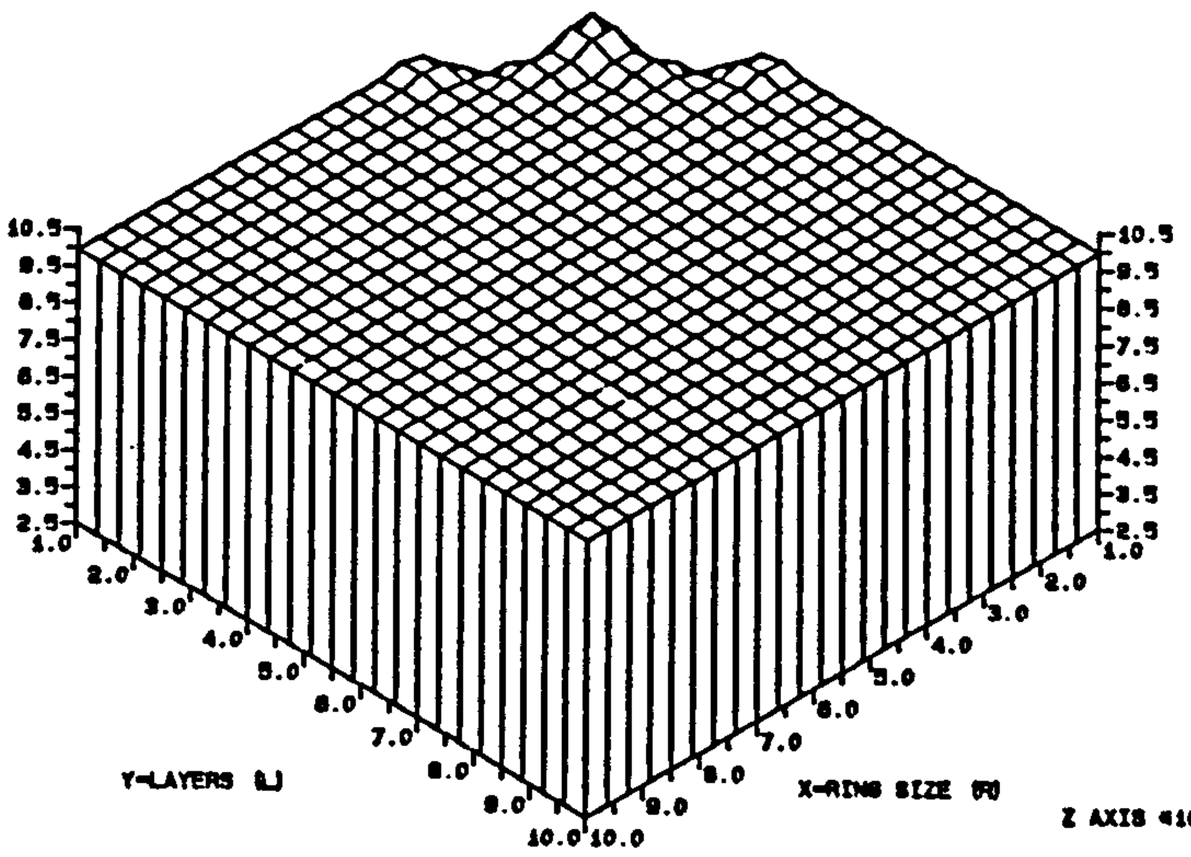
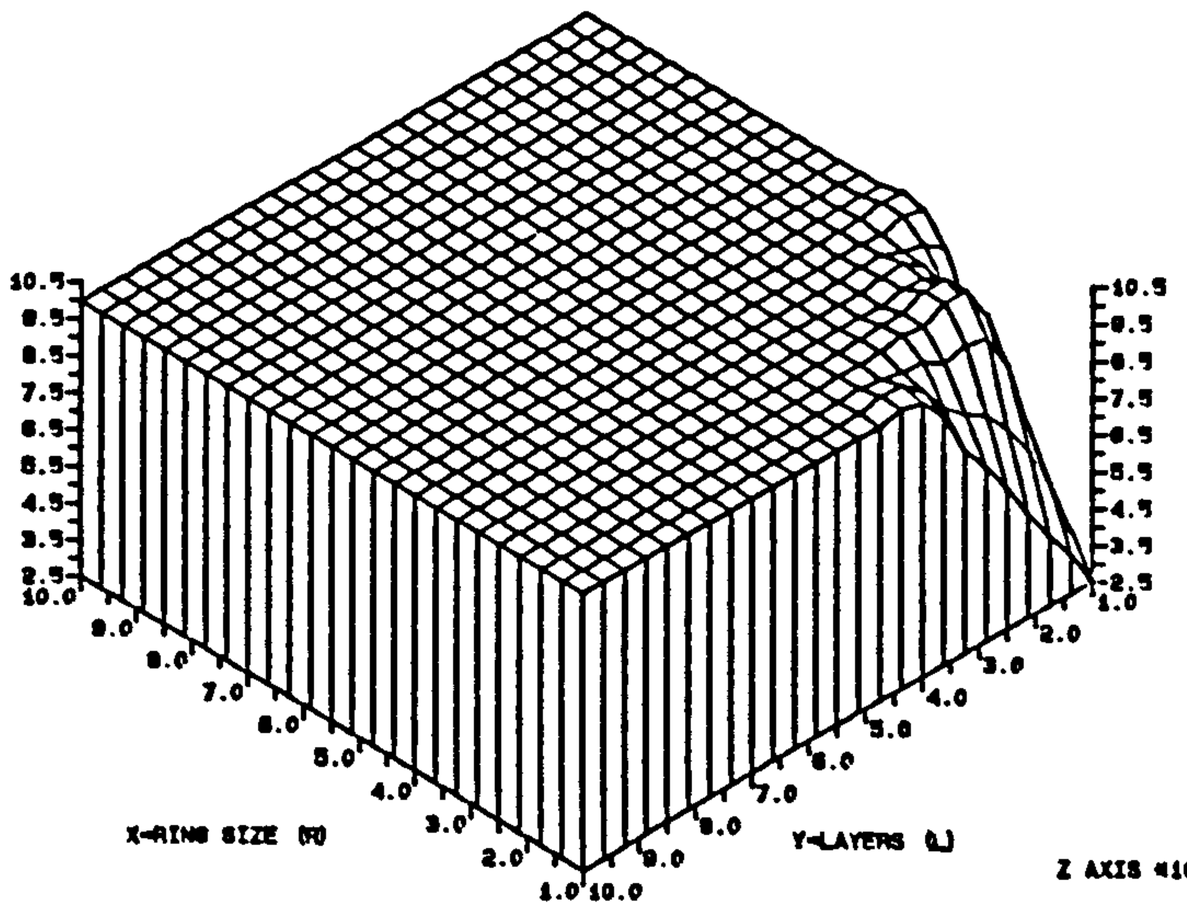


FIG. 5A9.4 HOMOGENEOUS COMMS4 FED AT ONE POINT WITH DATA OF TYPE 10. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





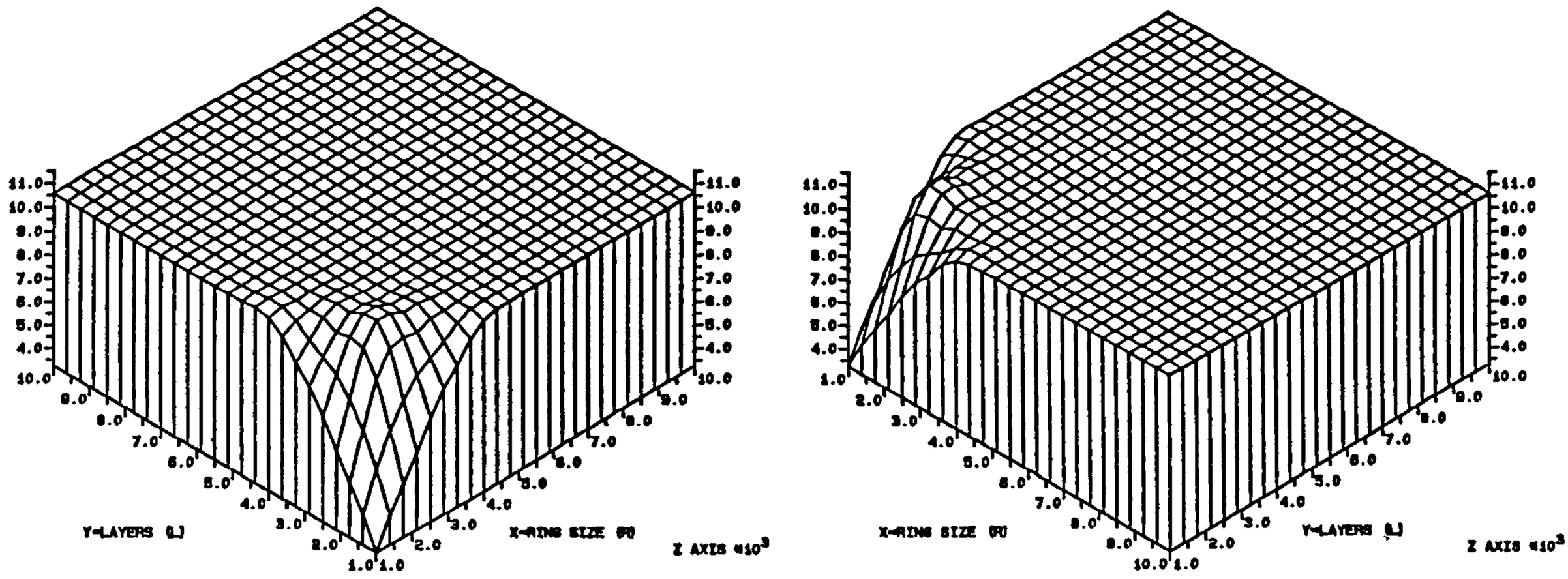
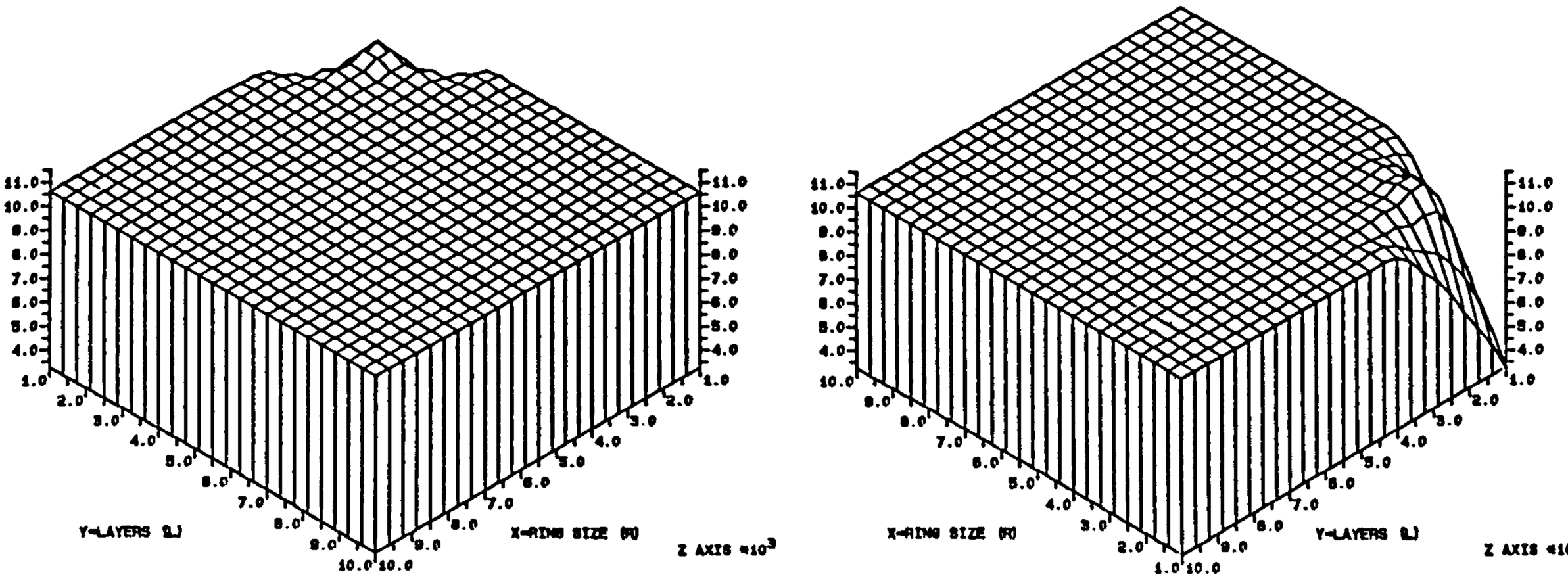


FIG. 5A10.1 HOMOGENEOUS COMMS1 FED AT ONE POINT WITH DATA OF TYPE R20. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.



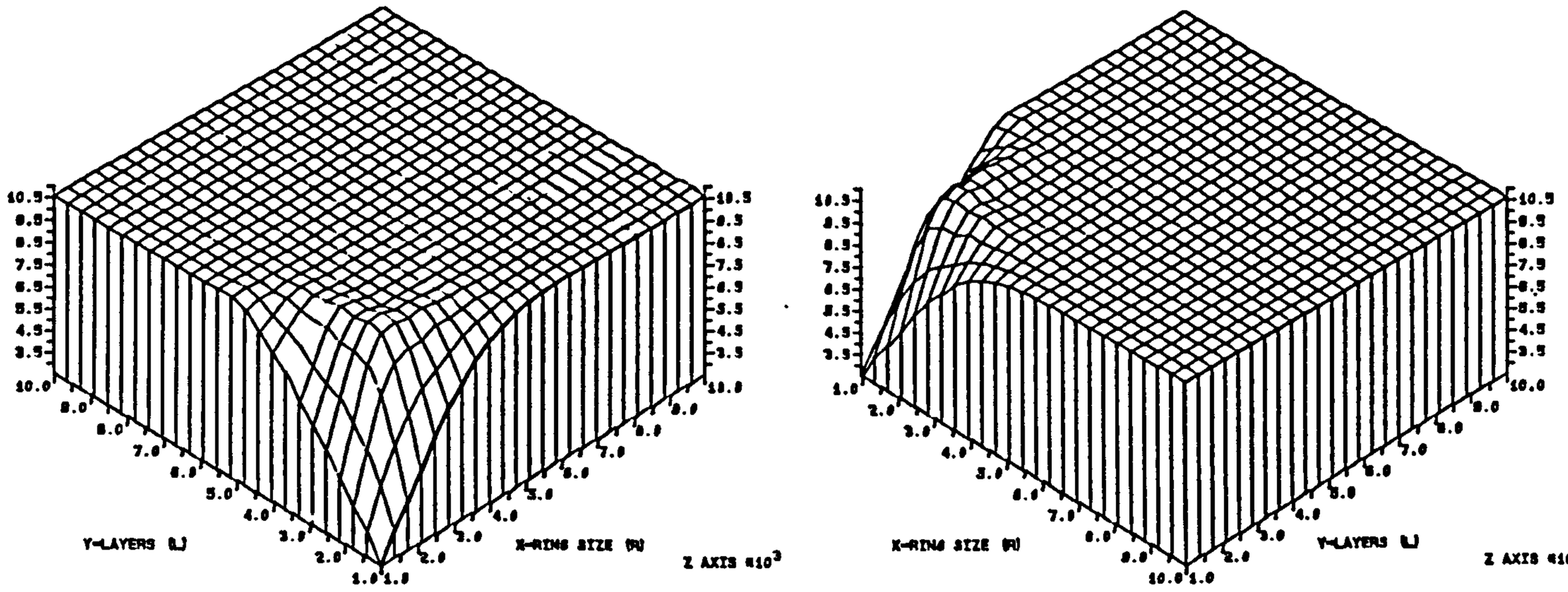
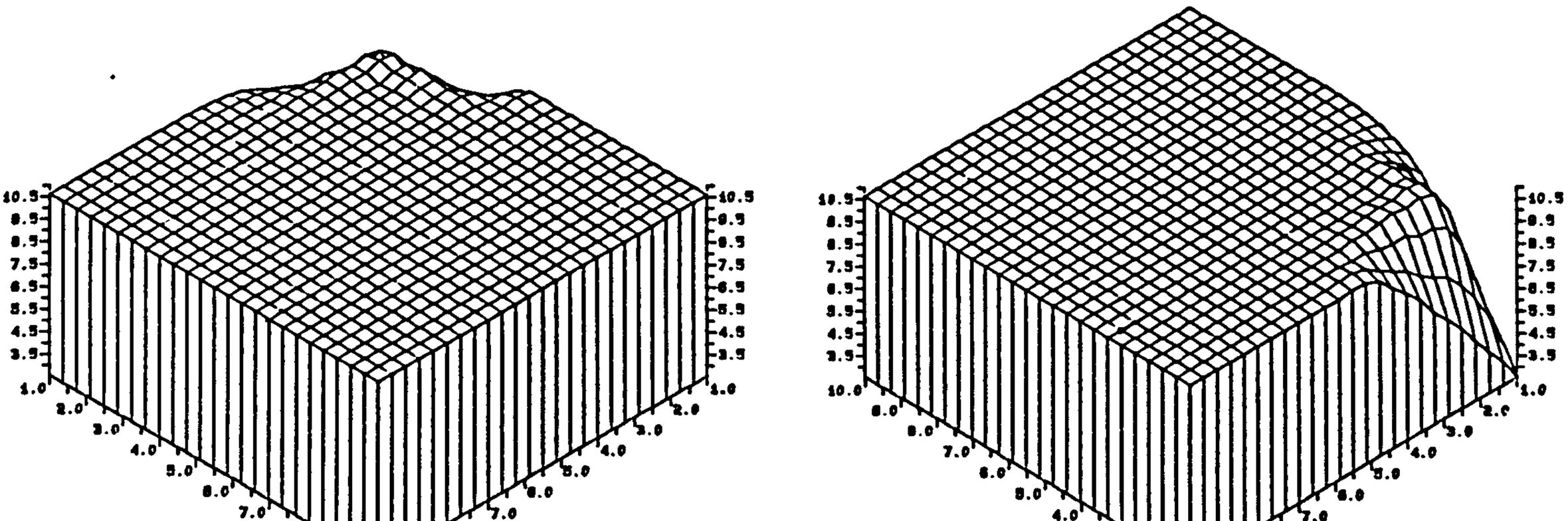
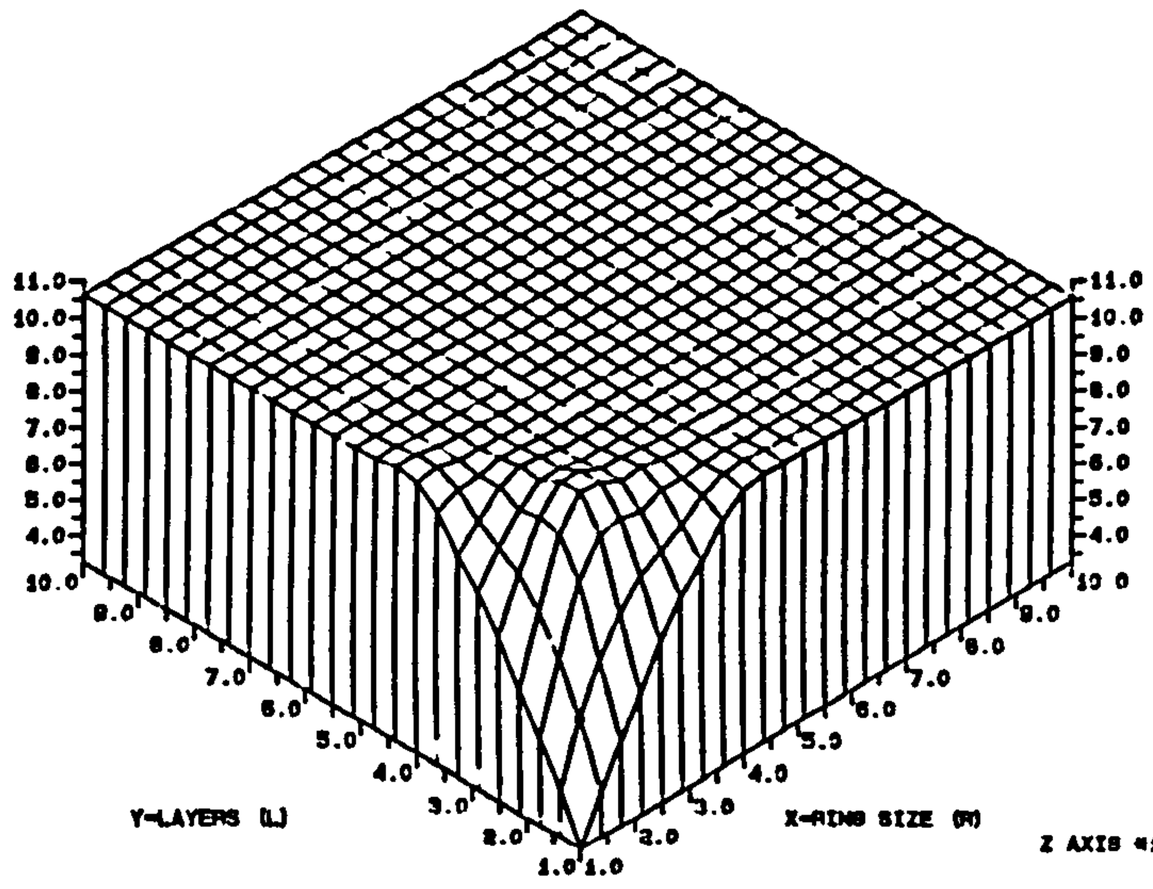


FIG. 5A10.2 HOMOGENEOUS COMMS2 FED AT ONE POINT WITH DATA OF TYPE R20. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





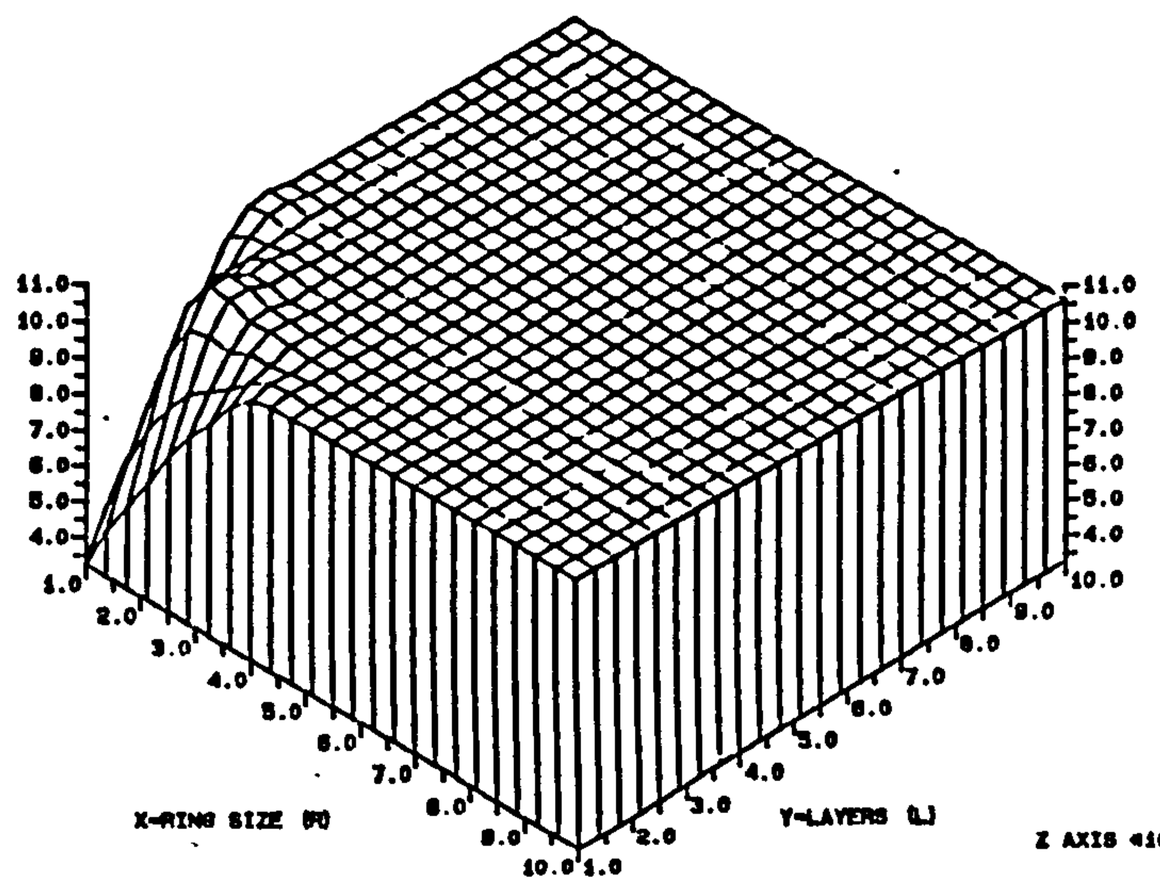
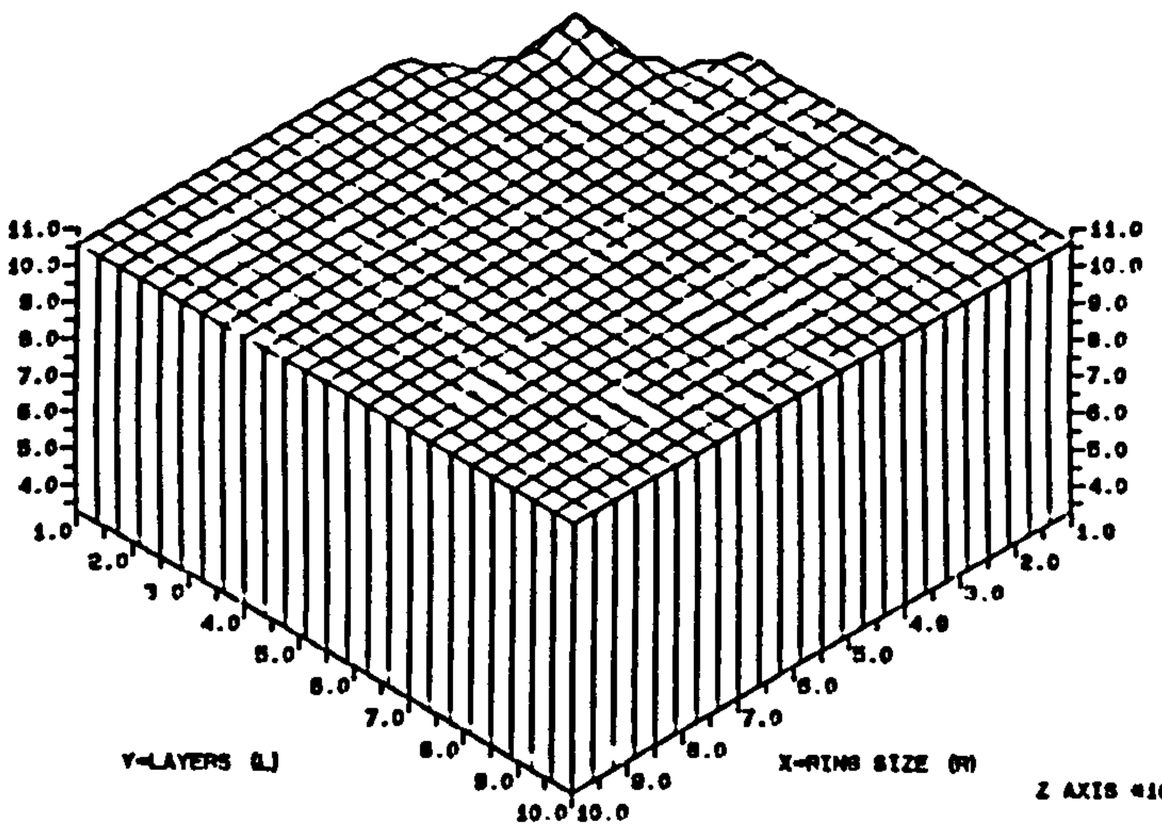
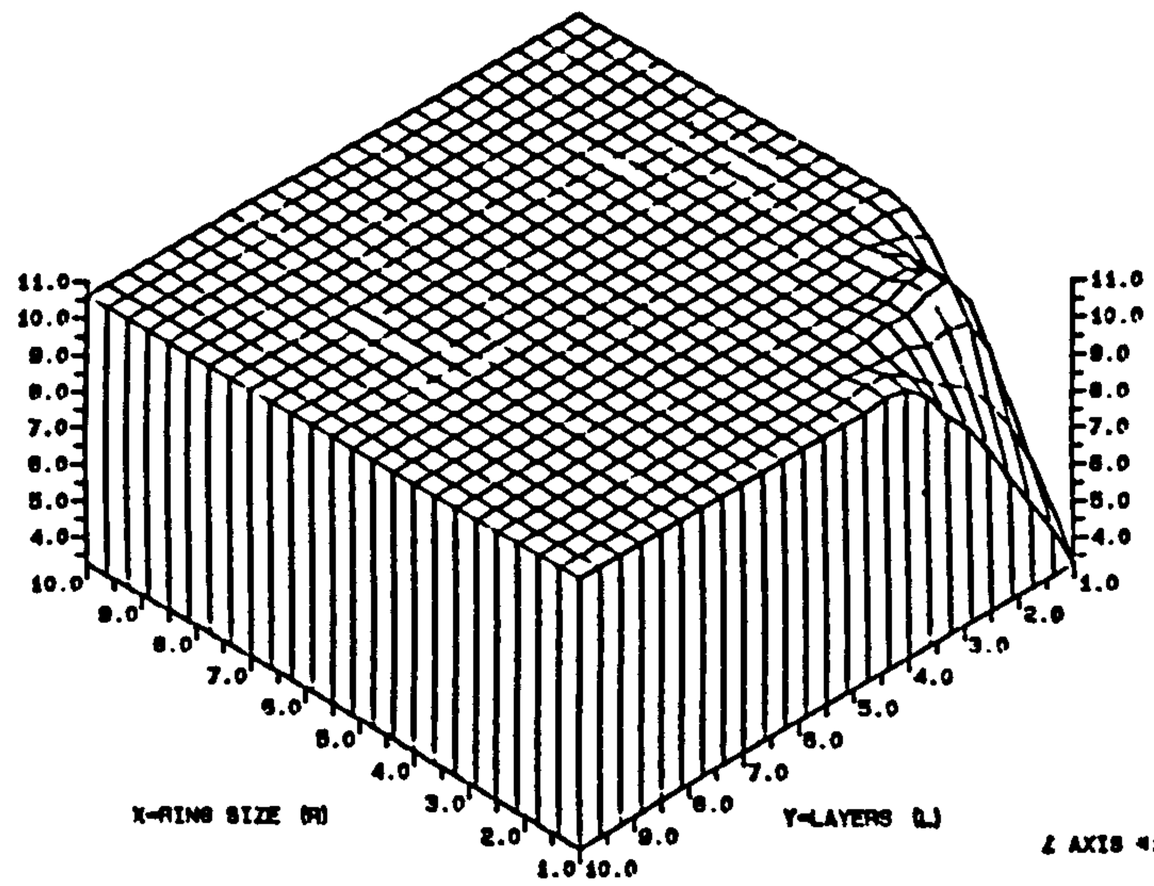


FIG. 5A10.3 HOMOGENEOUS COMMS3 FED AT ONE POINT WITH DATA OF TYPE R20. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





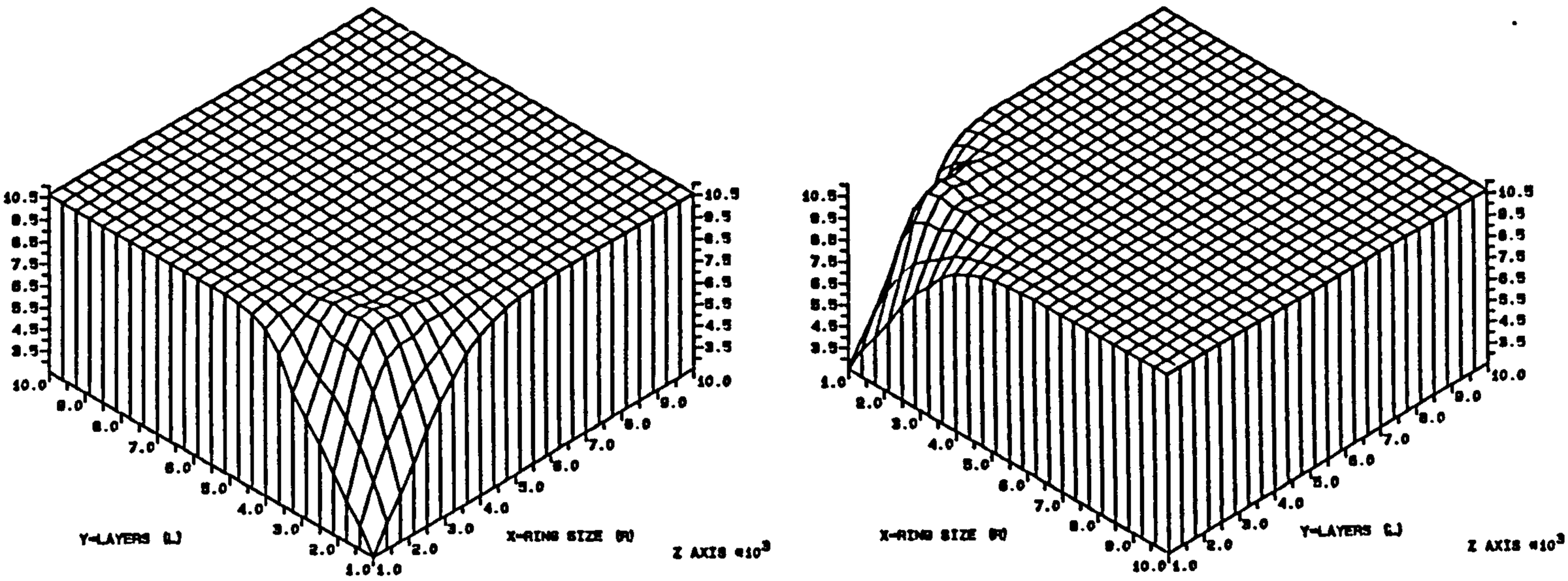
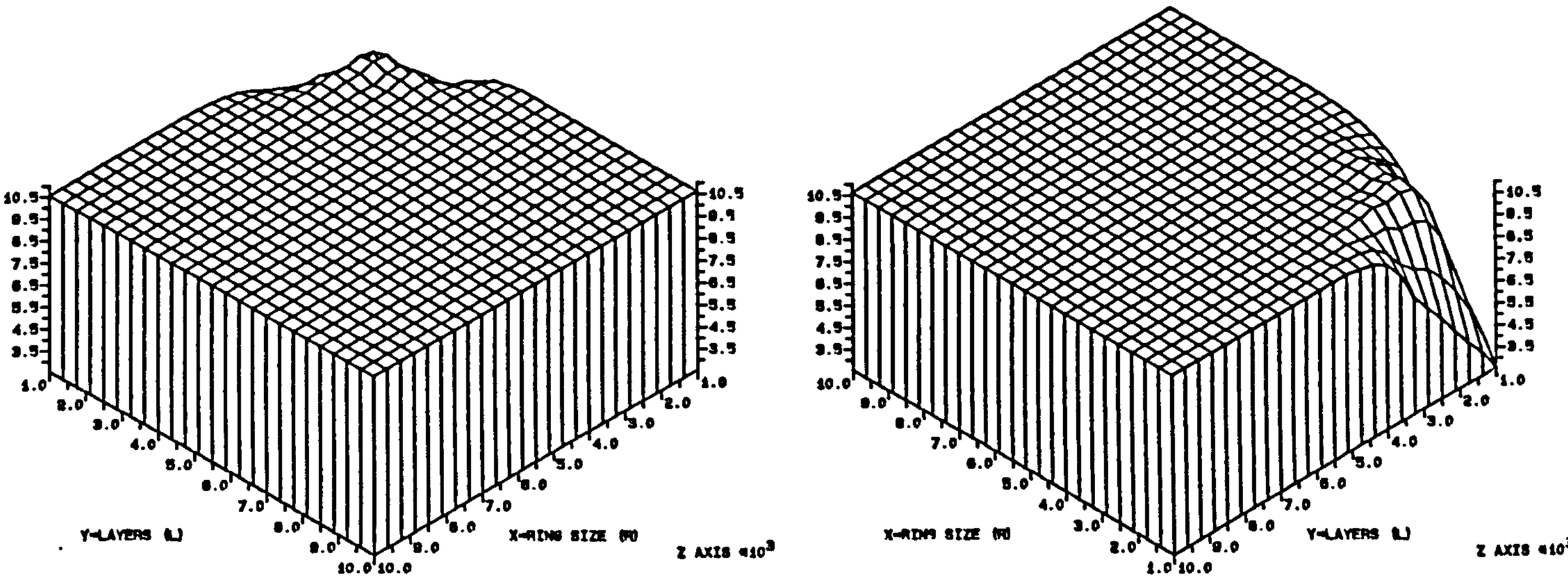
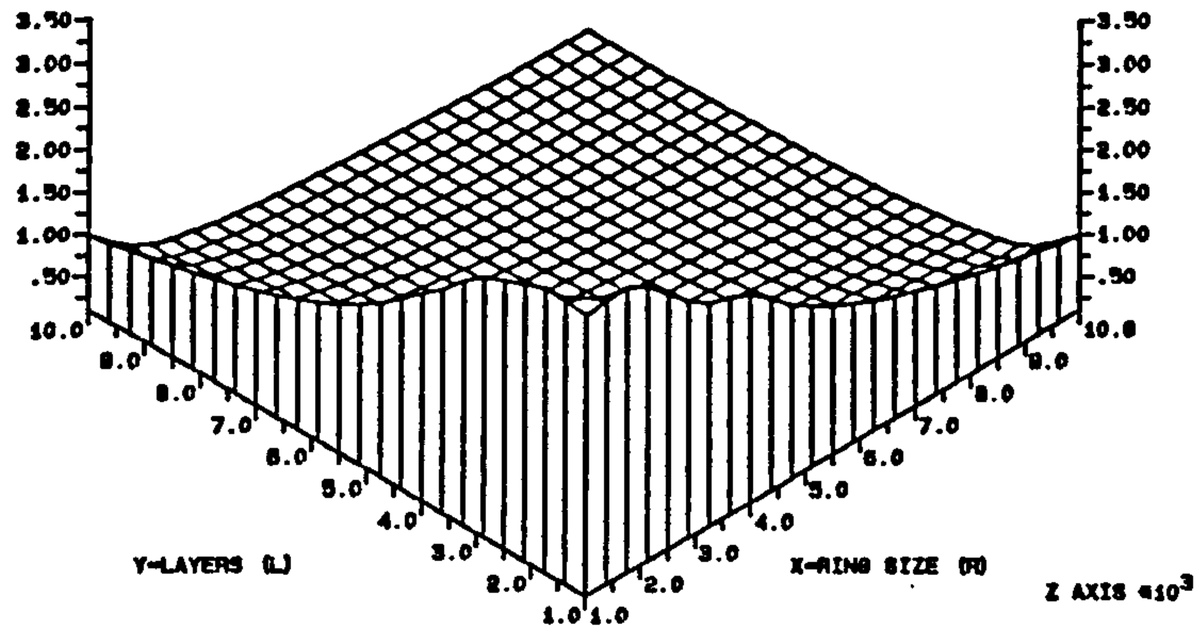


FIG. 5A10.4 HOMOGENEOUS COMMS4 FED AT ONE POINT WITH DATA OF TYPE R20. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





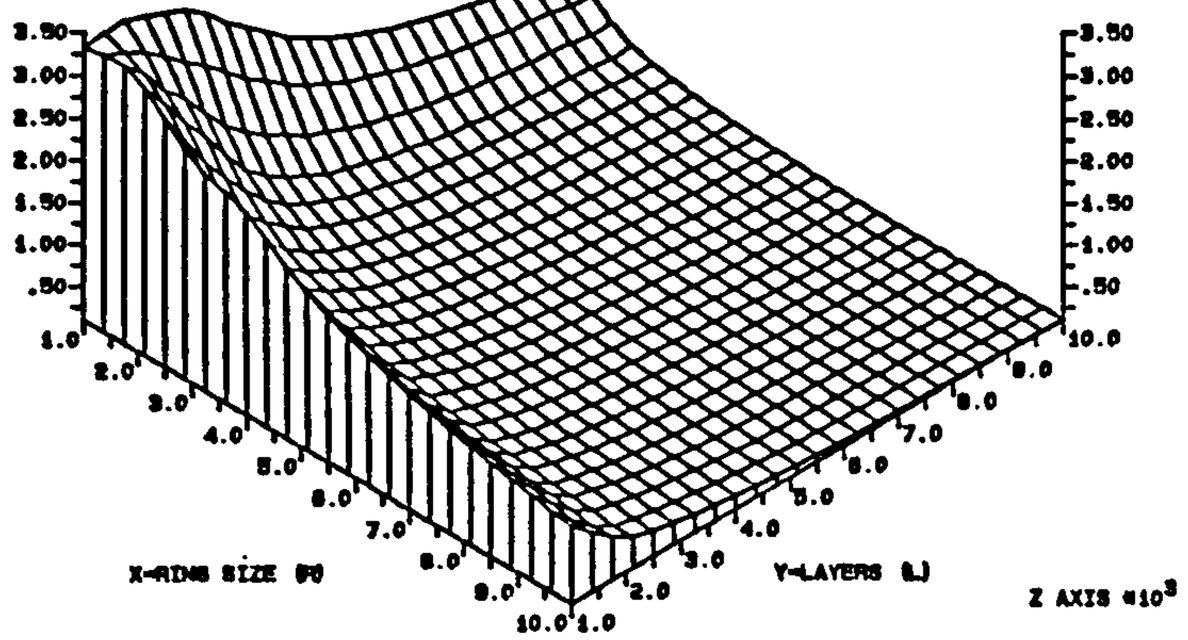
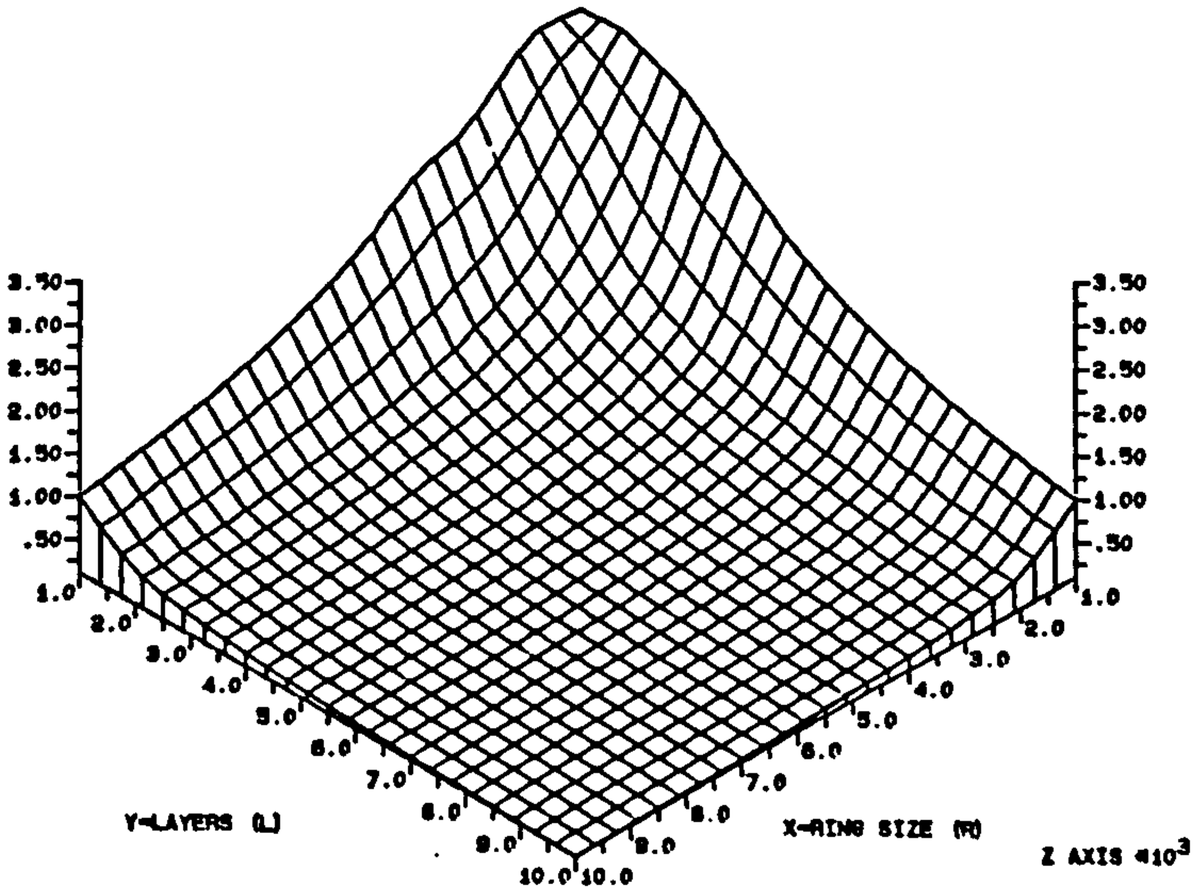
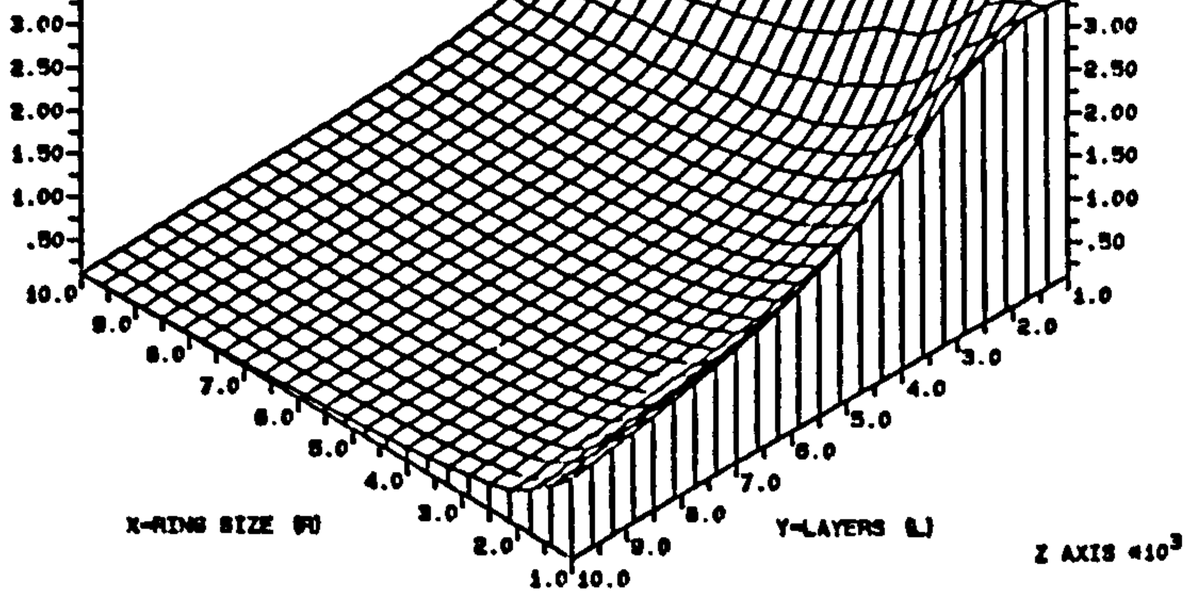


FIG. 5A11.1 HOMOGENEOUS COMMS1 FED AT ONE POINT WITH DATA OF TYPE 10. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





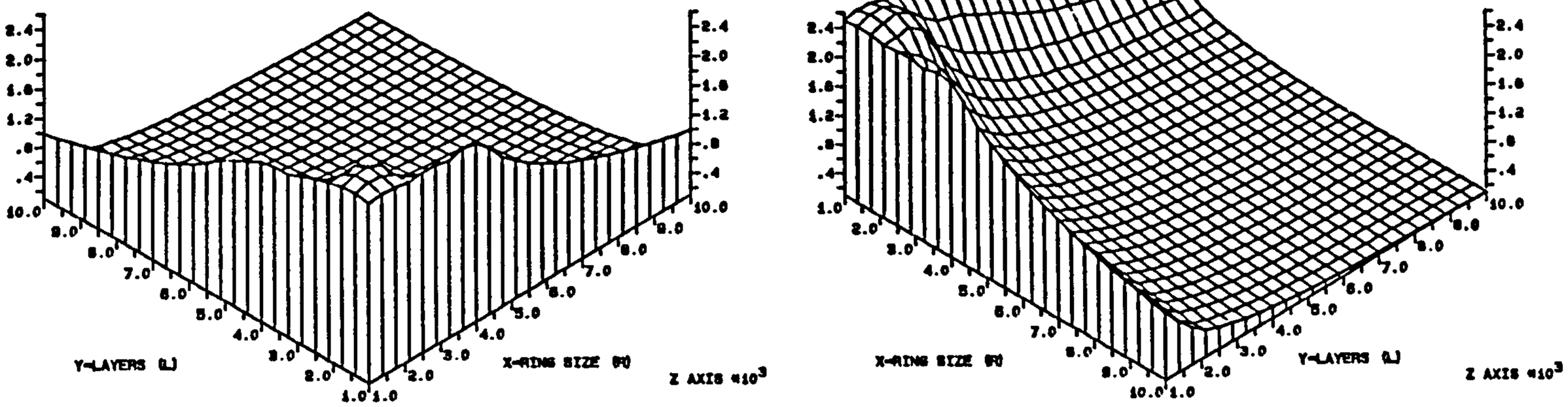
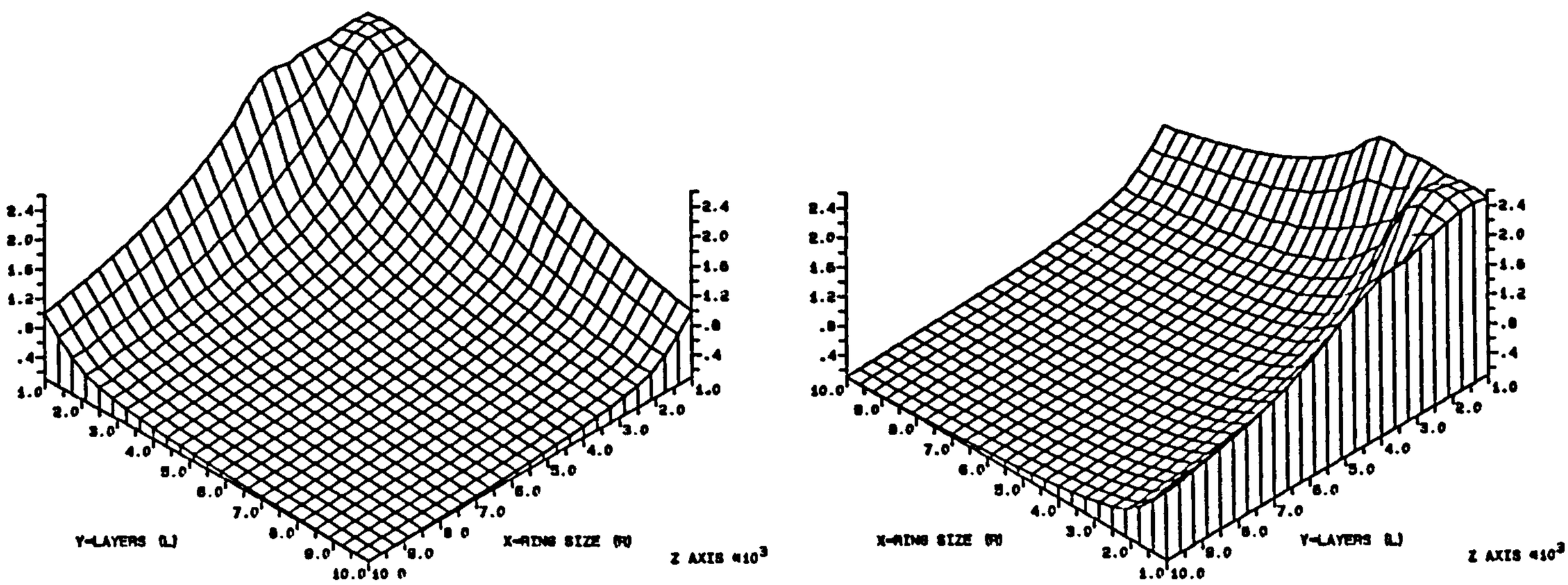
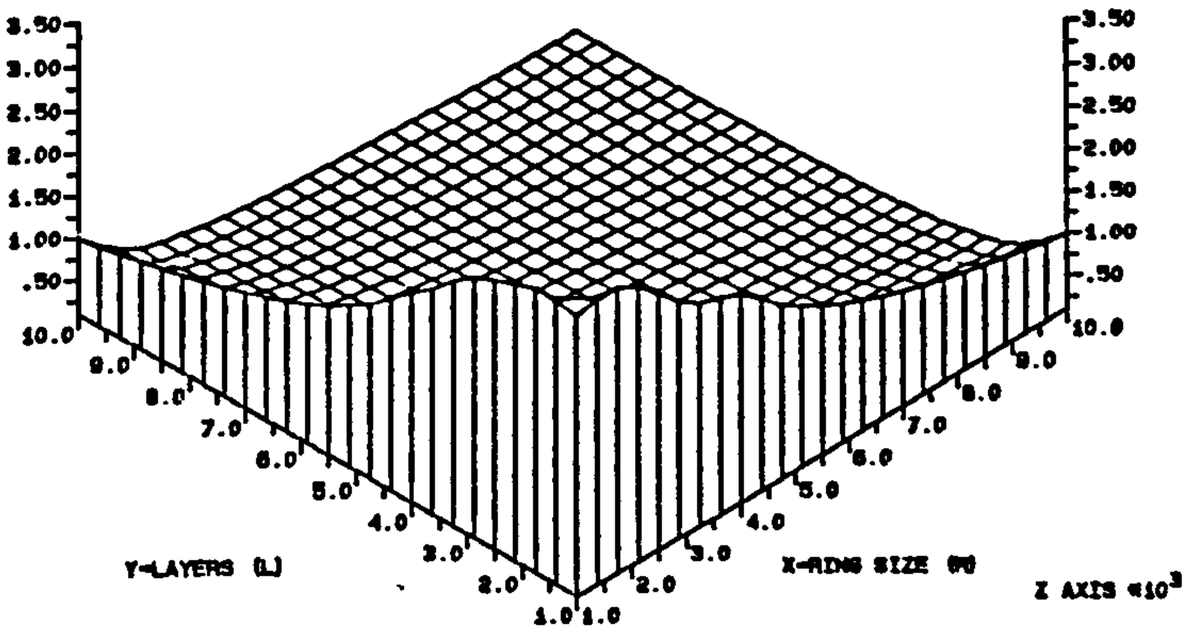


FIG. 5A11.2 HOMOGENEOUS COMMS2 FED AT ONE POINT WITH DATA OF TYPE 10. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





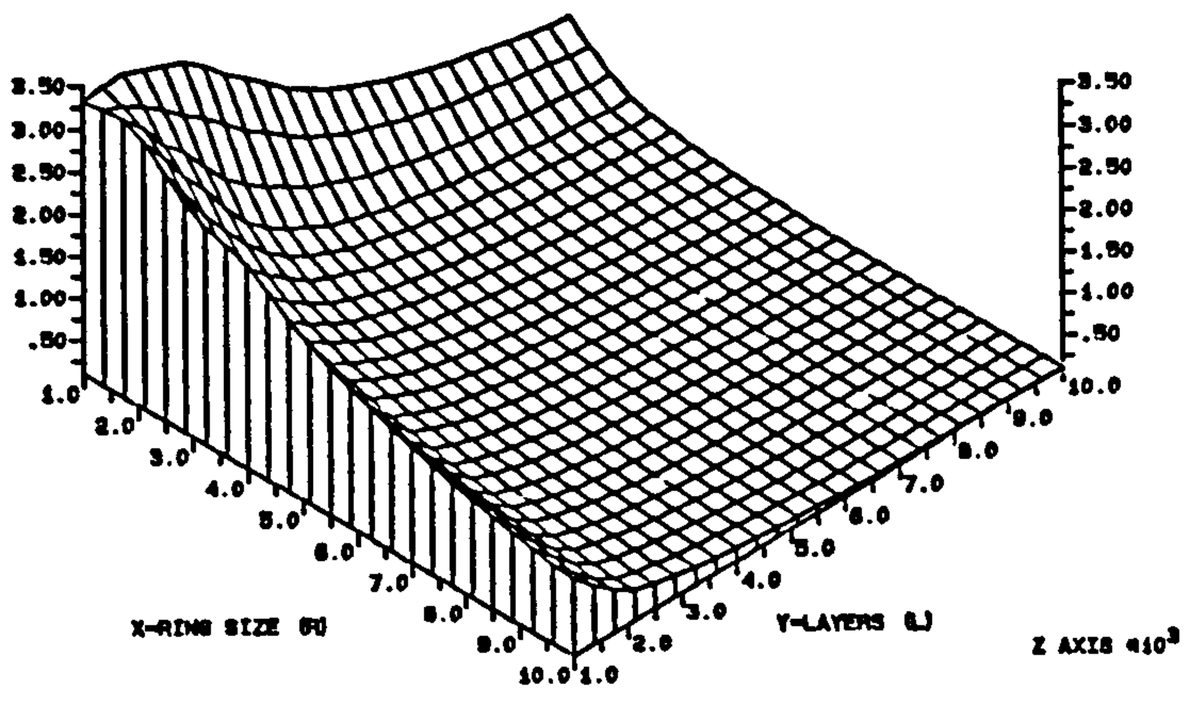
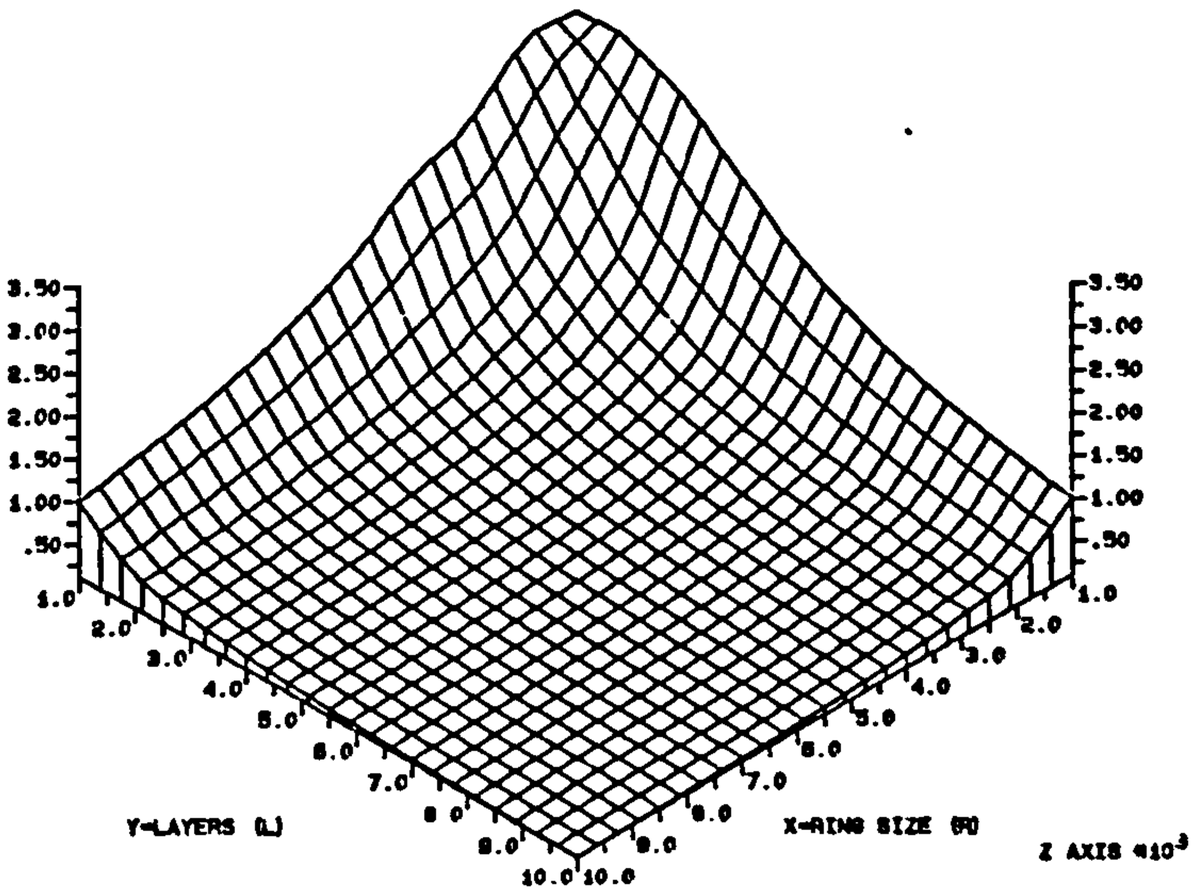
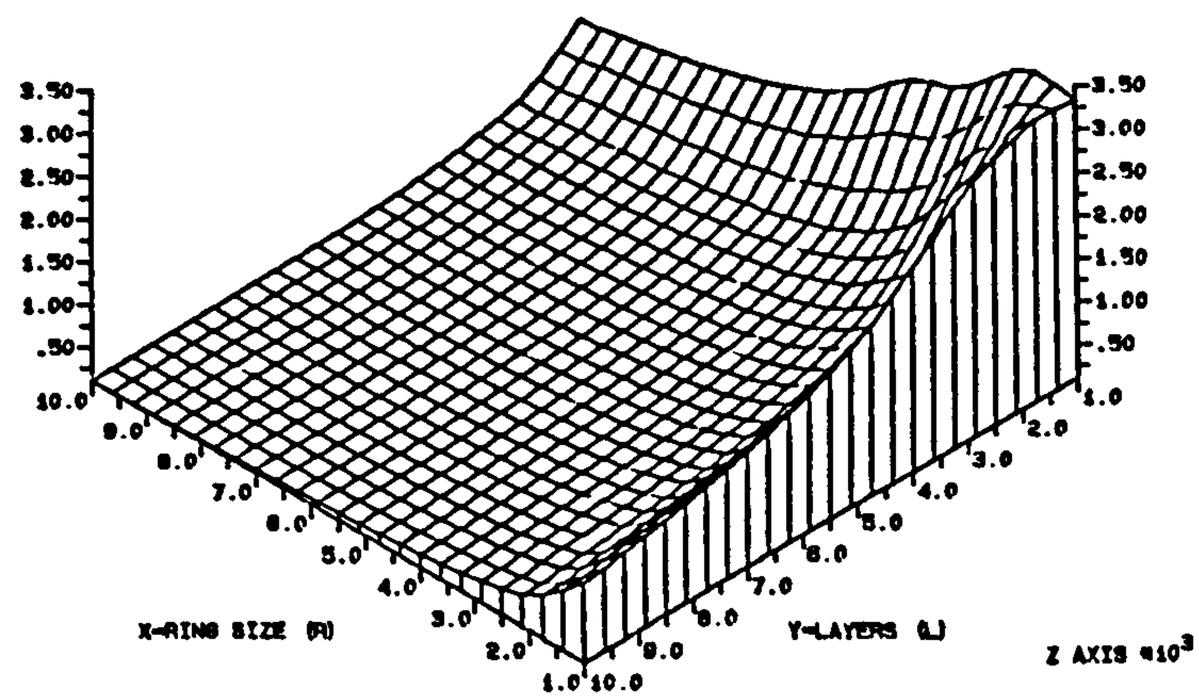
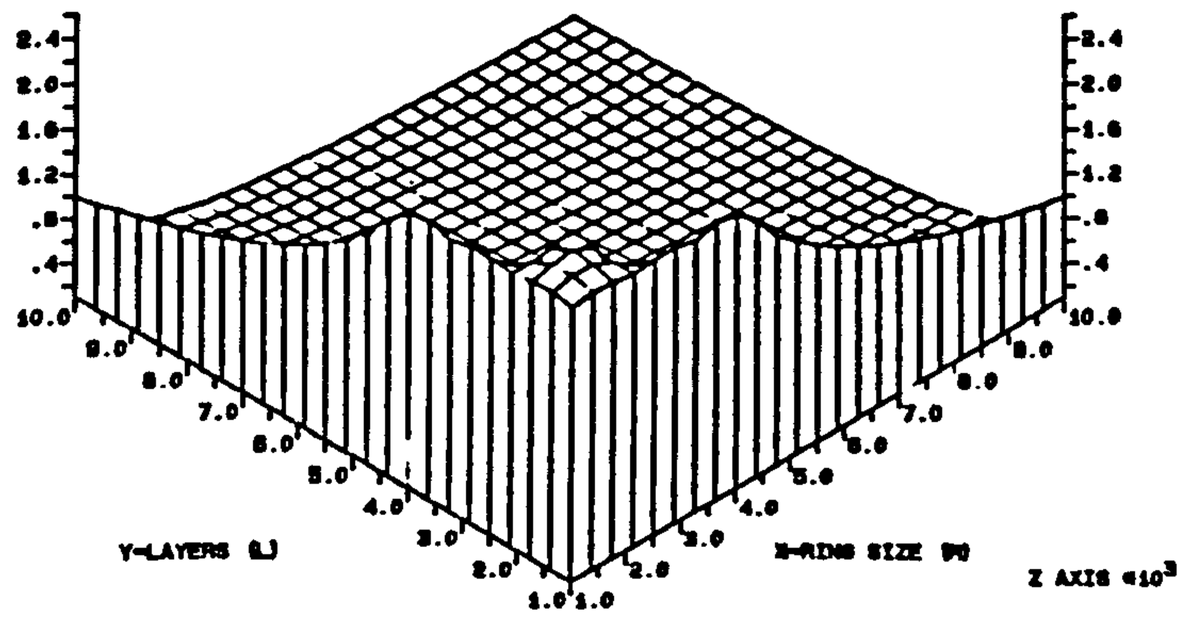


FIG. 5A11.3 HOMOGENEOUS COMMS3 FED AT ONE POINT WITH DATA OF TYPE 10. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.







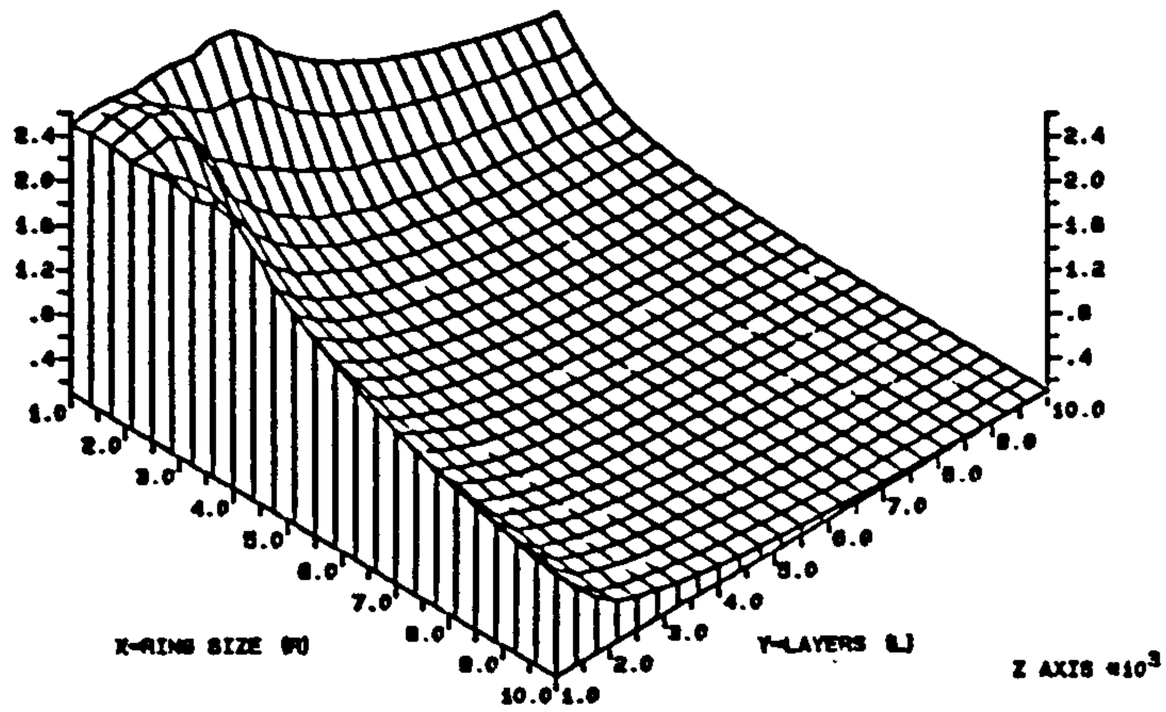
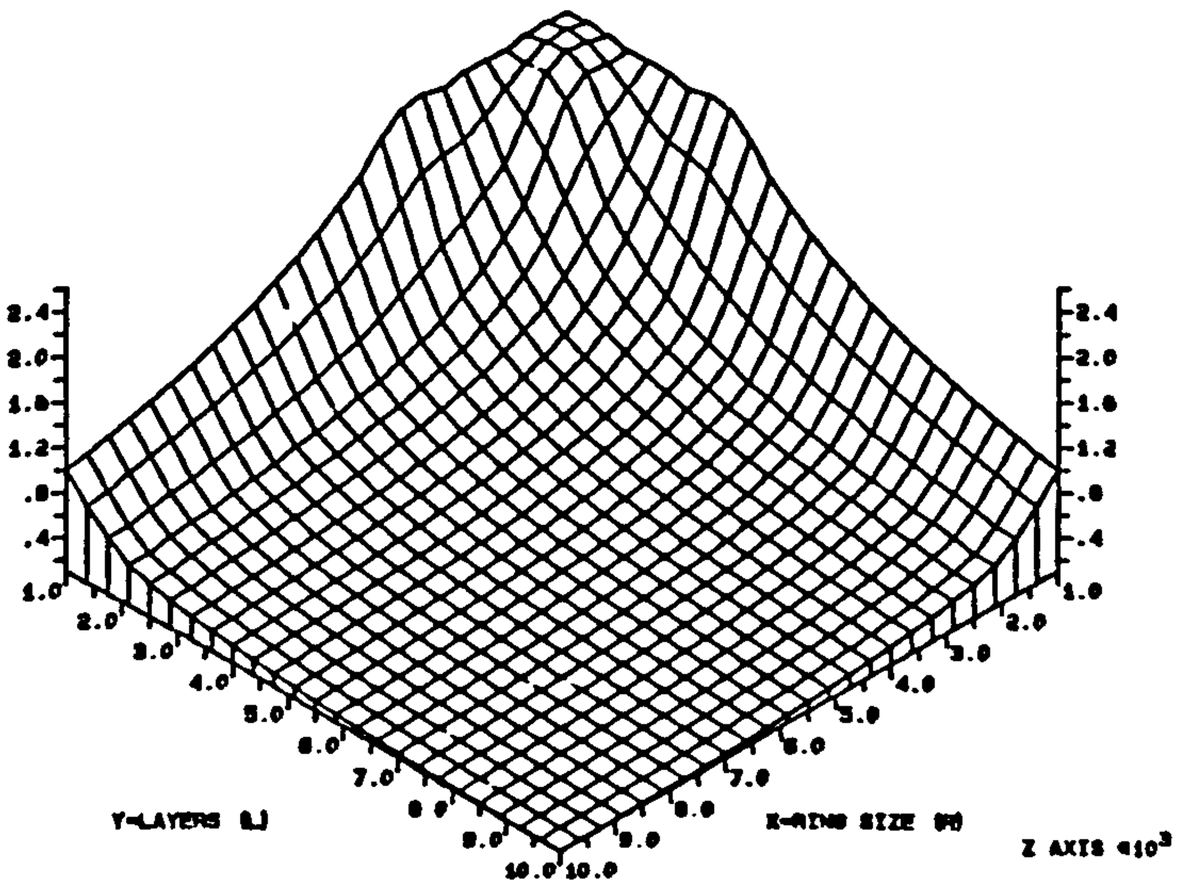
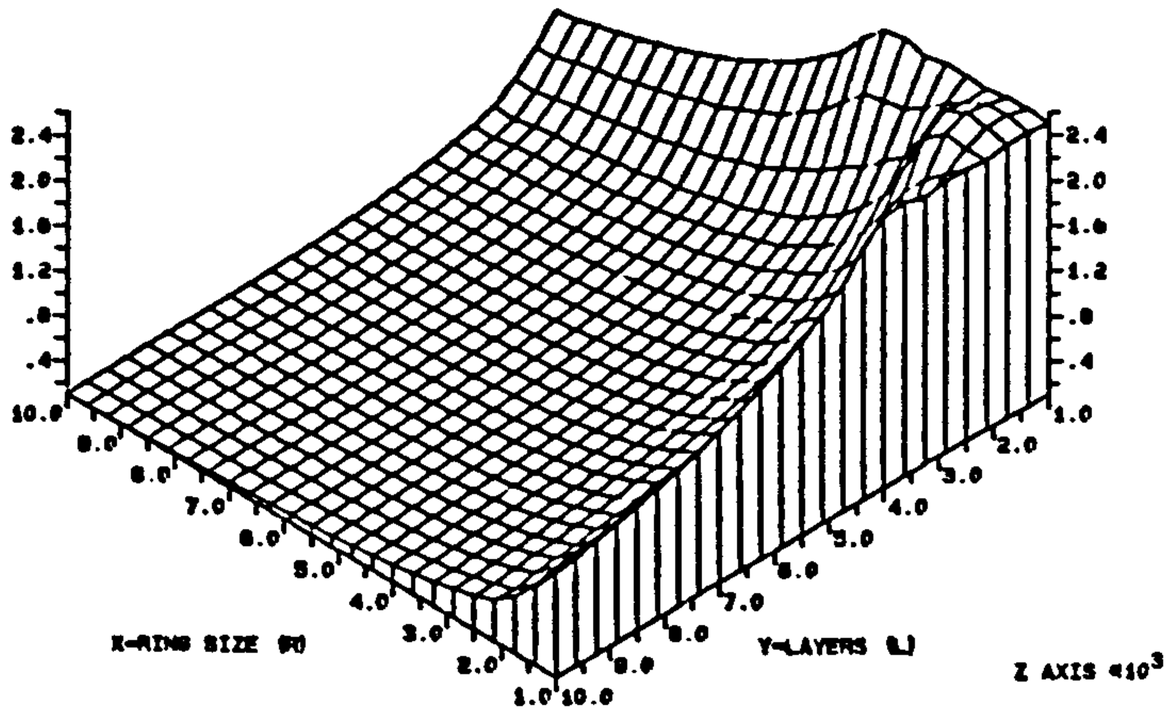
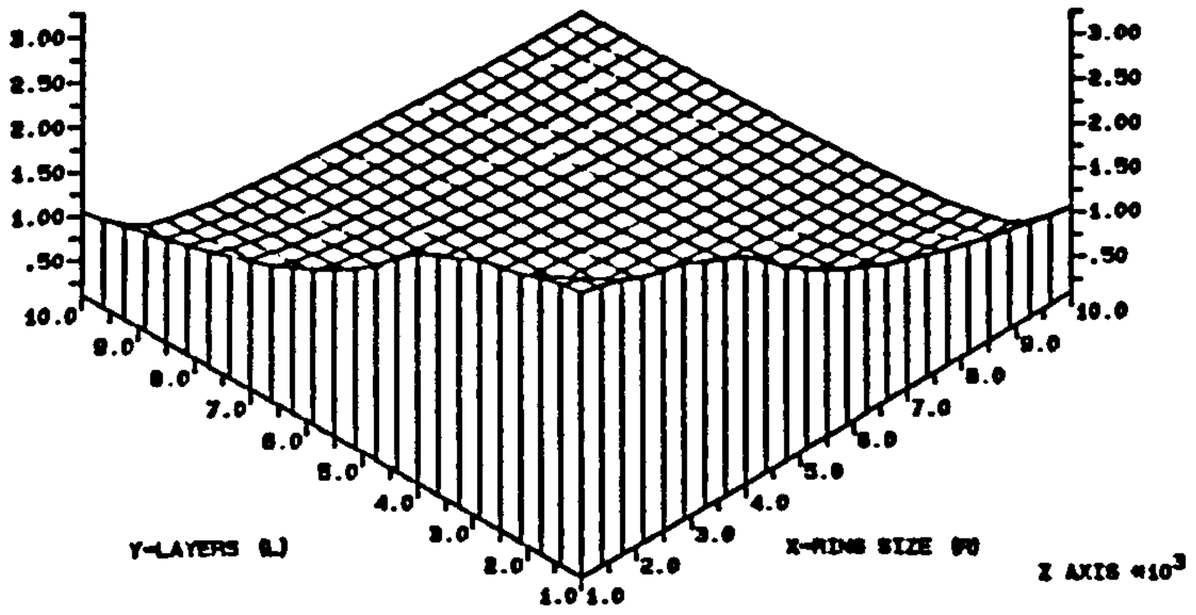


FIG. 5A11.4 HOMOGENEOUS COMMS4 FED AT ONE POINT WITH DATA OF TYPE 10. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.







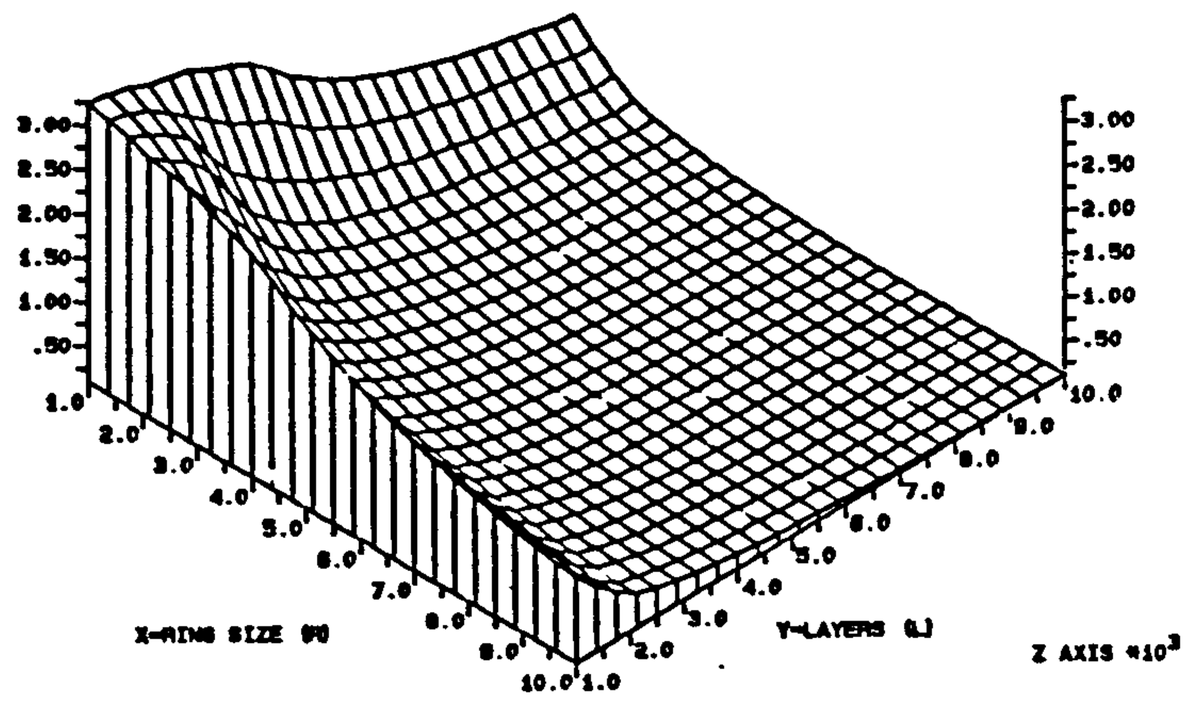
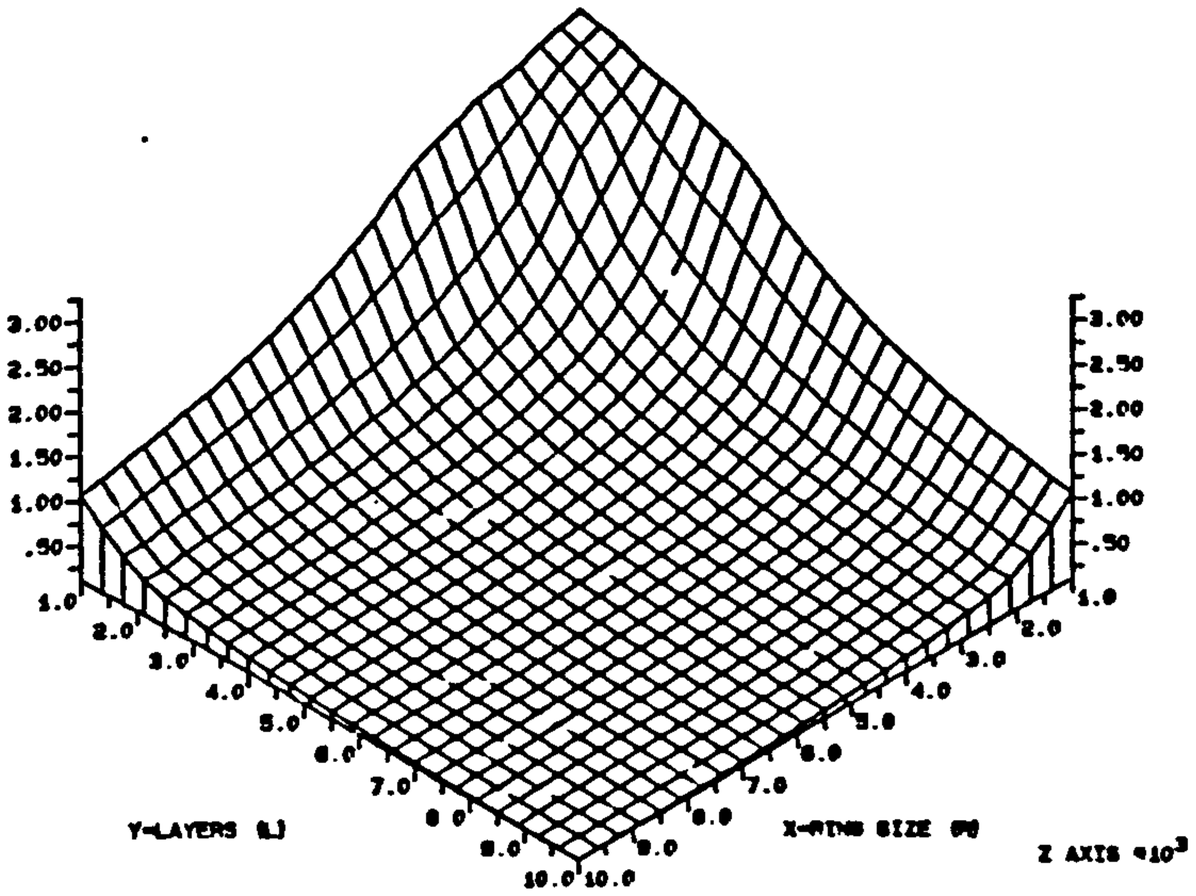
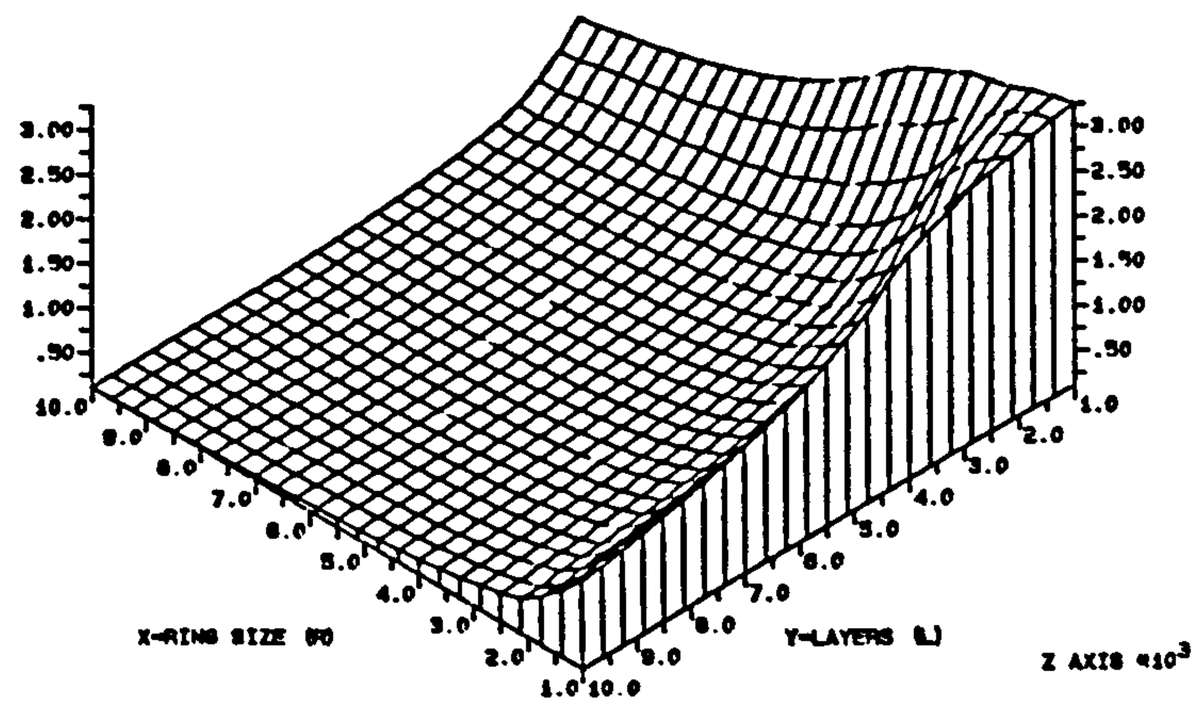


FIG. 5A12.1 HOMOGENEOUS COMMS1 FED AT ONE POINT WITH DATA OF TYPE R20. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





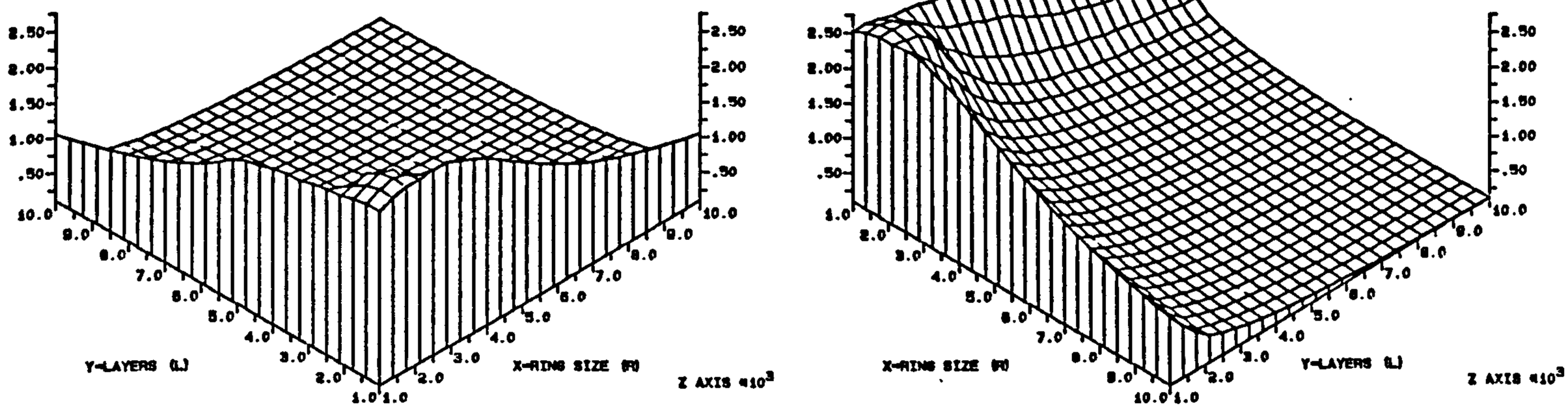
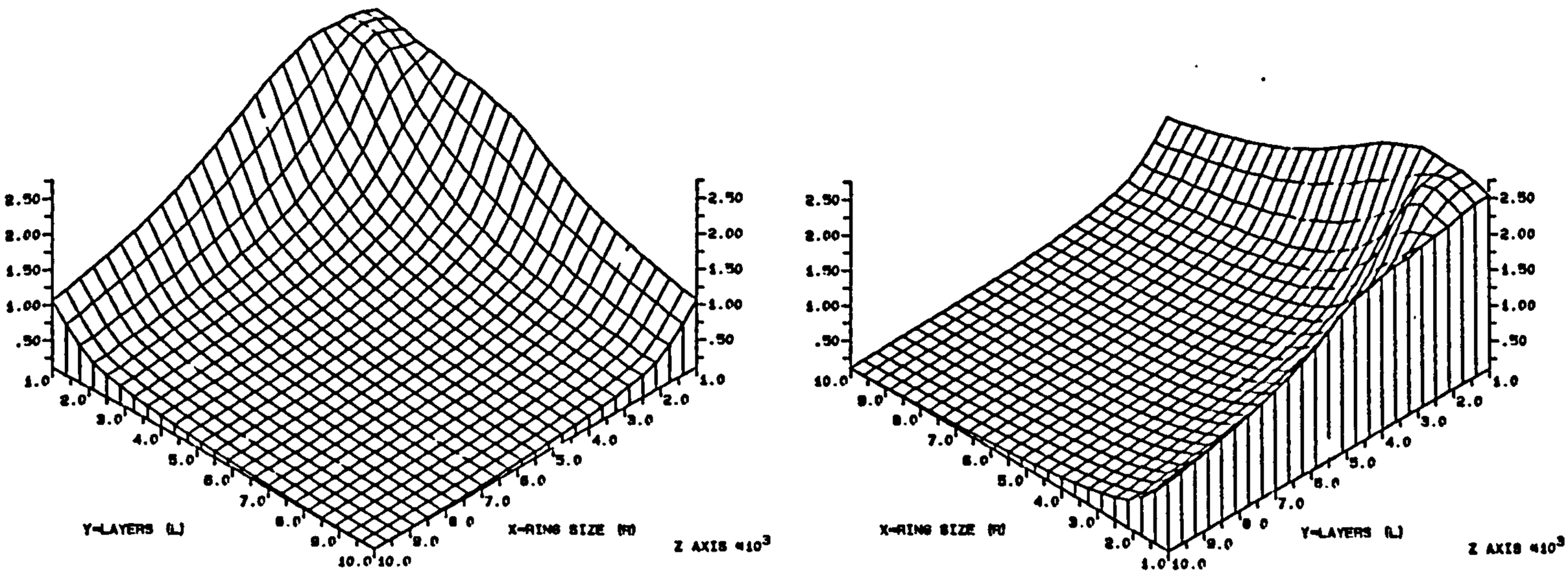


FIG. 5A12.2 HOMOGENEOUS COMMS2 FED AT ONE POINT WITH DATA OF TYPE R20. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



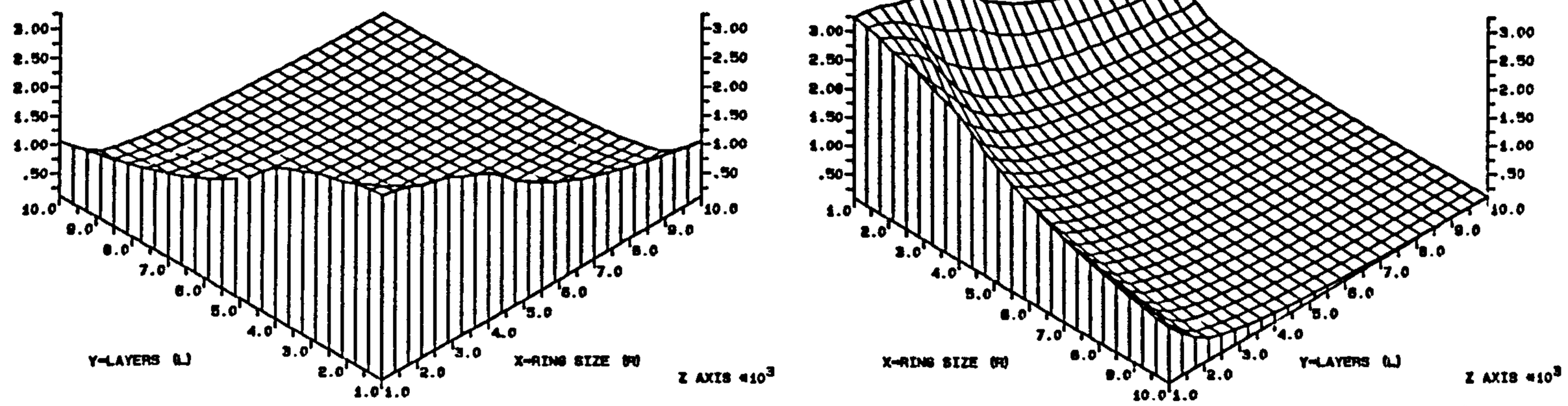
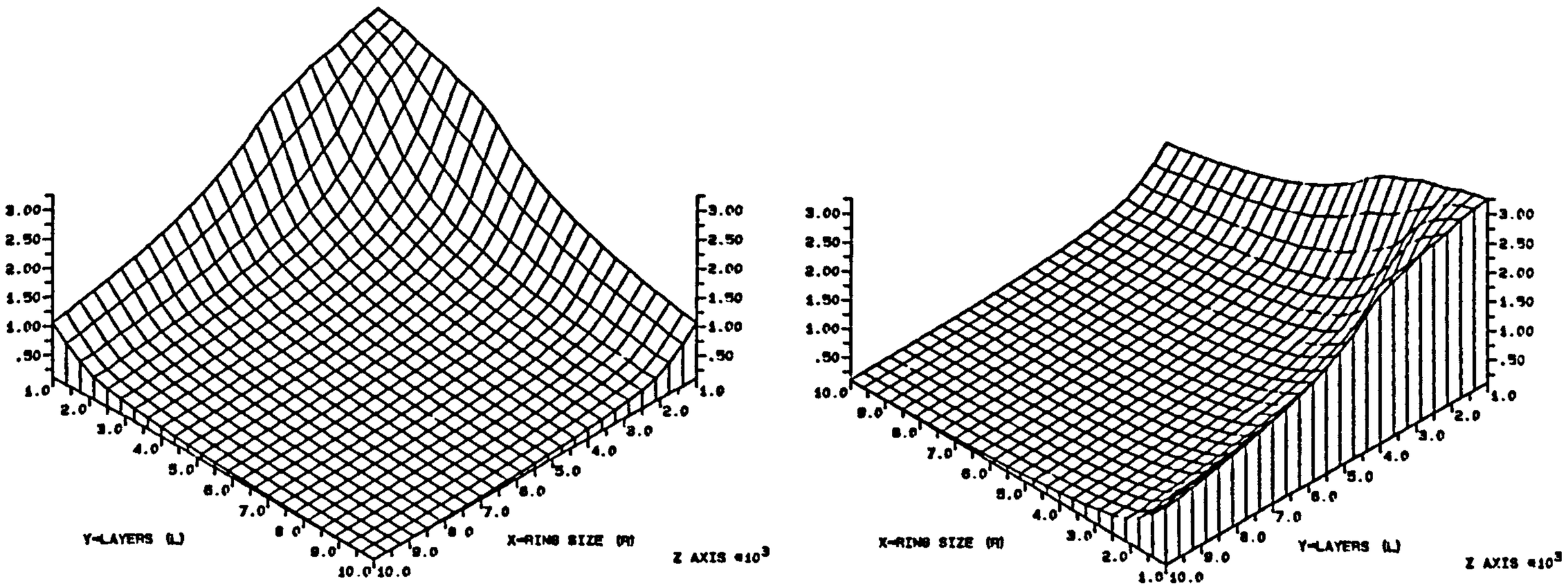


FIG. 5A12.3 HOMOGENEOUS COMMS3 FED AT ONE POINT WITH DATA OF TYPE R20. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.



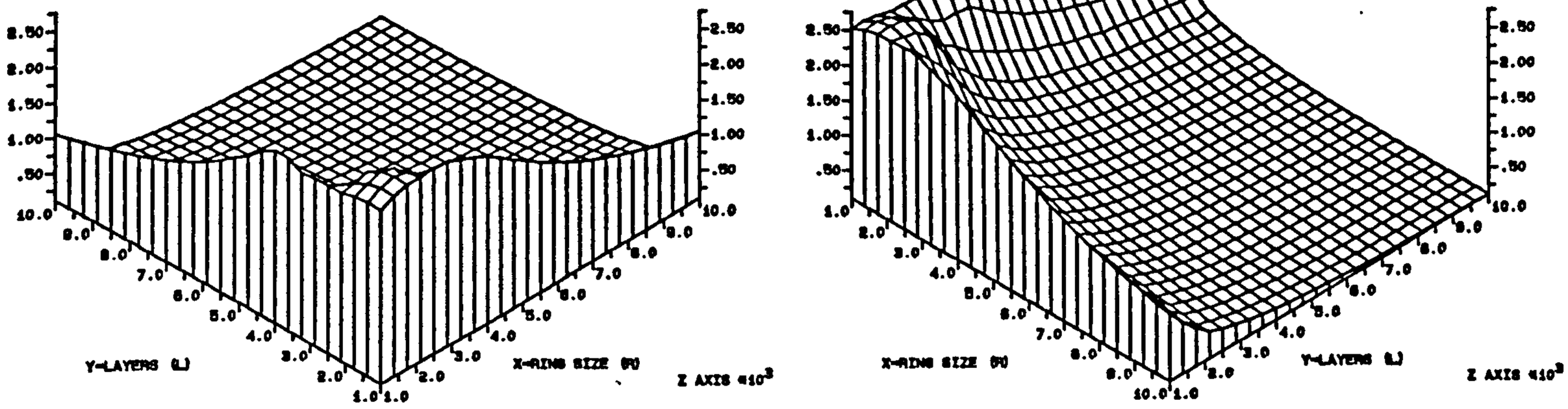
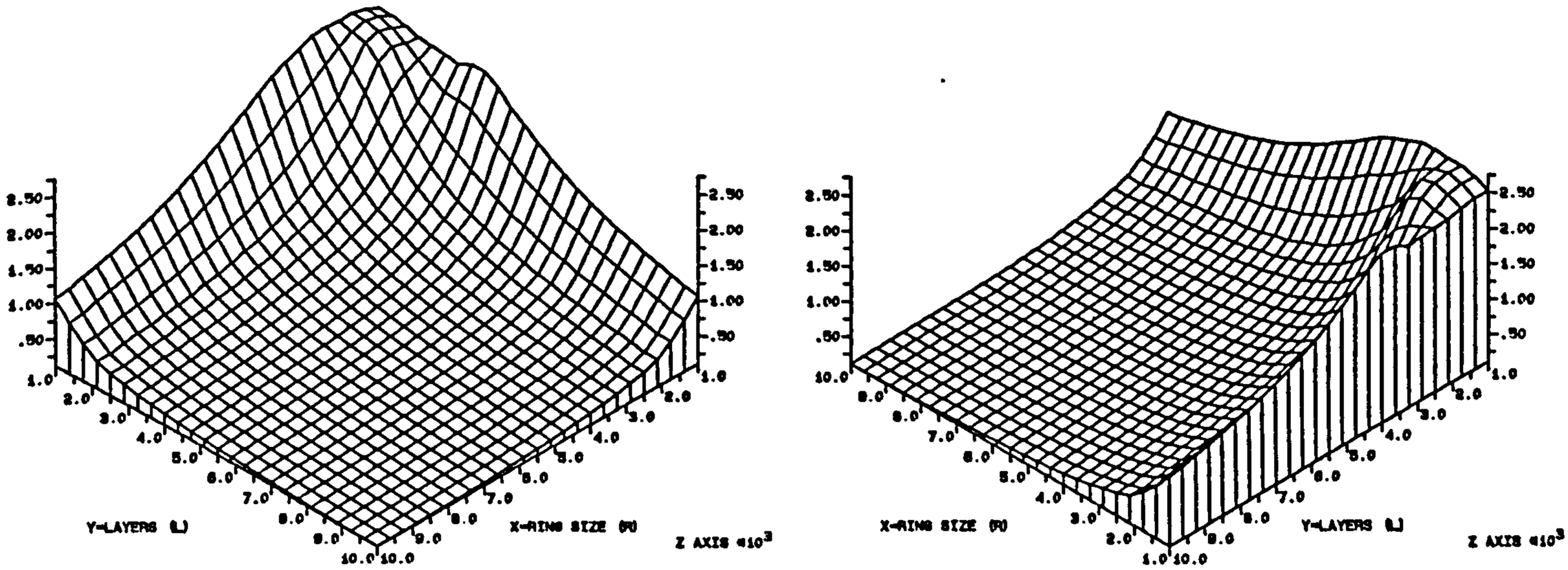
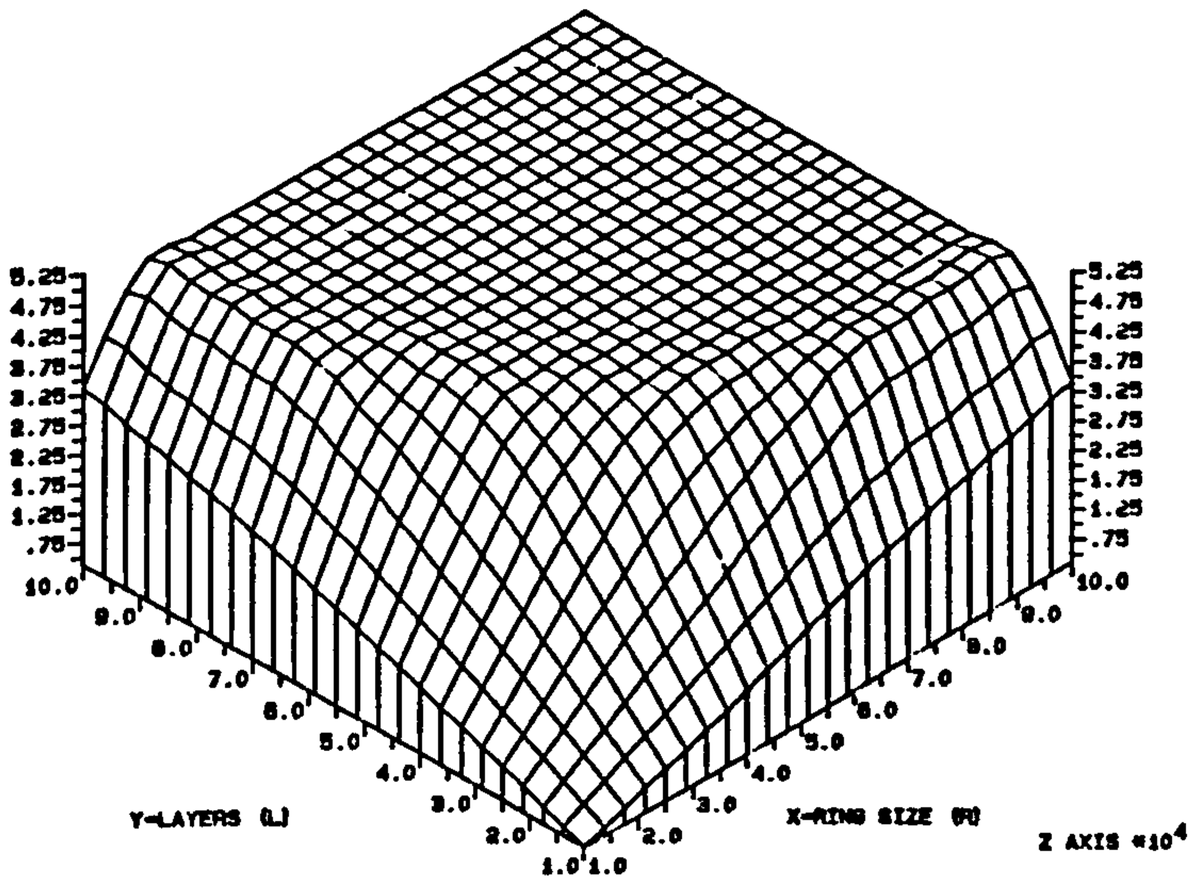


FIG. 5A12.4 HOMOGENEOUS COMMS4 FED AT ONE POINT WITH DATA OF TYPE R20. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





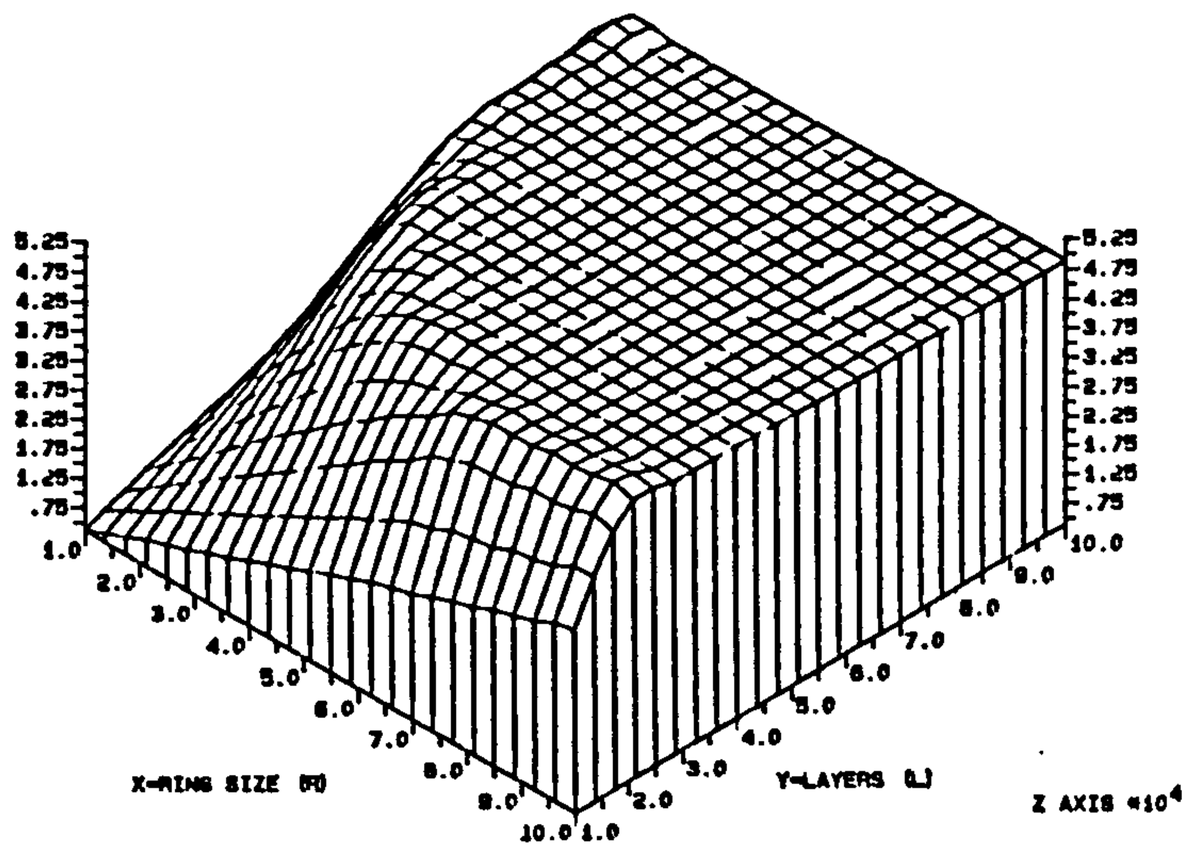
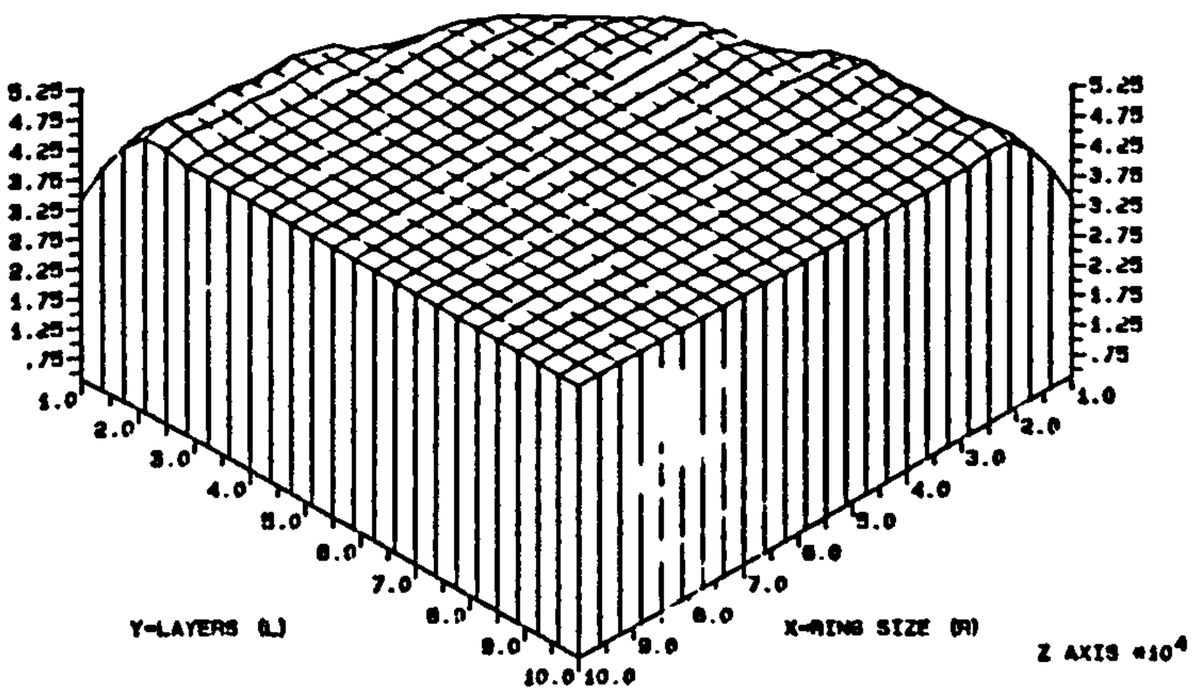
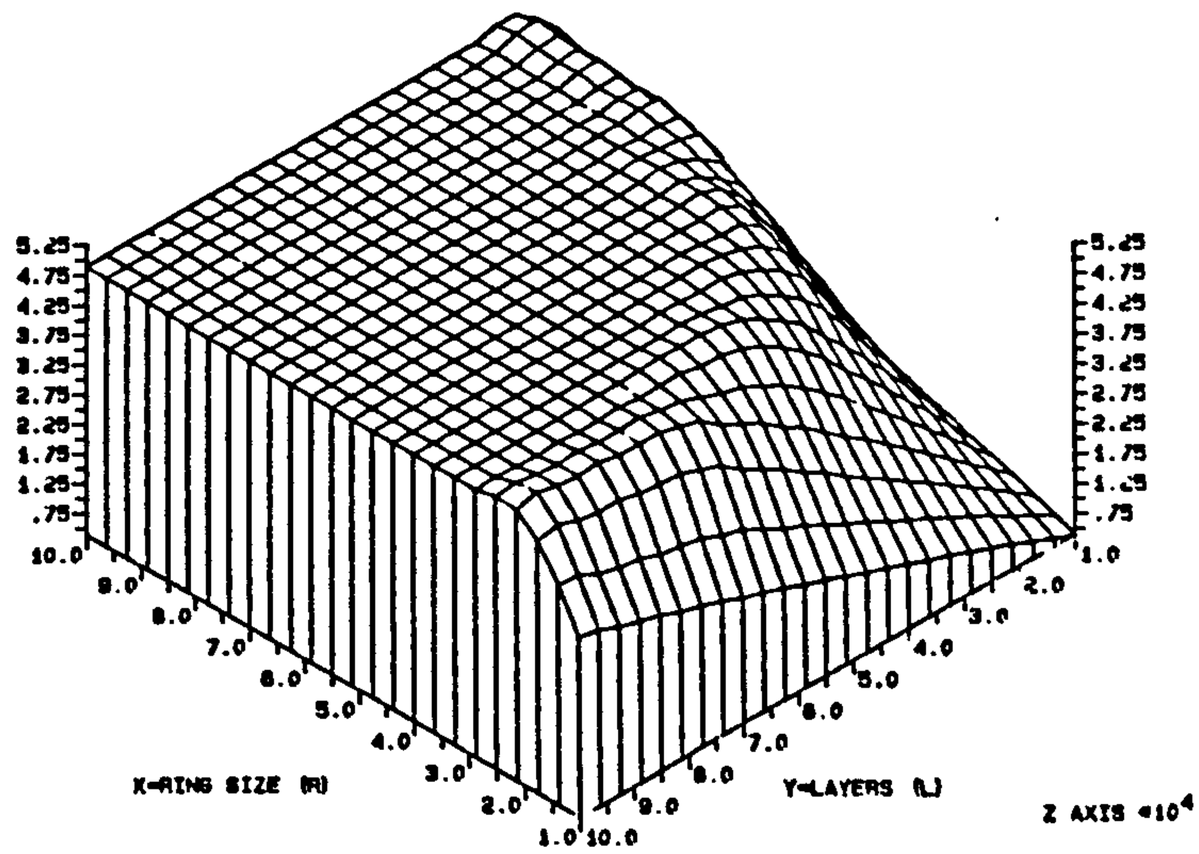


FIG. 5A13.1 HOMOGENEOUS COMMS1 FED AT ONE POINT WITH DATA OF TYPE 50. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





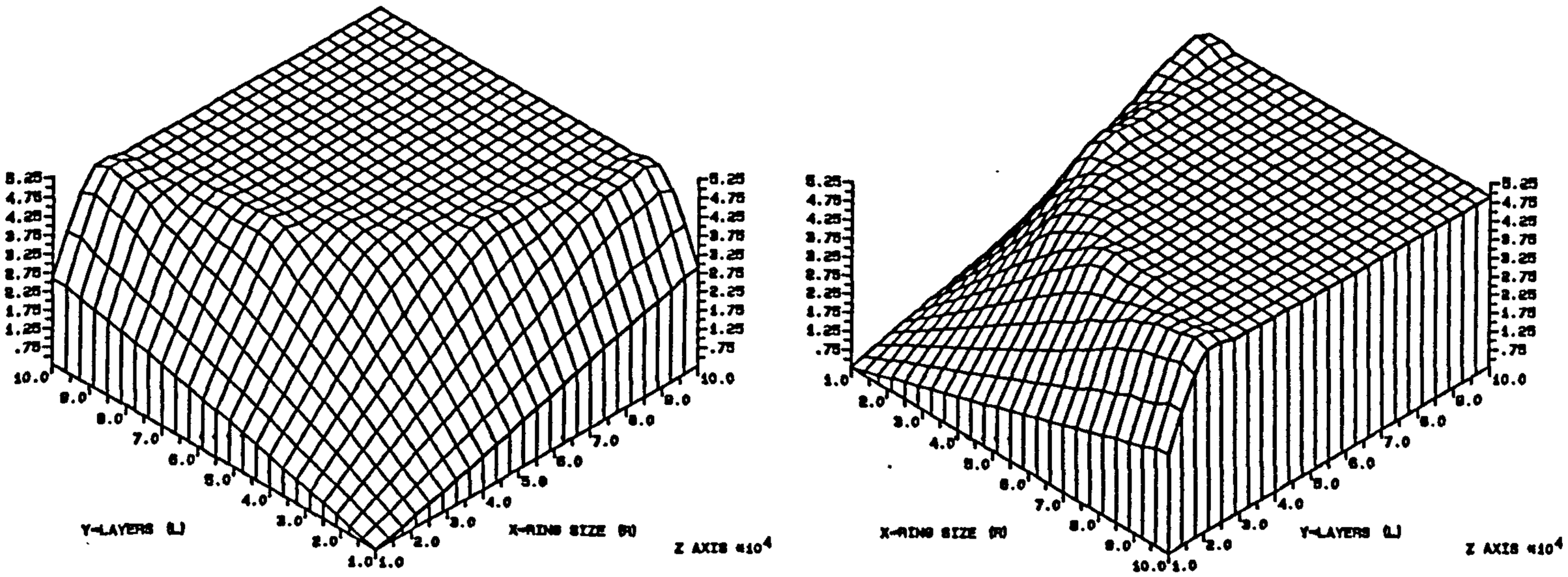
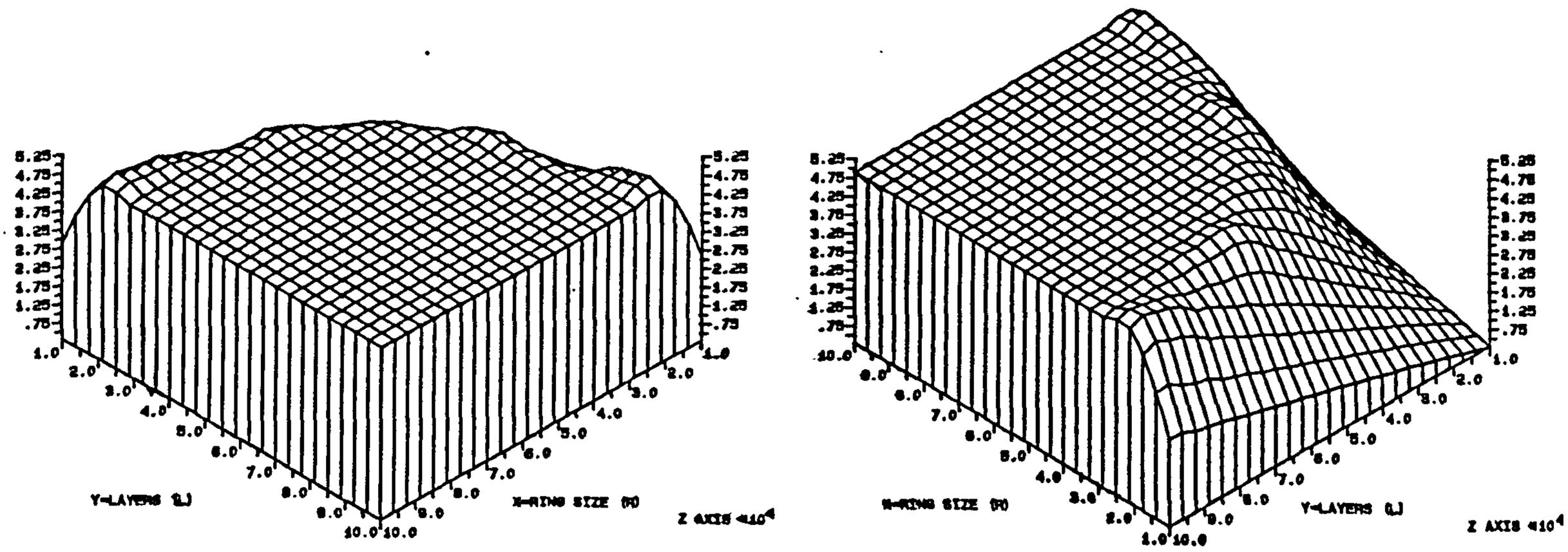


FIG. 5A13.2 HOMOGENEOUS COMMS2 FED AT ONE POINT WITH DATA OF TYPE 50. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.



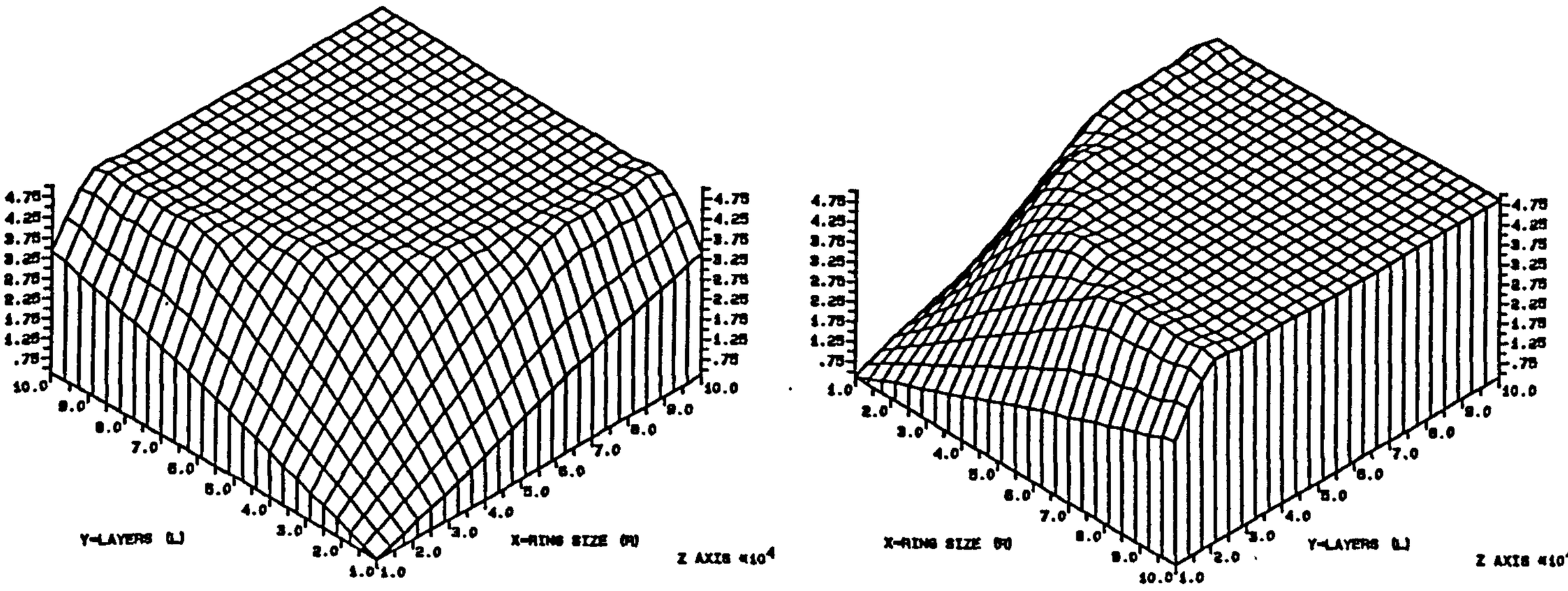
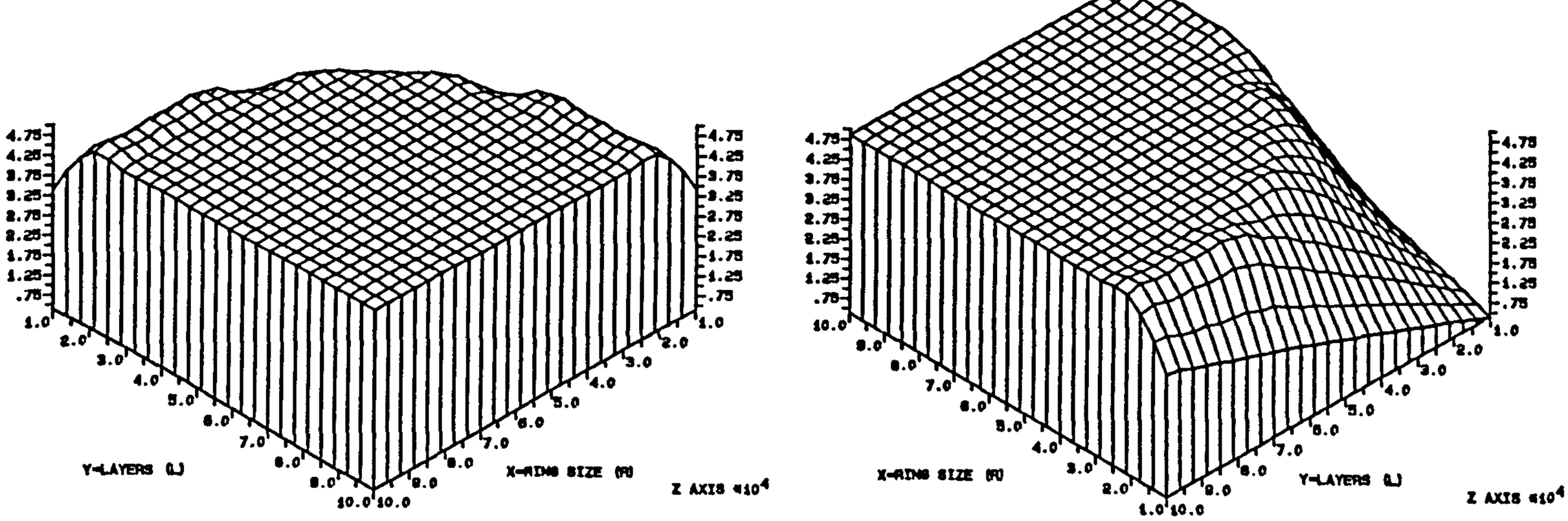


FIG. 5A13.3 HOMOGENEOUS COMMS3 FED AT ONE POINT WITH DATA OF TYPE 50. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.



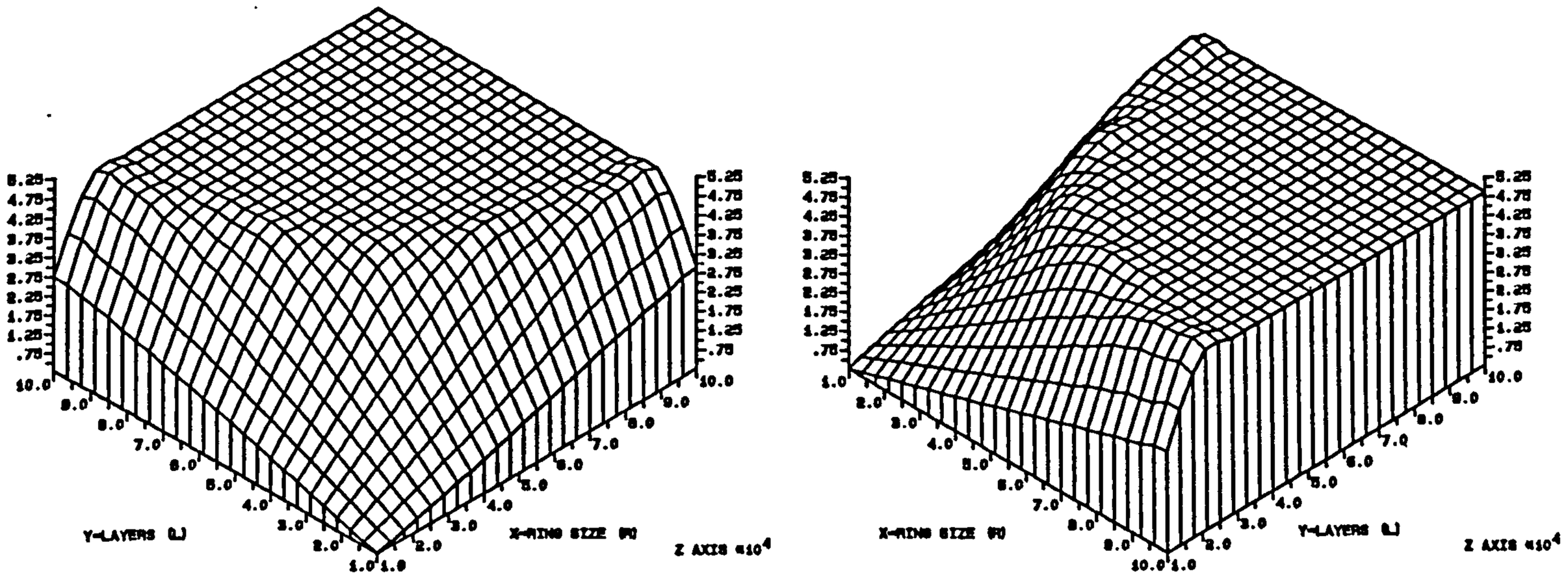
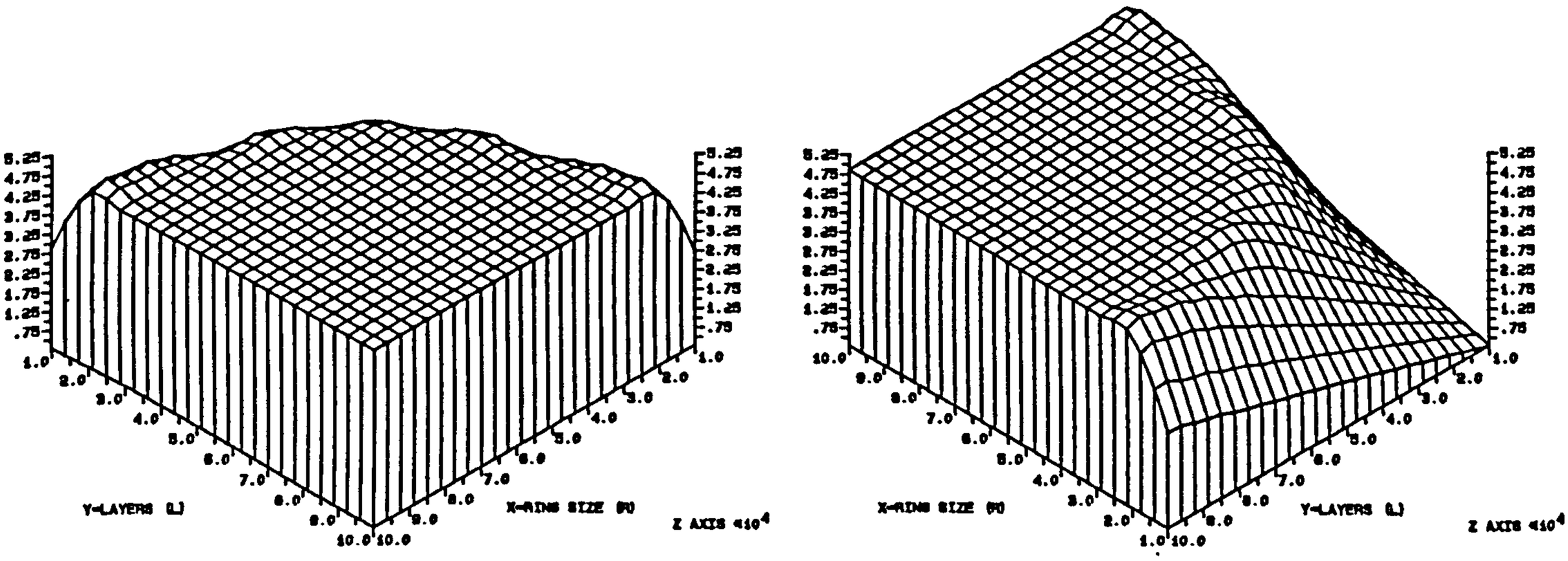


FIG. 5A13.4 HOMOGENEOUS COMMS4 FED AT ONE POINT WITH DATA OF TYPE 50. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.



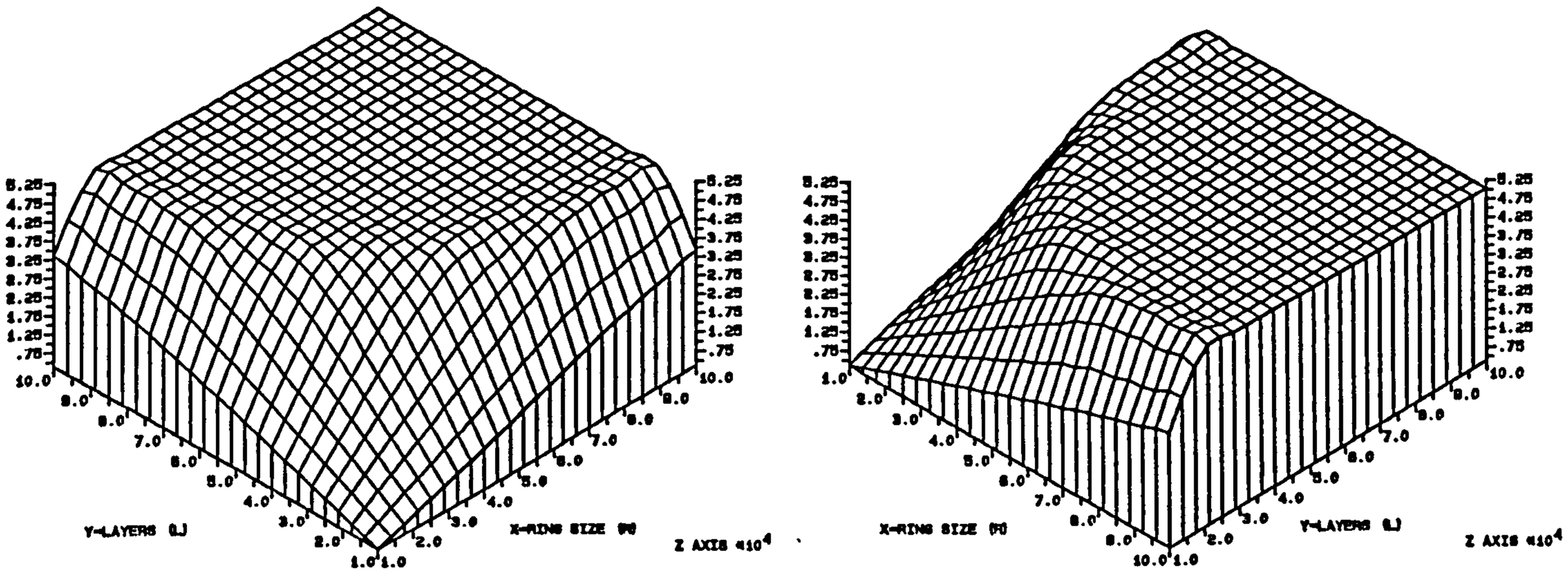
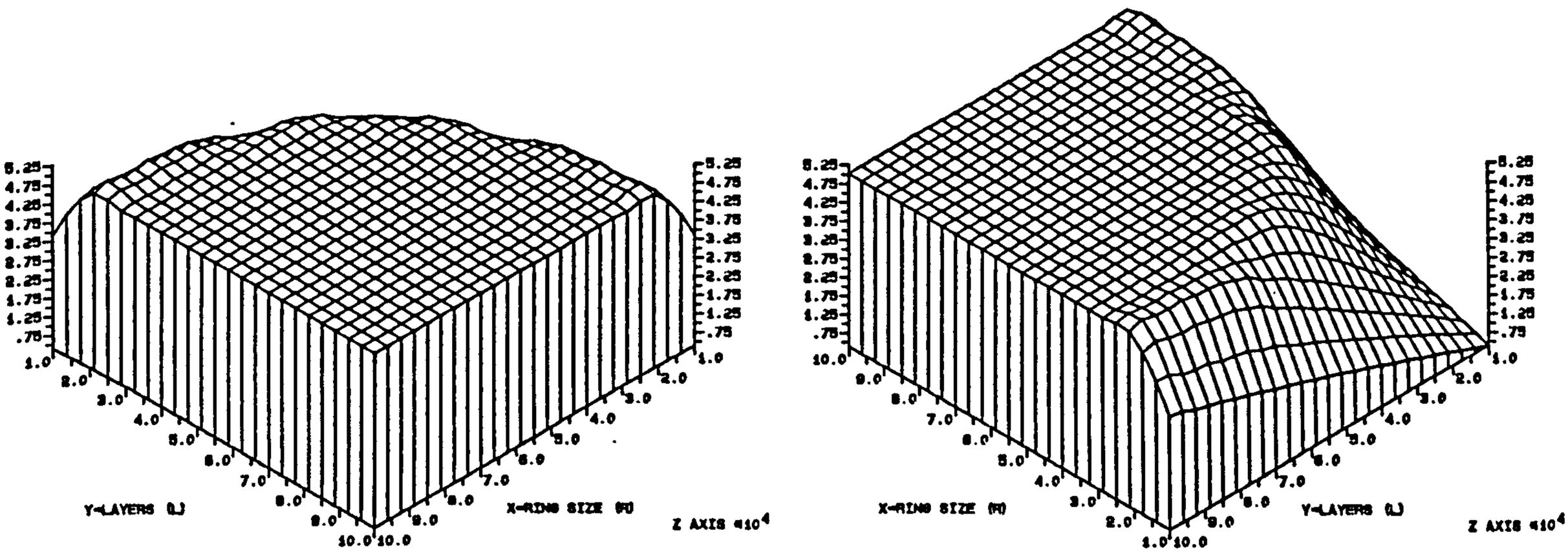


FIG. 5A14.1 HOMOGENEOUS COMMS1 FED AT ONE POINT WITH DATA OF TYPE R100. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.



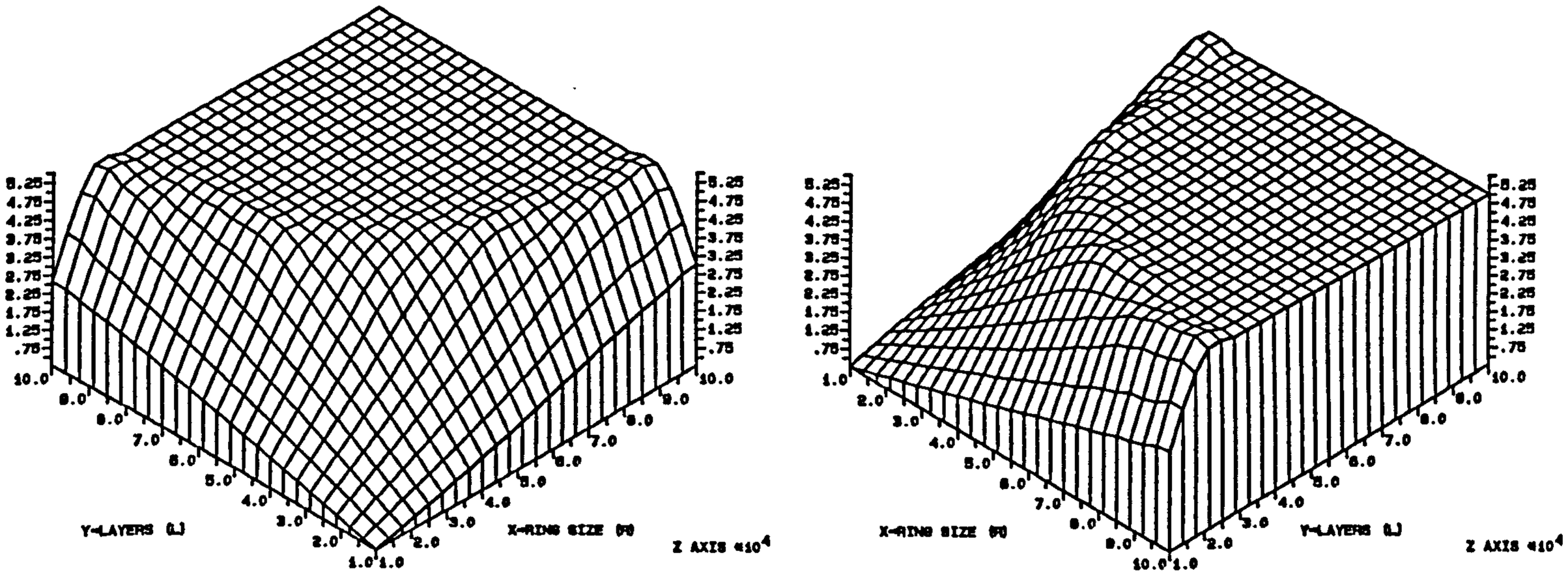
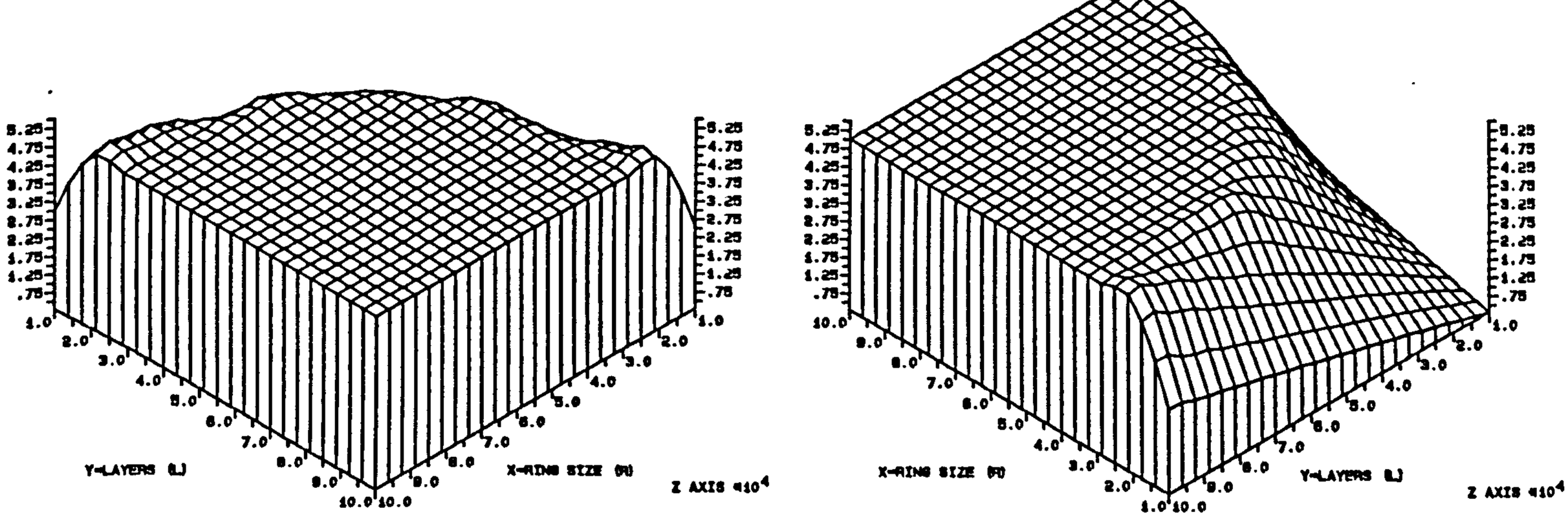
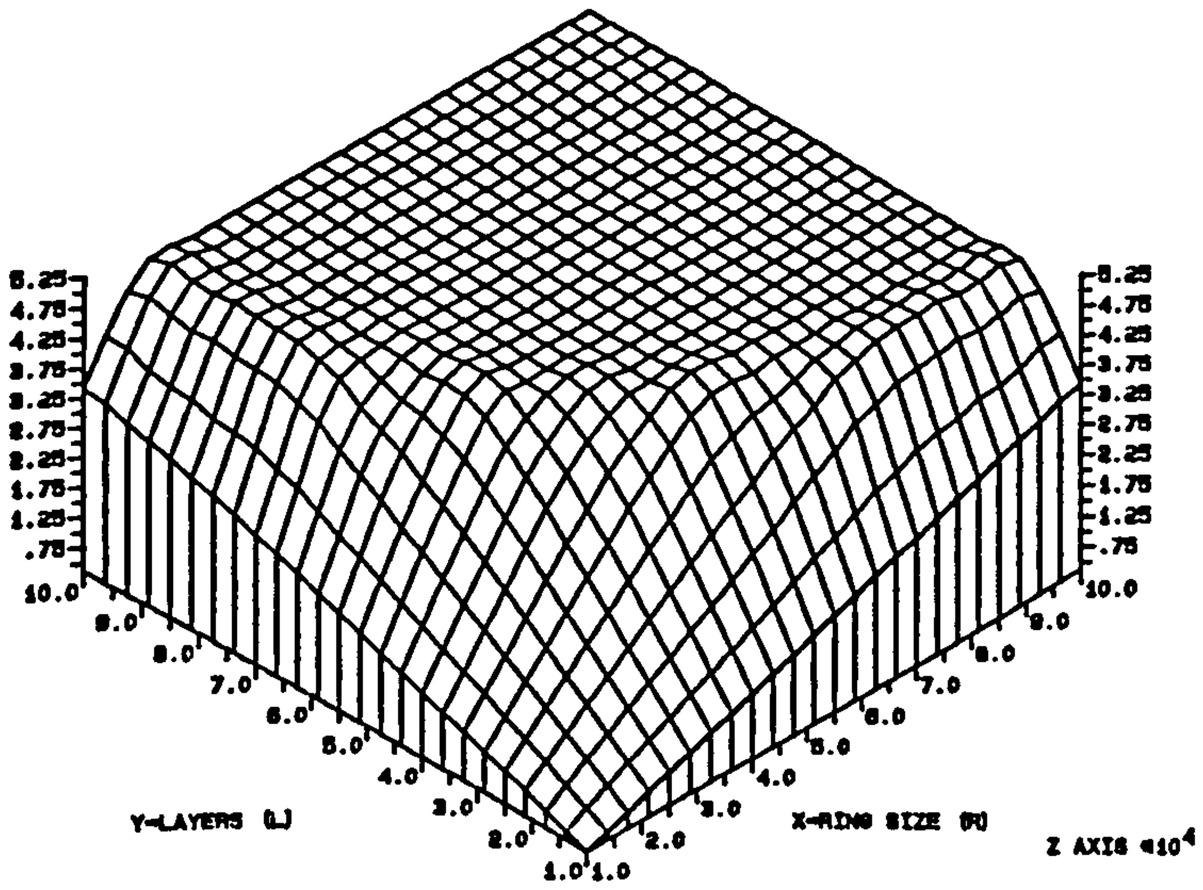


FIG. 5A14.2 HOMOGENEOUS COMMS2 FED AT ONE POINT WITH DATA OF TYPE R100. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.





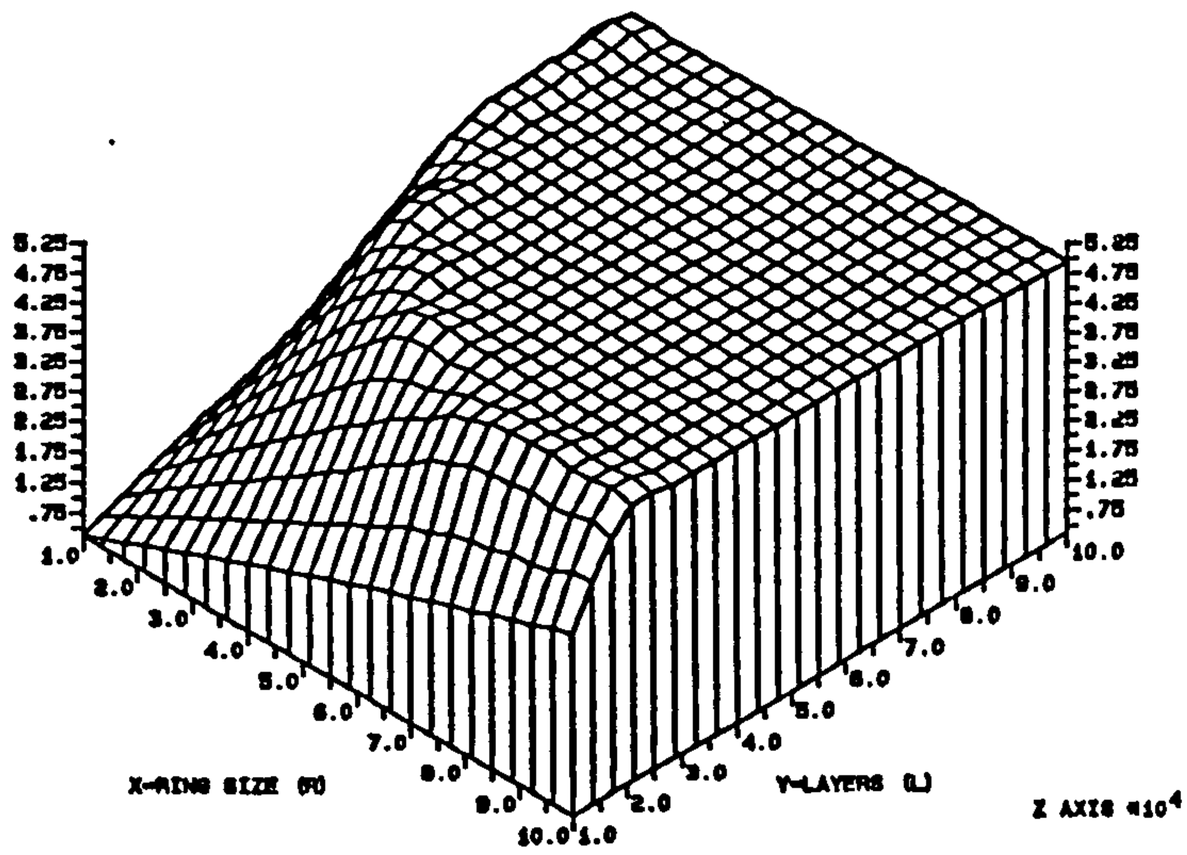
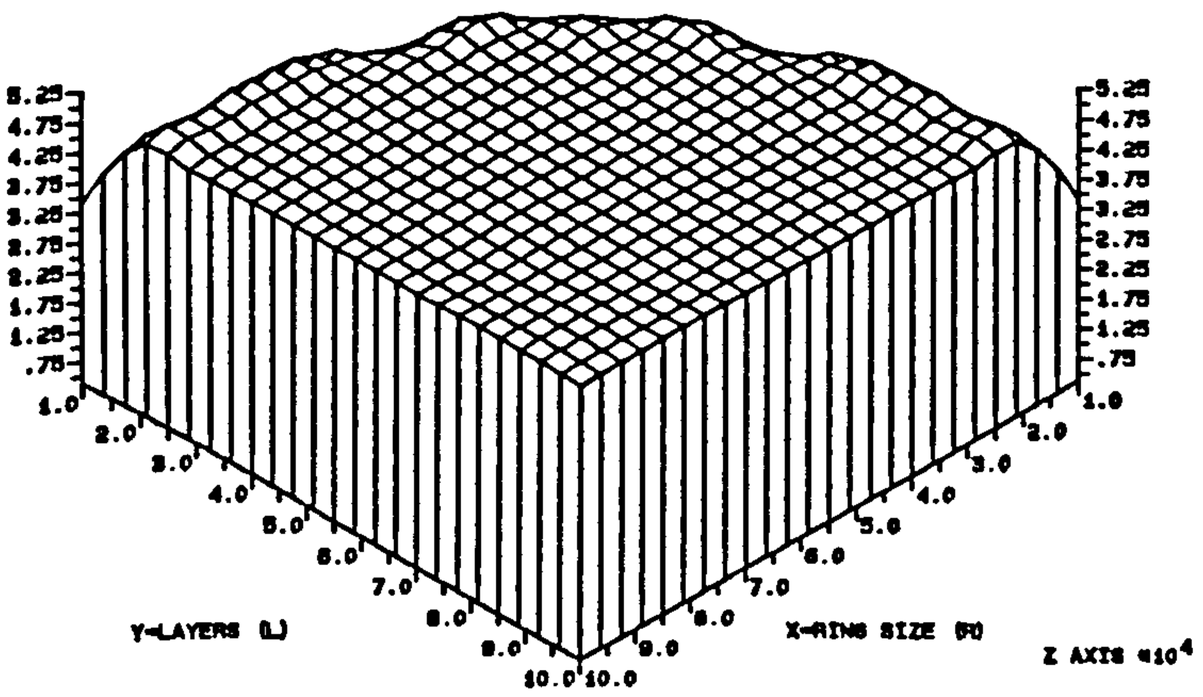
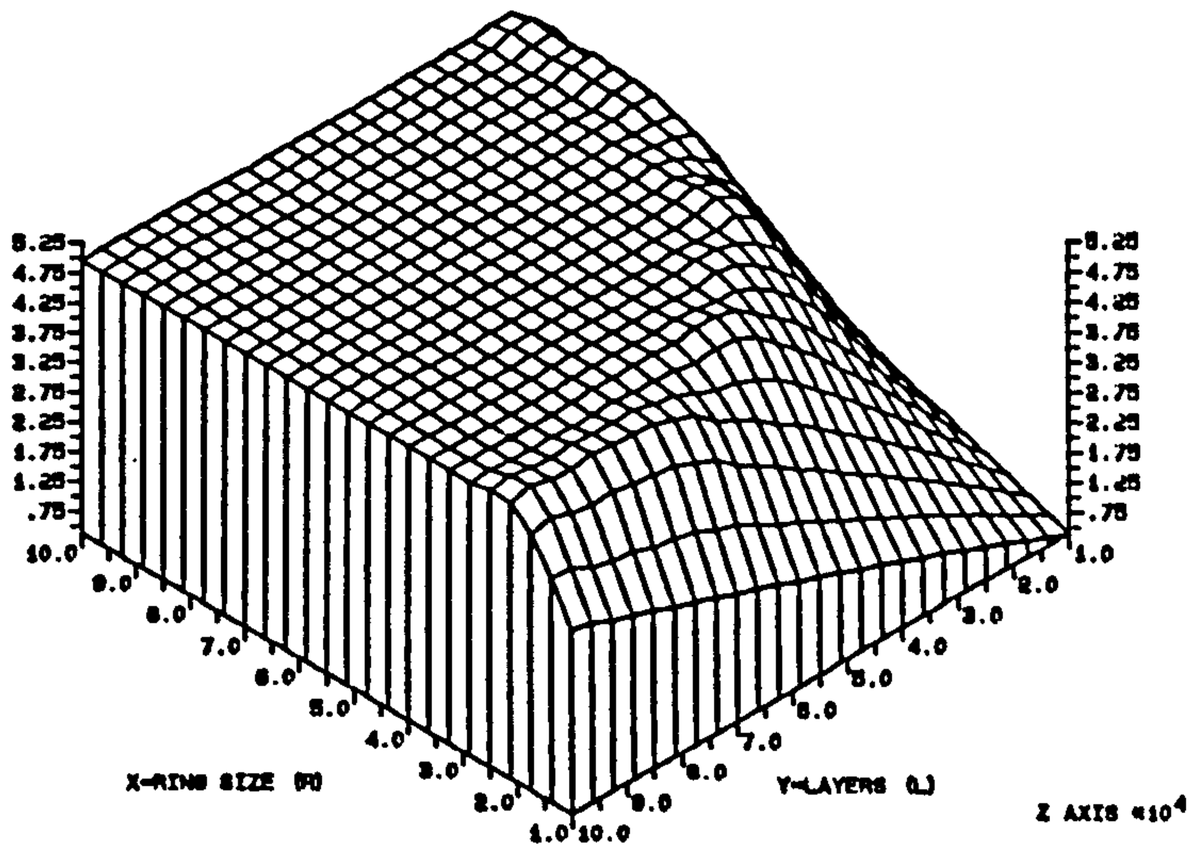
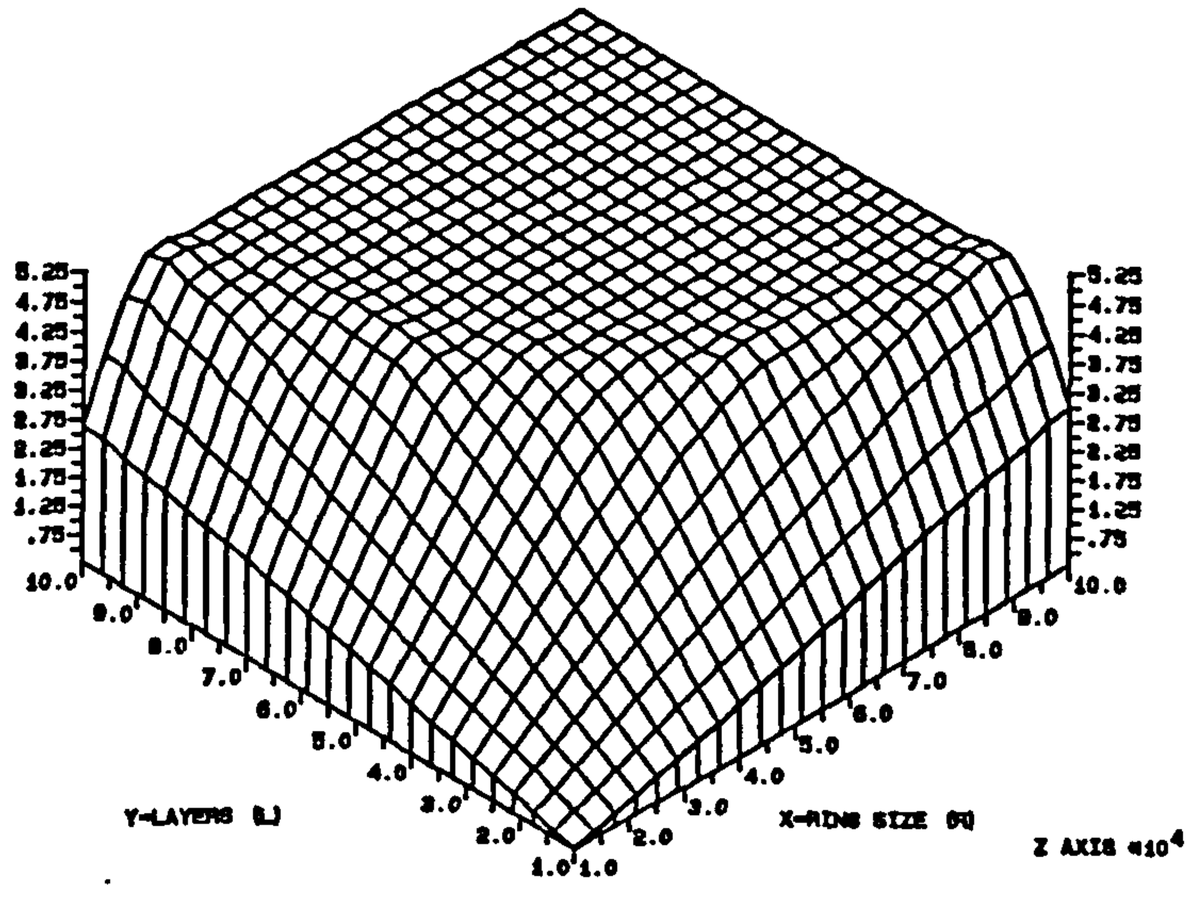


FIG. 5A14.3 HOMOGENEOUS COMMS3 FED AT ONE POINT WITH DATA OF TYPE R100. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.







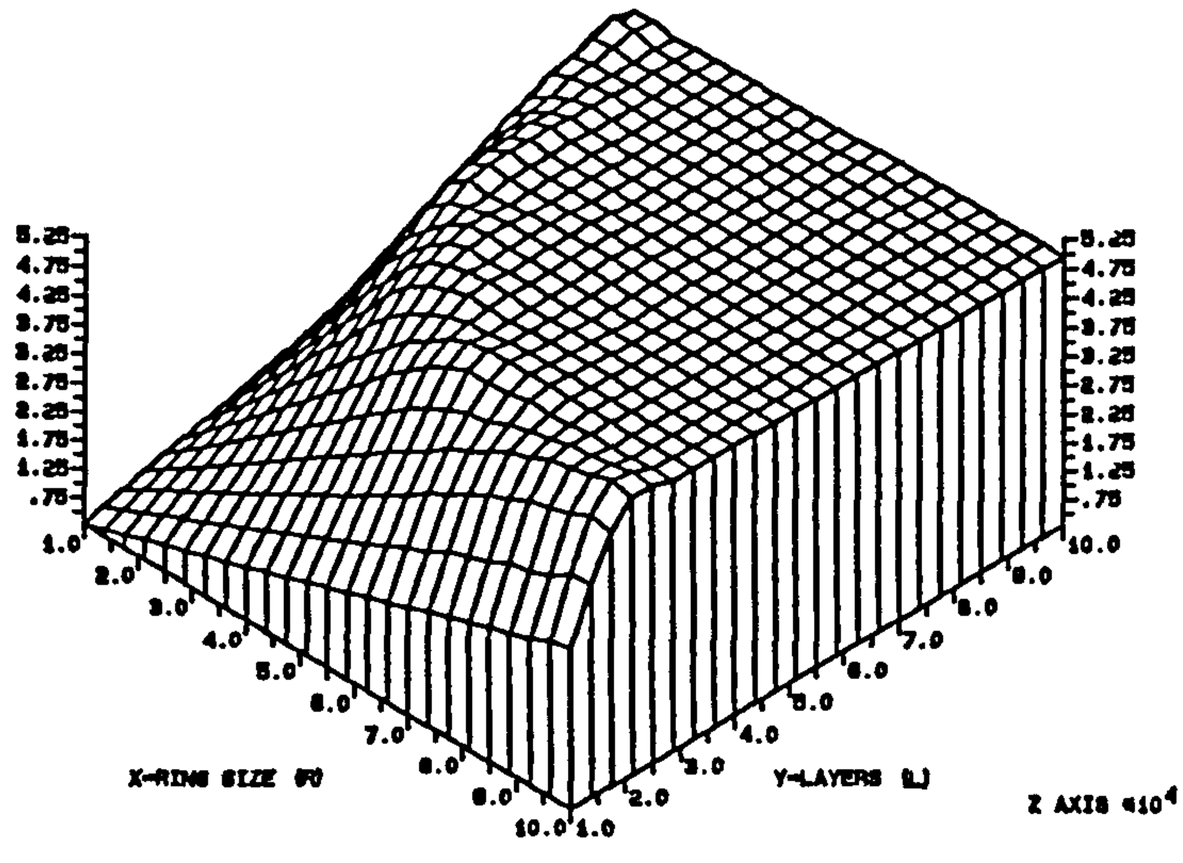
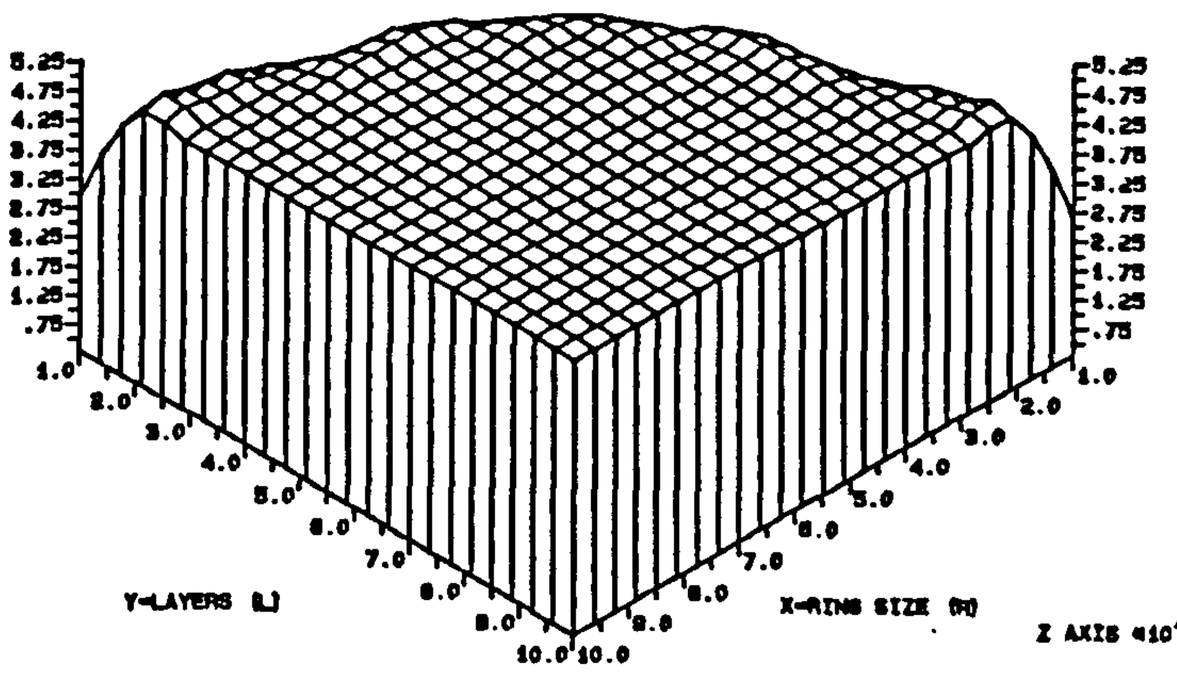
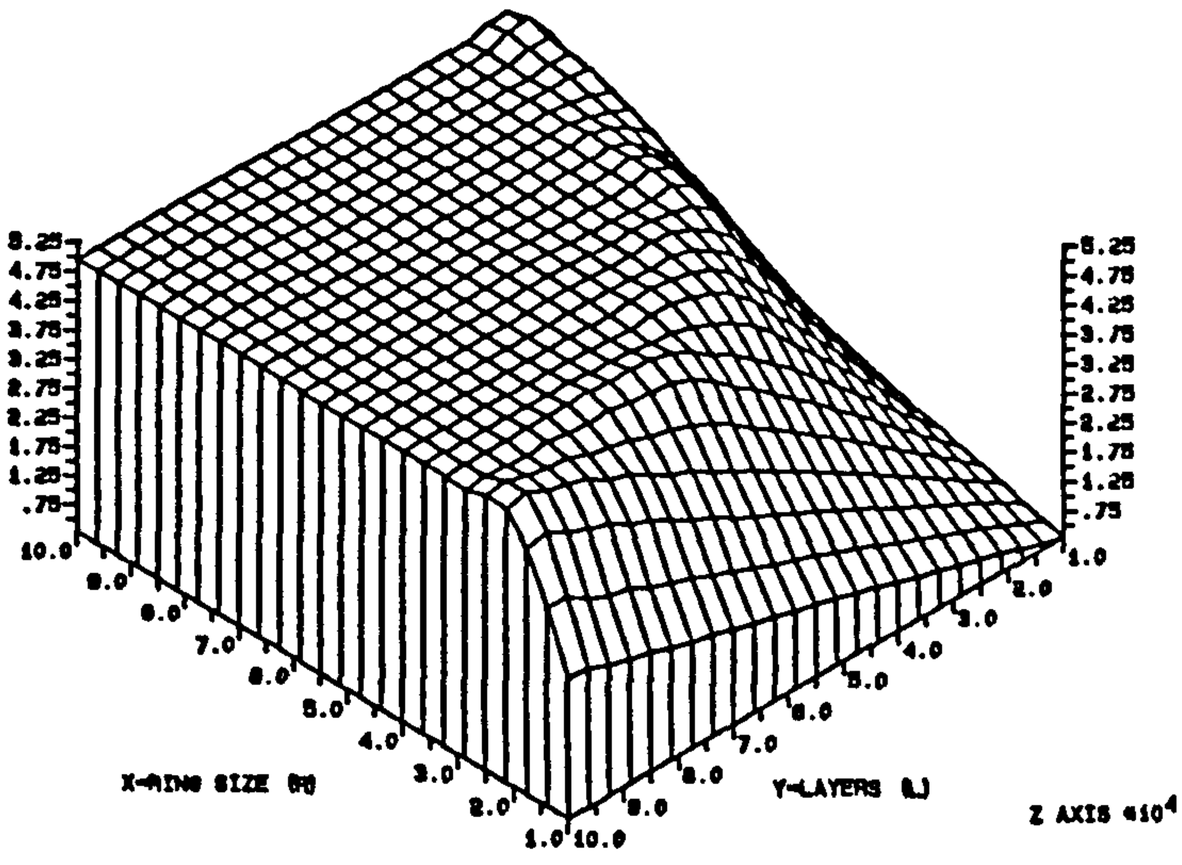
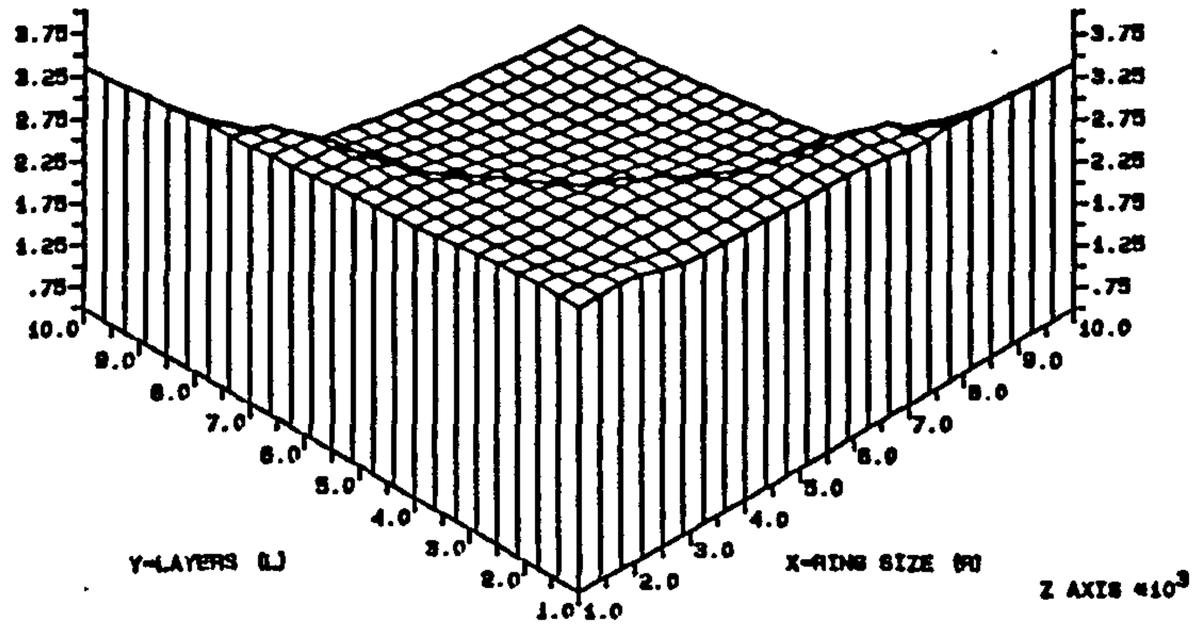


FIG. 5A14.4 HOMOGENEOUS COMMS4 FED AT ONE POINT WITH DATA OF TYPE R100. Z=WEIGHTED TOTAL PROCESSING OVER 1000 ITERATIONS.







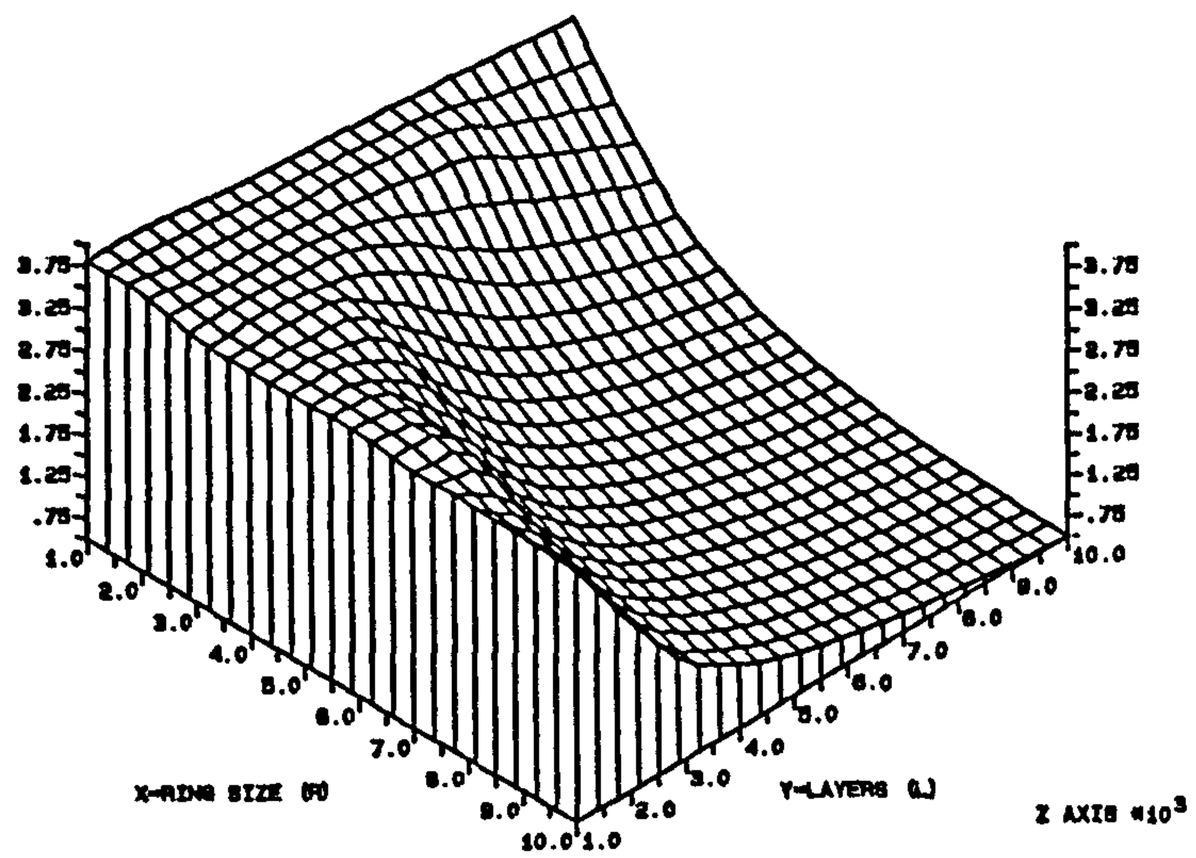
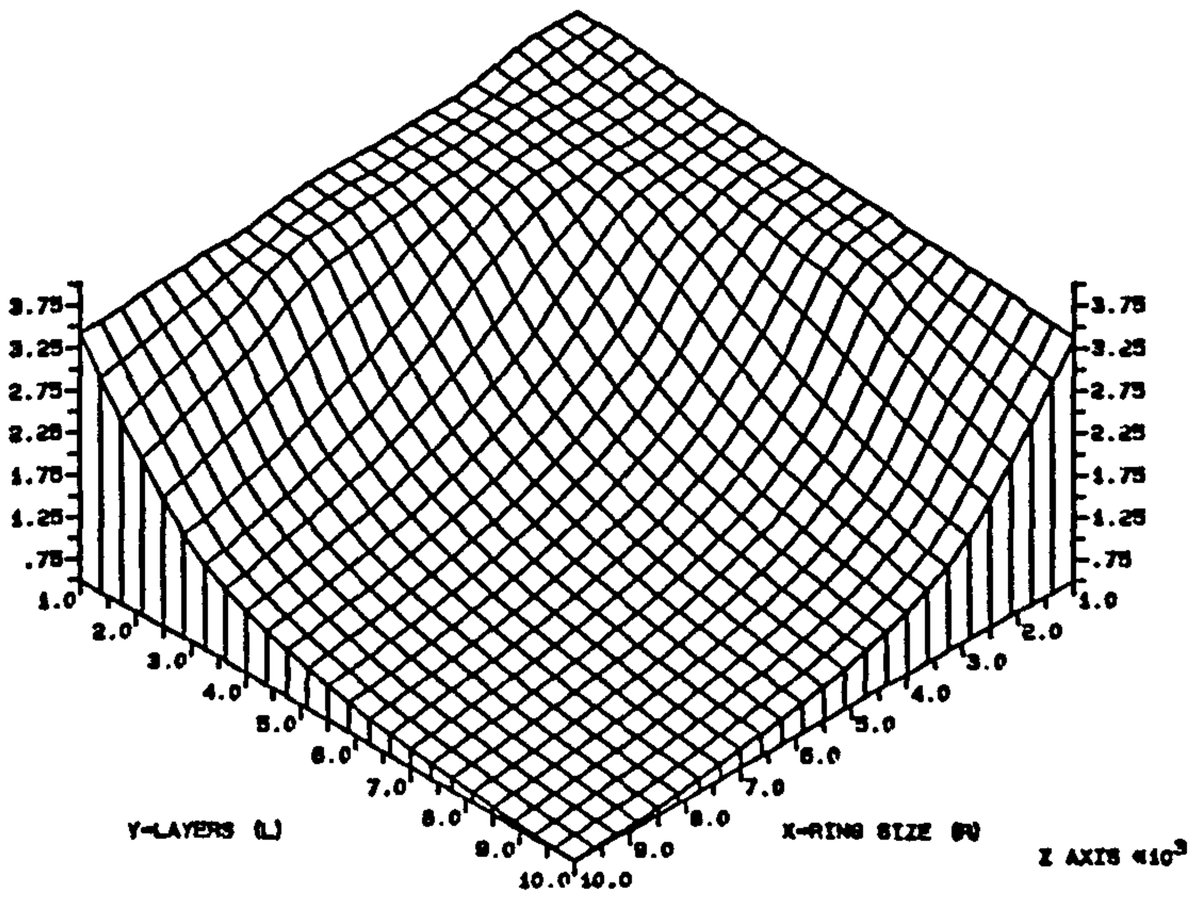
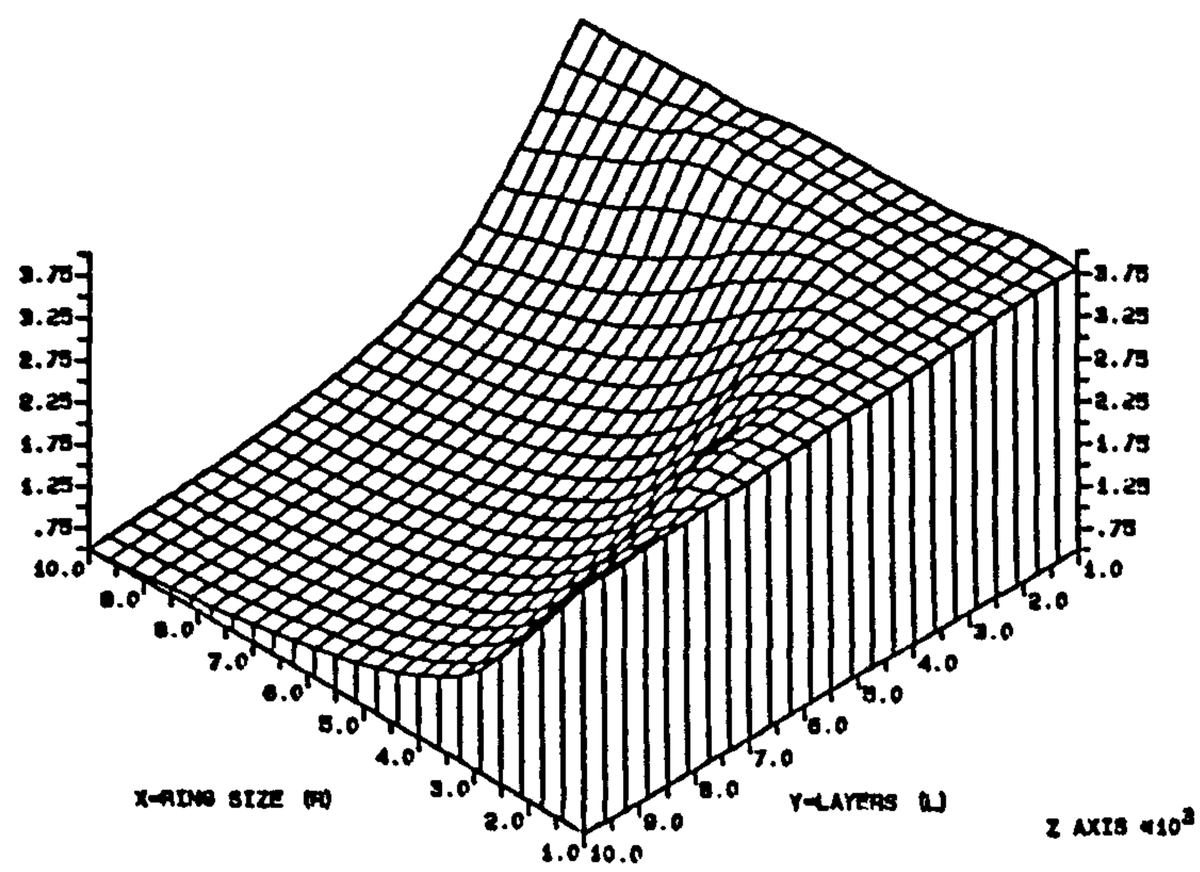


FIG. 5A15.1 HOMOGENEOUS COMMS1 FED AT ONE POINT WITH DATA OF TYPE 50. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





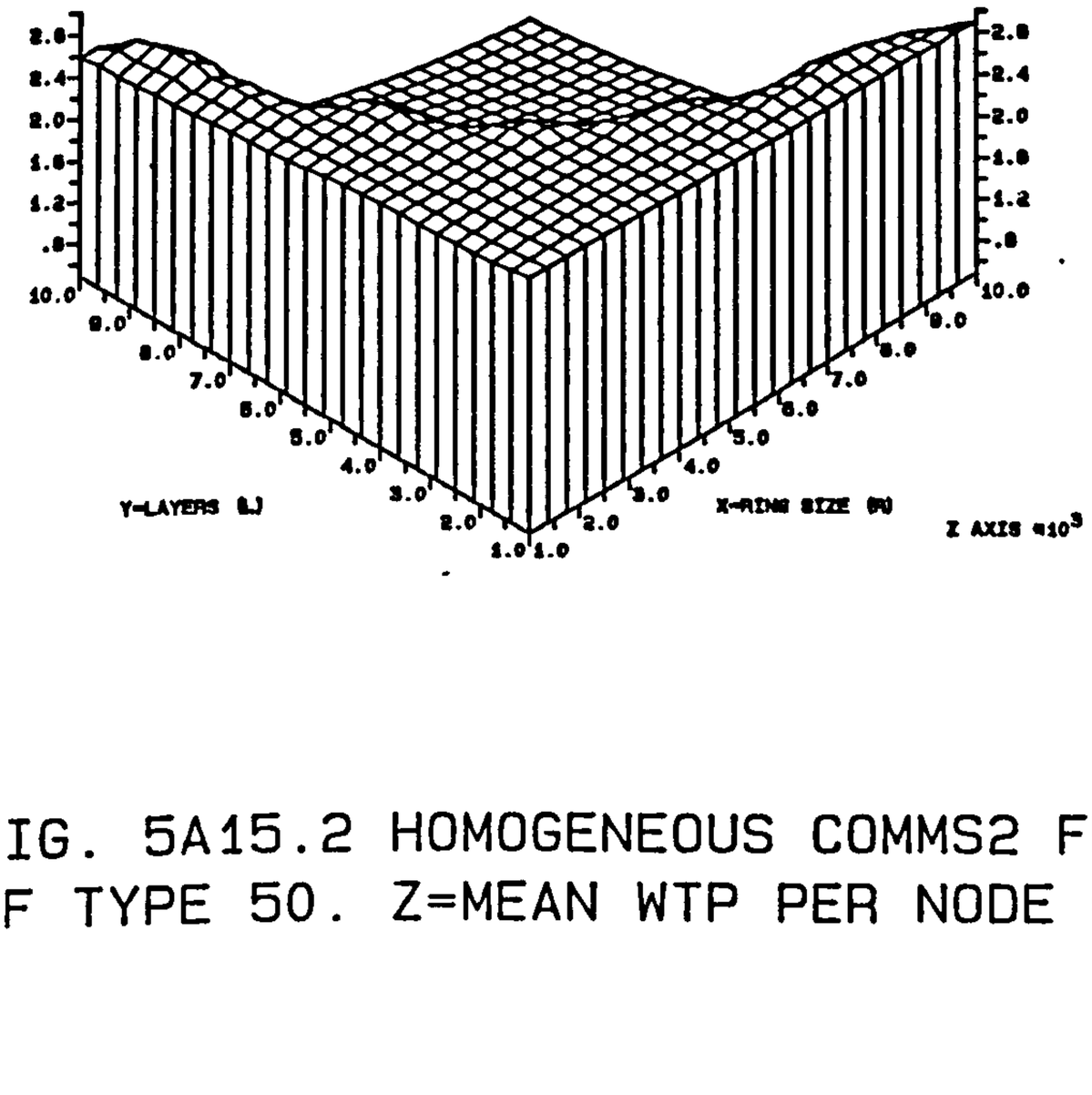
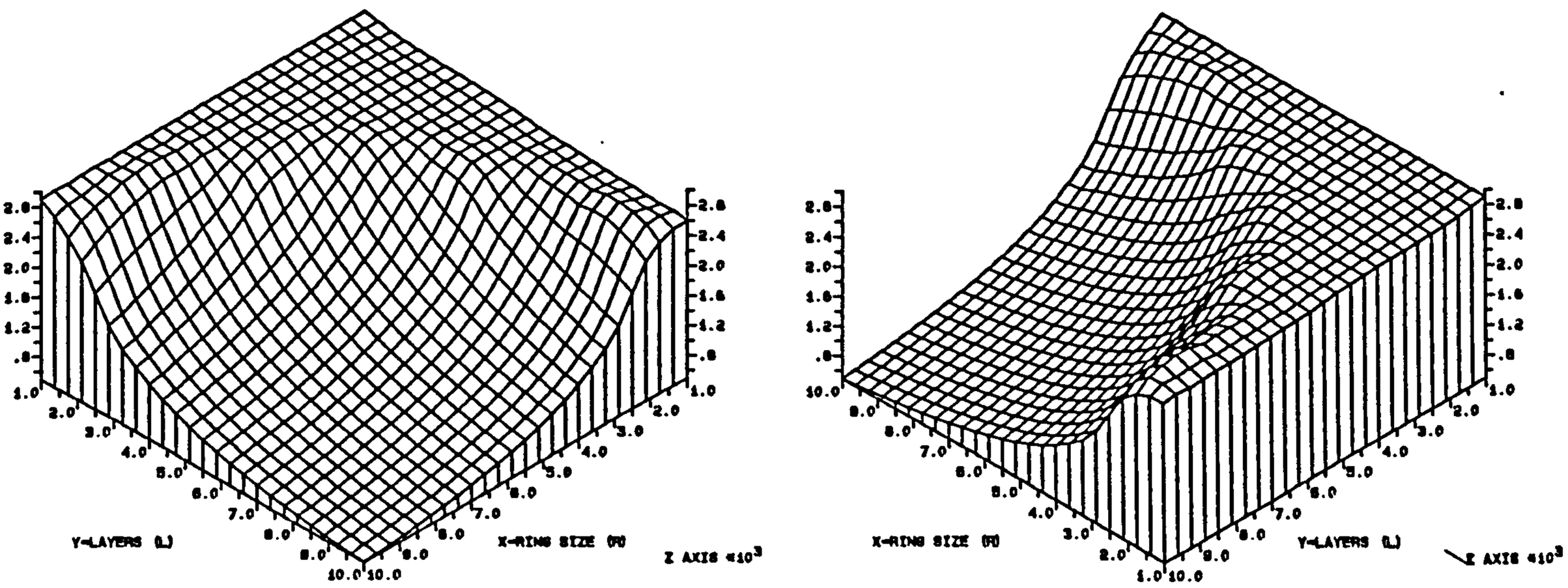
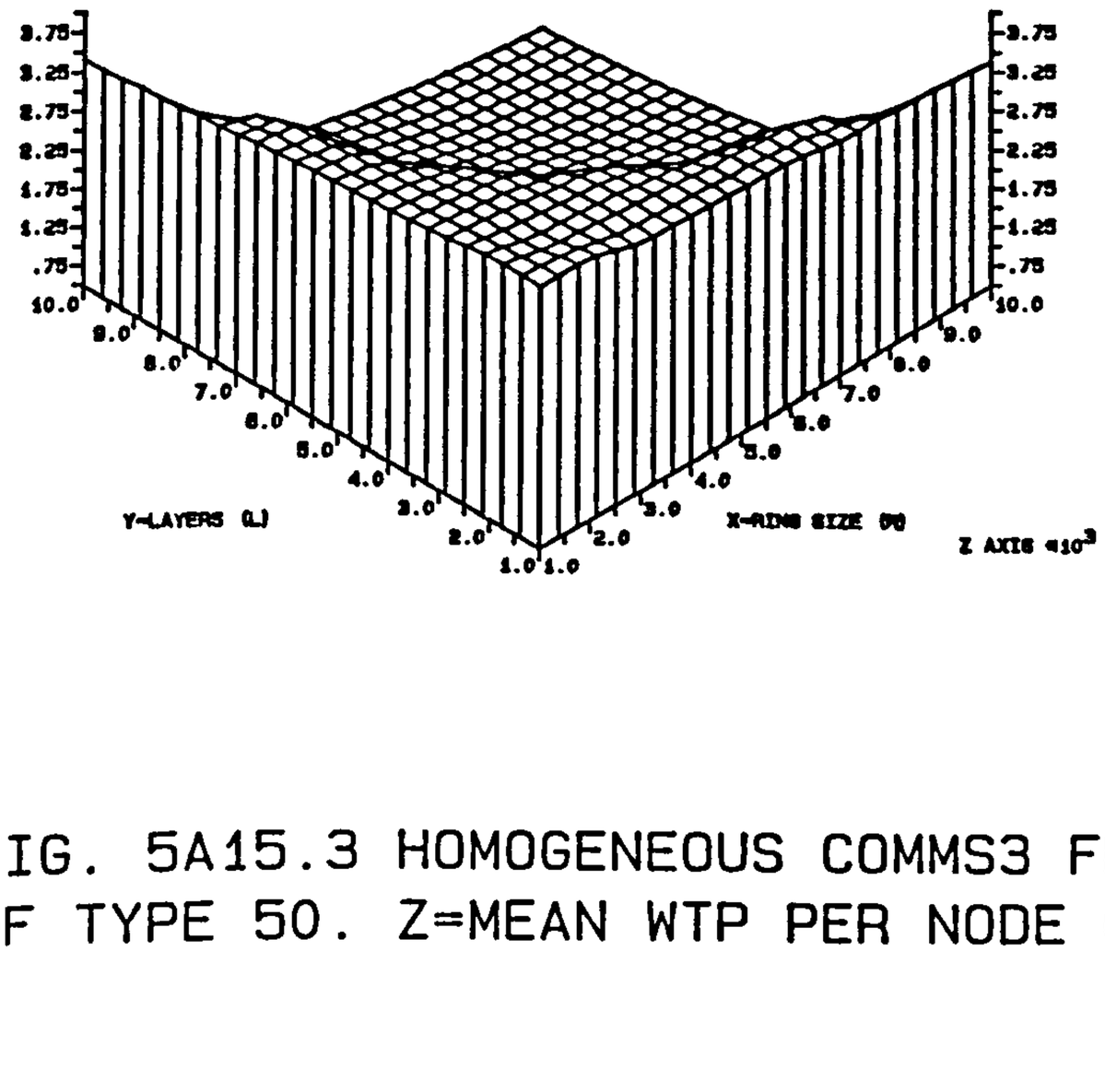


FIG. 5A15.2 HOMOGENEOUS COMMS2 FED AT ONE POINT WITH DATA OF TYPE 50. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





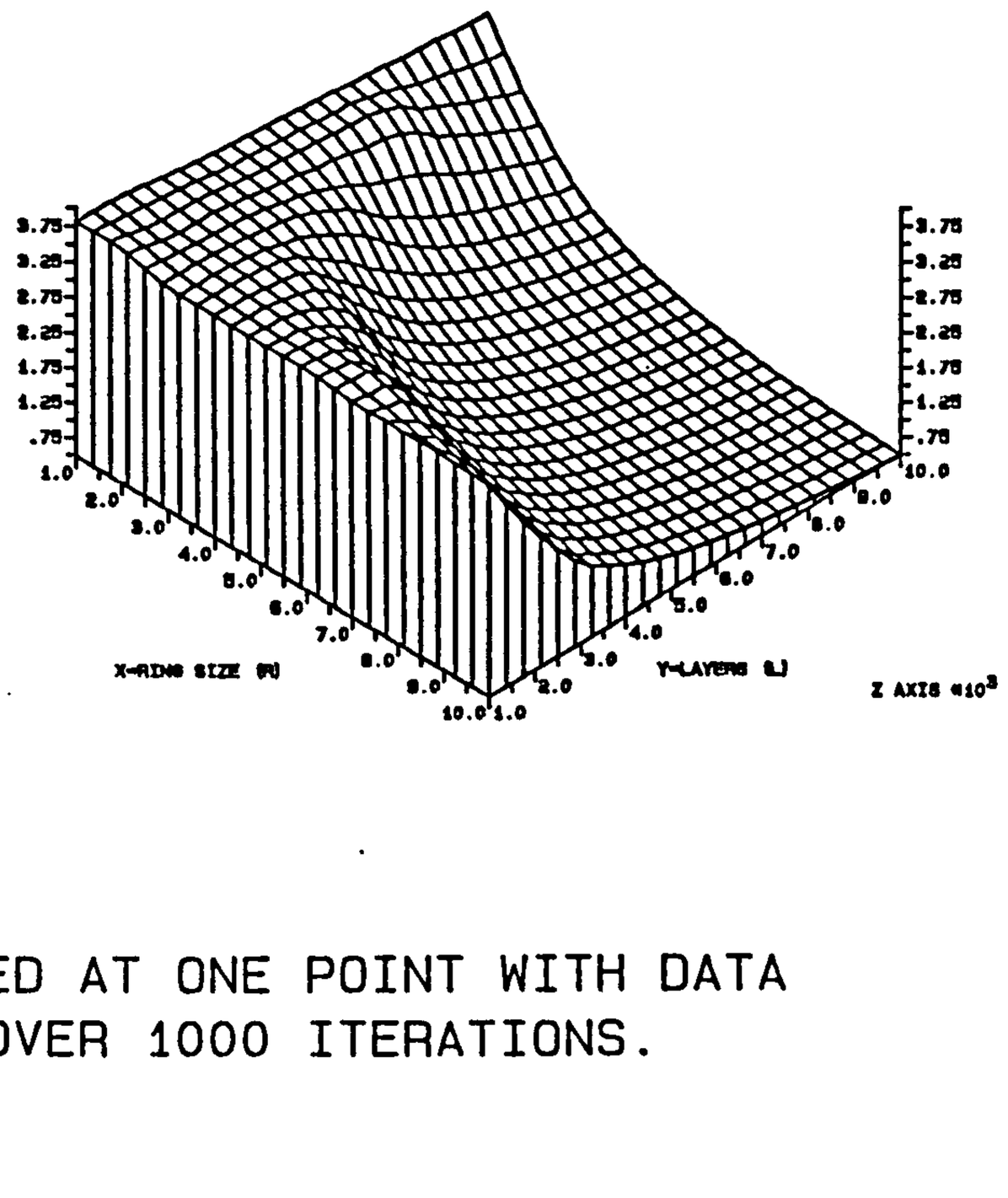
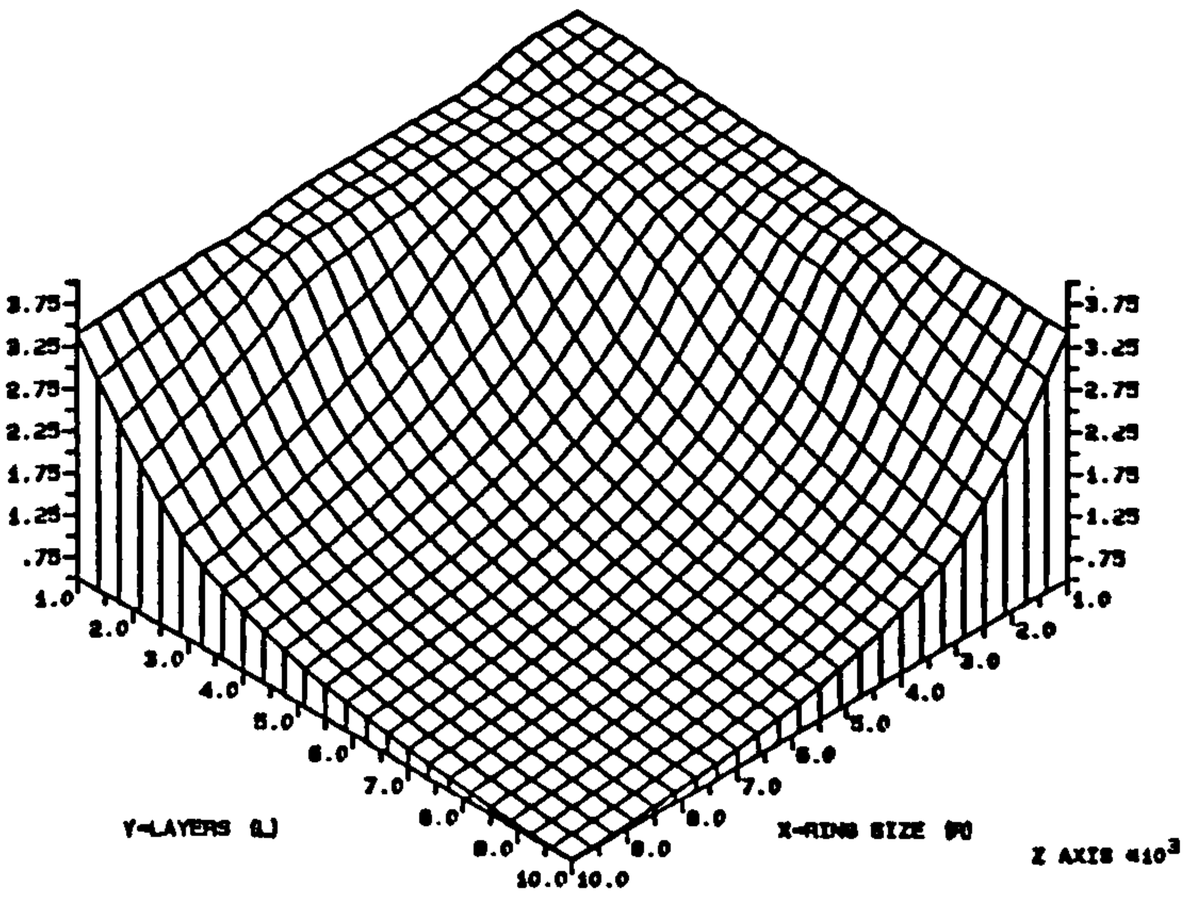
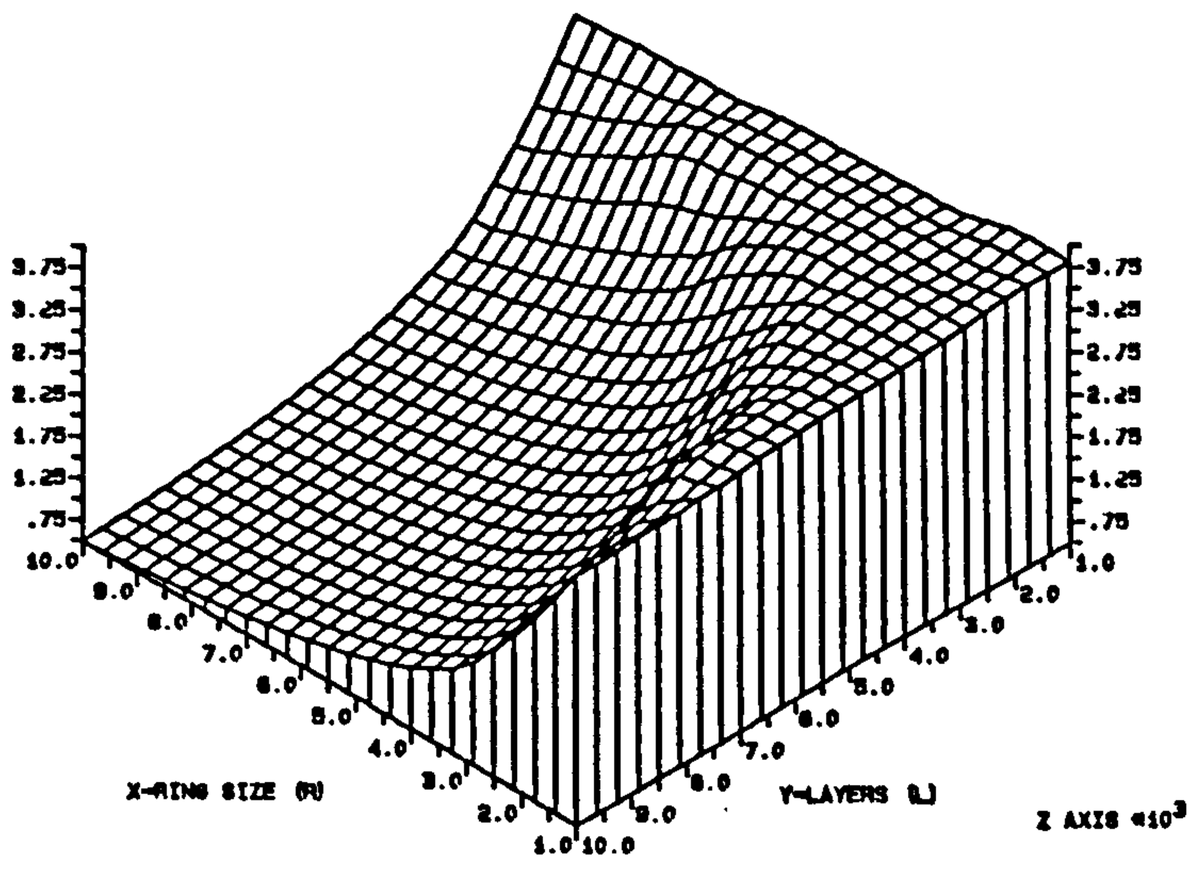


FIG. 5A15.3 HOMOGENEOUS COMMS3 FED AT ONE POINT WITH DATA OF TYPE 50. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





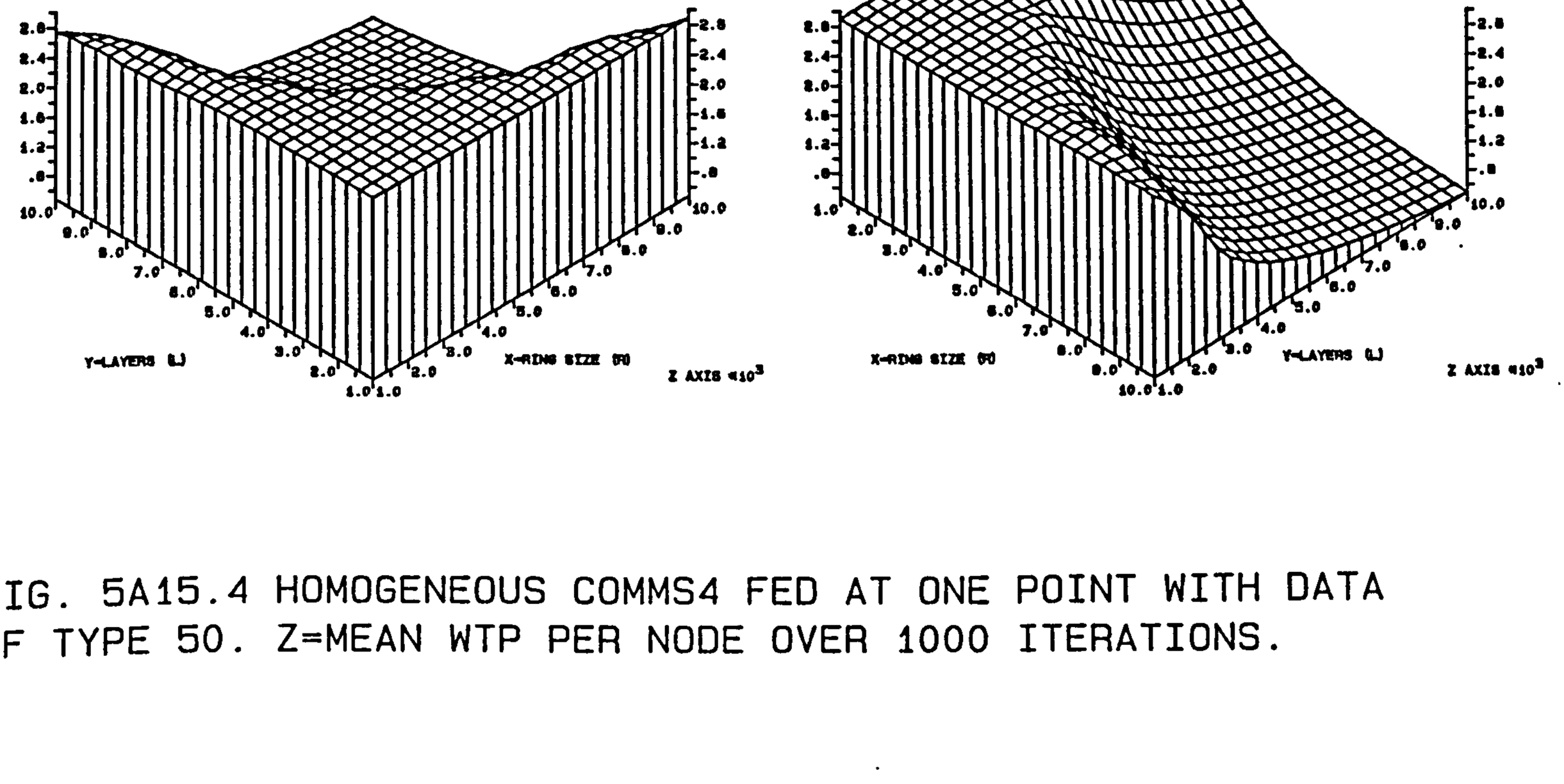
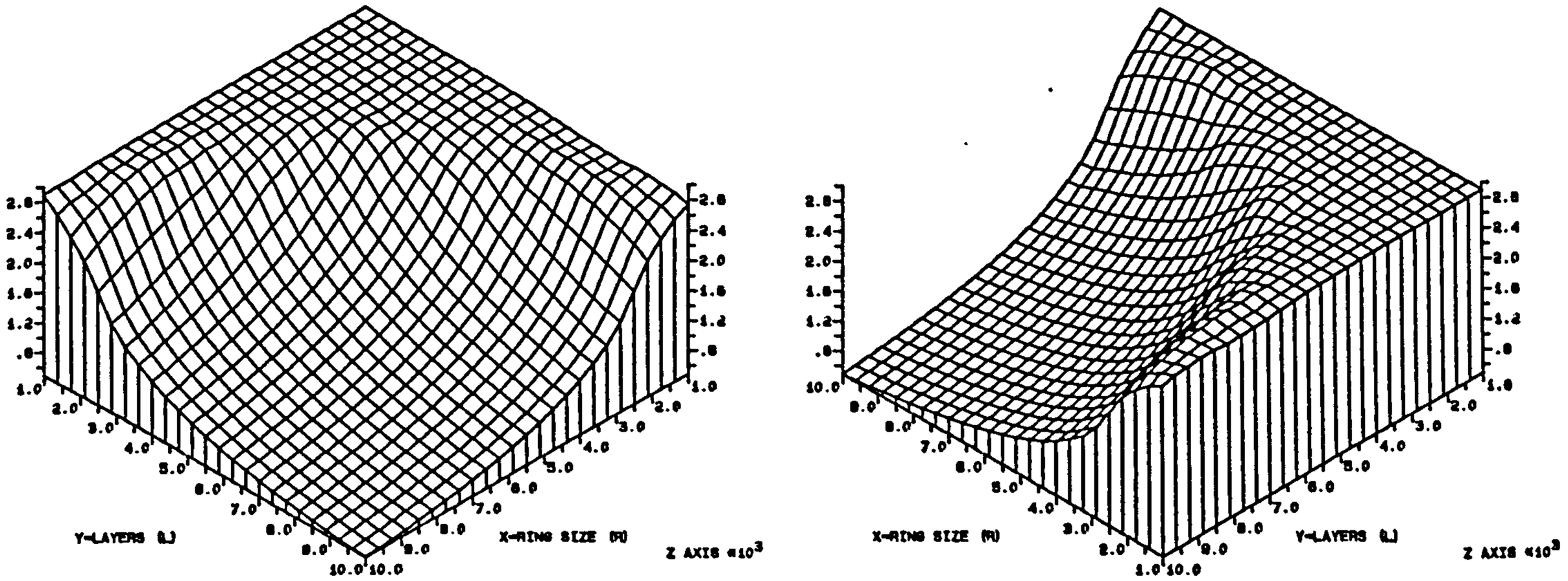
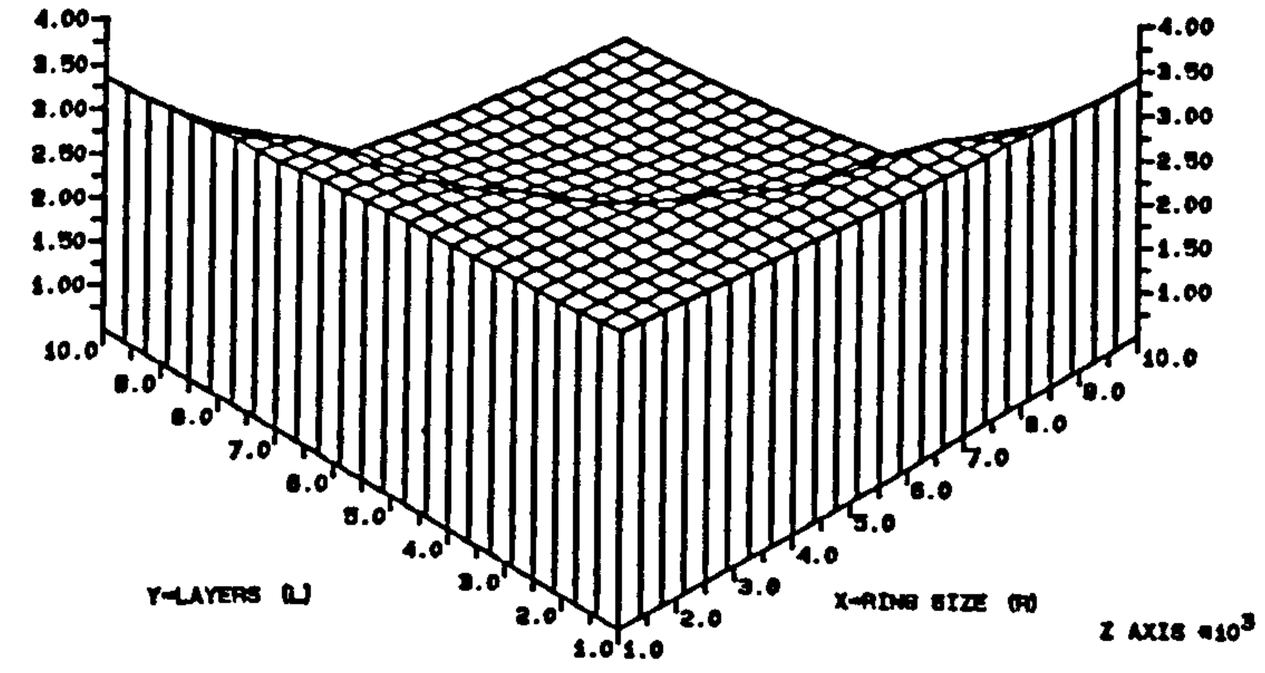


FIG. 5A15.4 HOMOGENEOUS COMMS4 FED AT ONE POINT WITH DATA OF TYPE 50. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





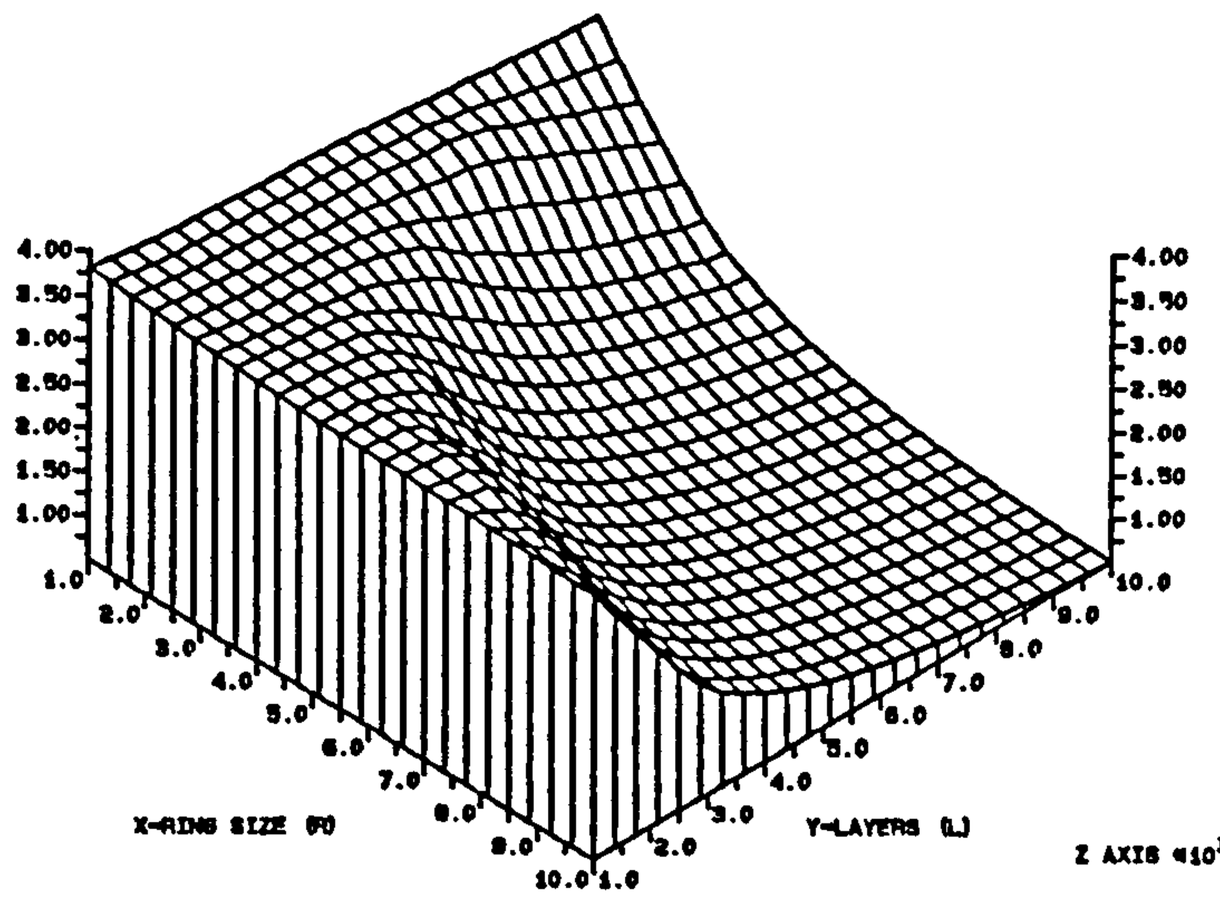
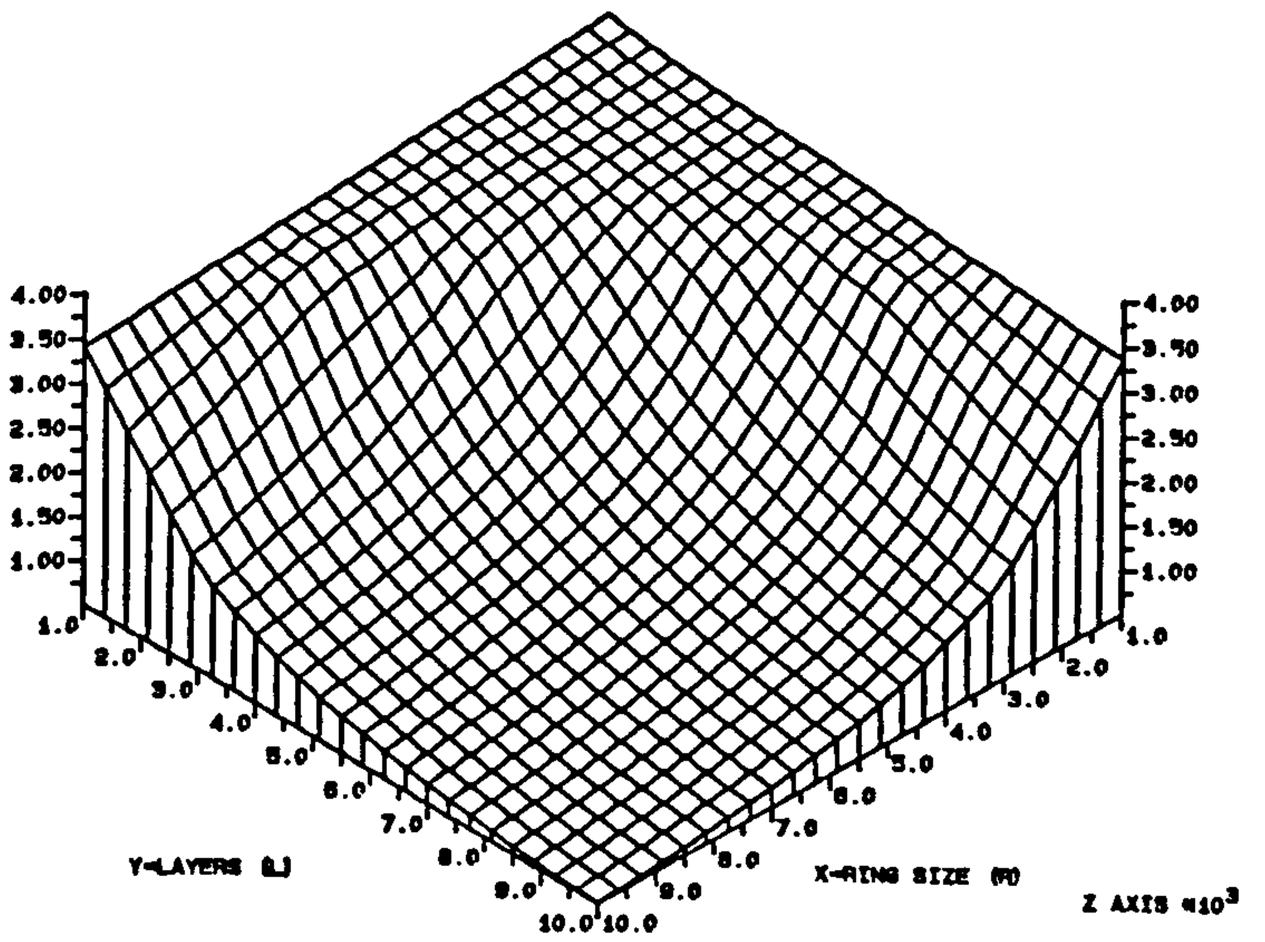
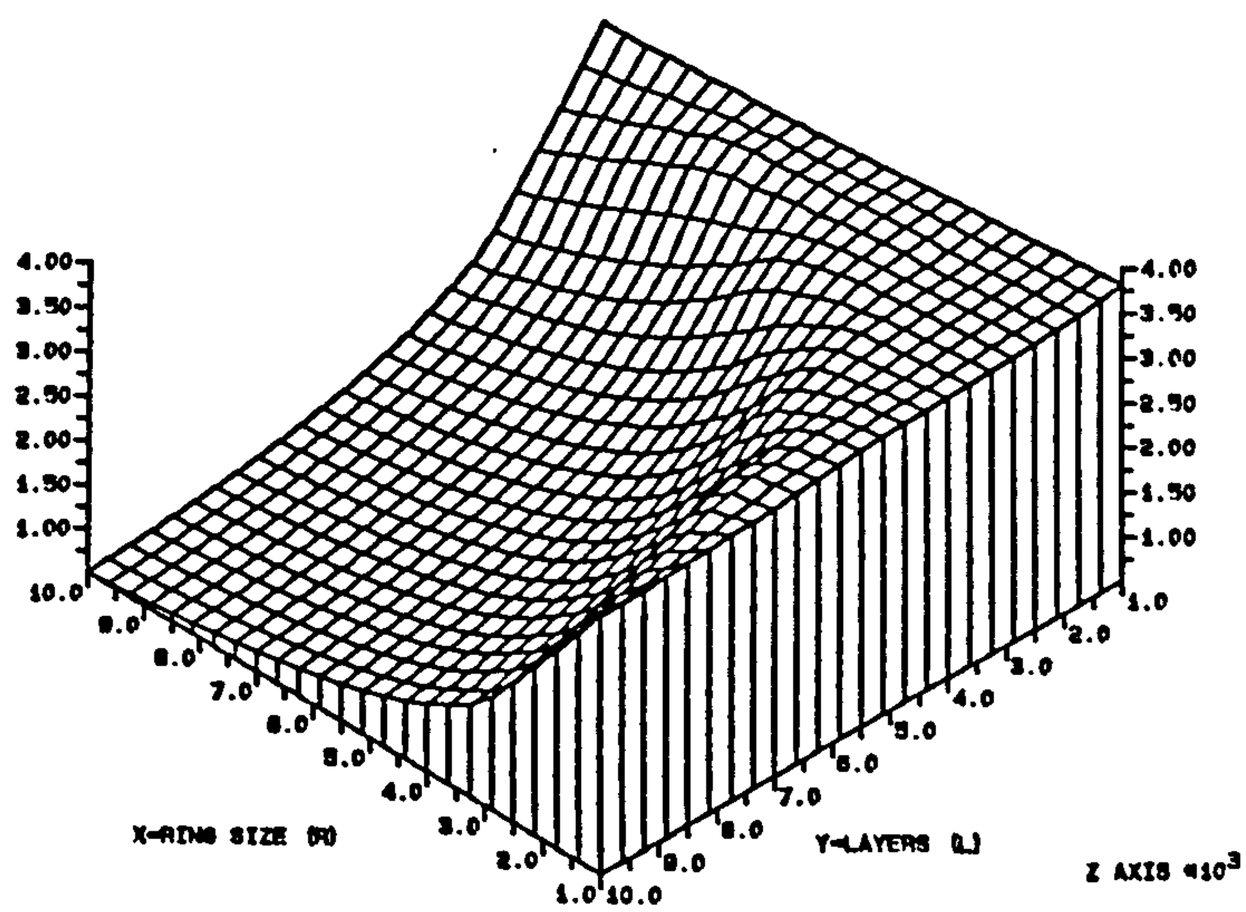


FIG. 5A16.1 HOMOGENEOUS COMMS1 FED AT ONE POINT WITH DATA OF TYPE R100. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





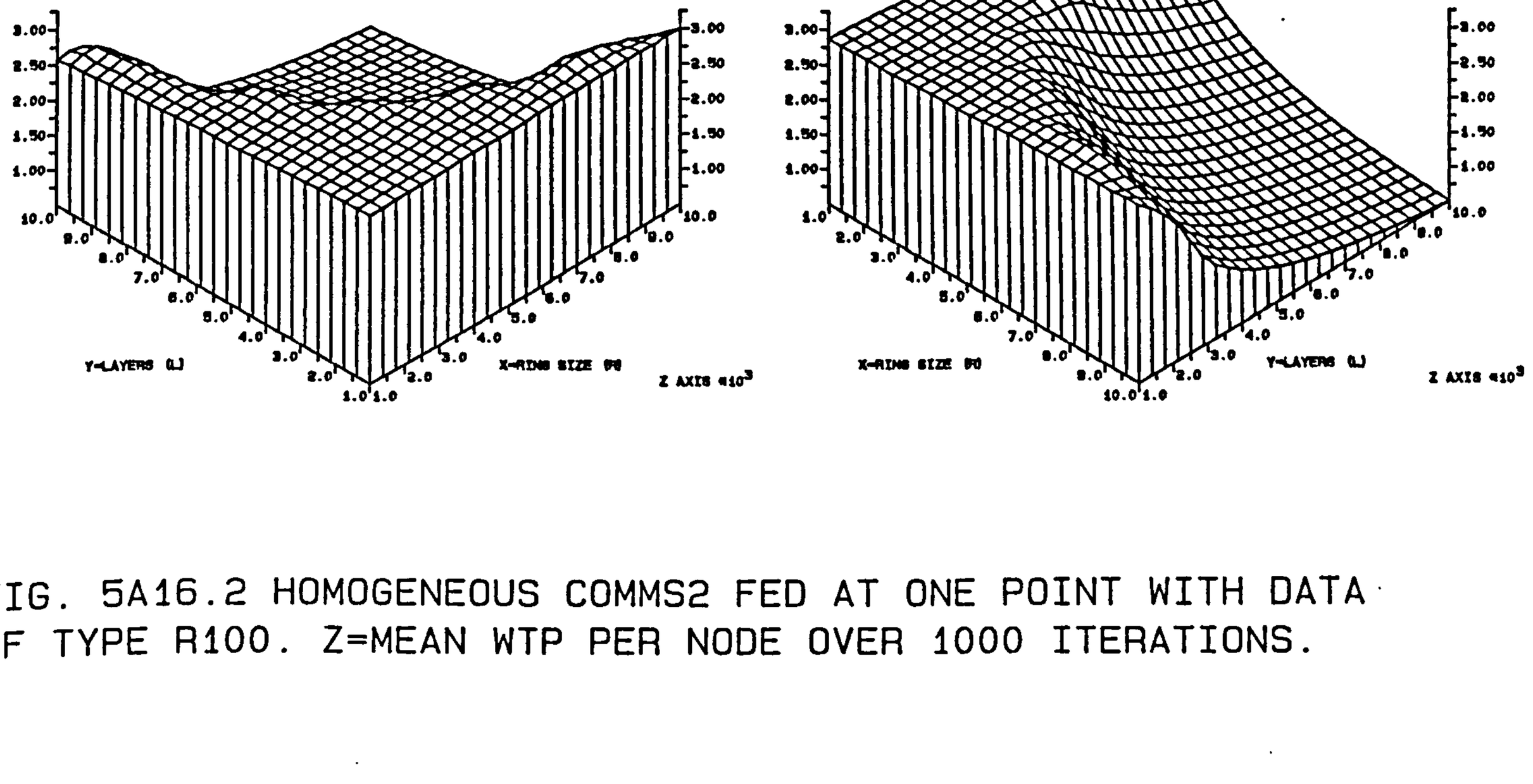
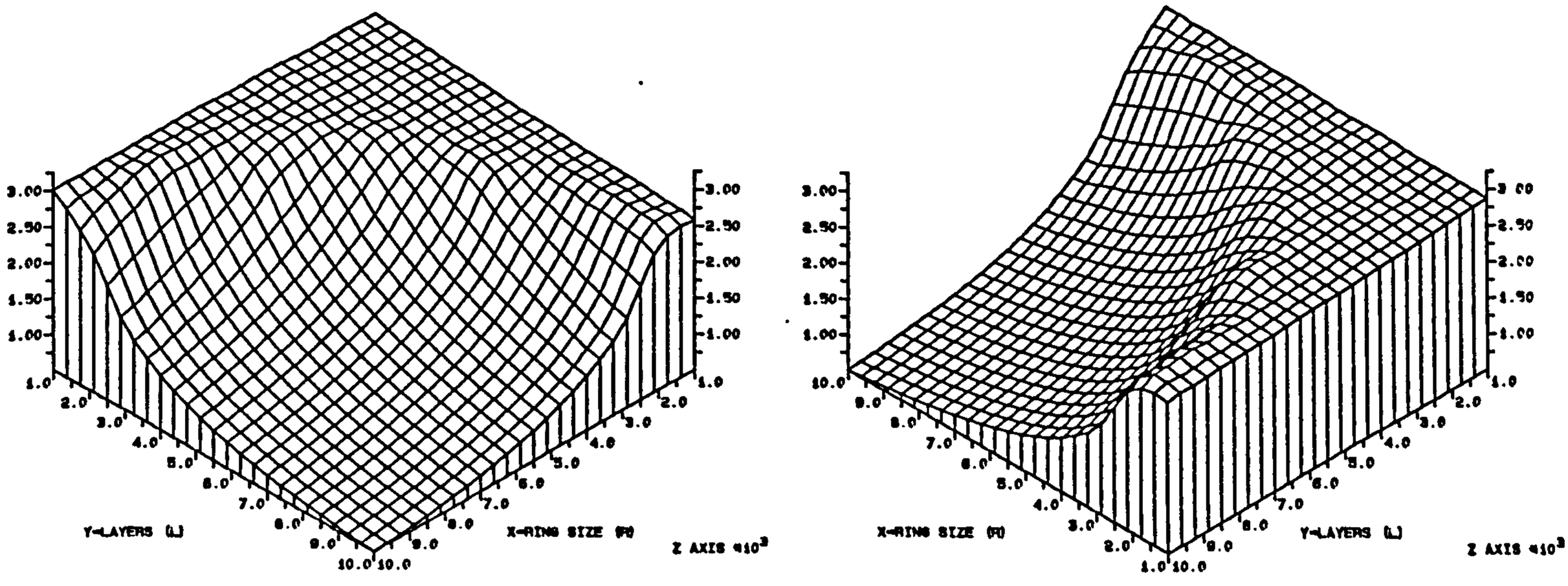
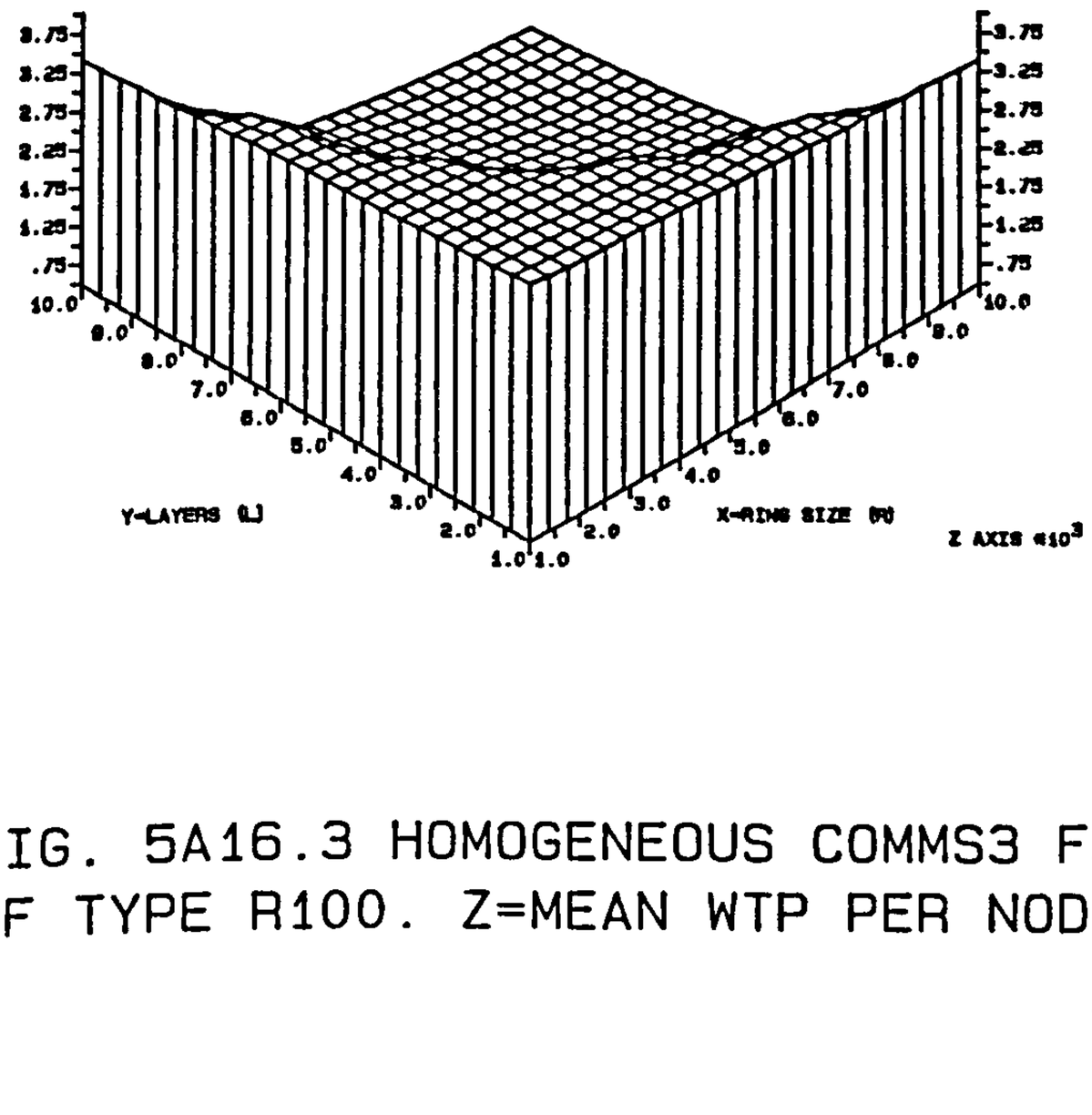


FIG. 5A16.2 HOMOGENEOUS COMMS2 FED AT ONE POINT WITH DATA OF TYPE R100. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





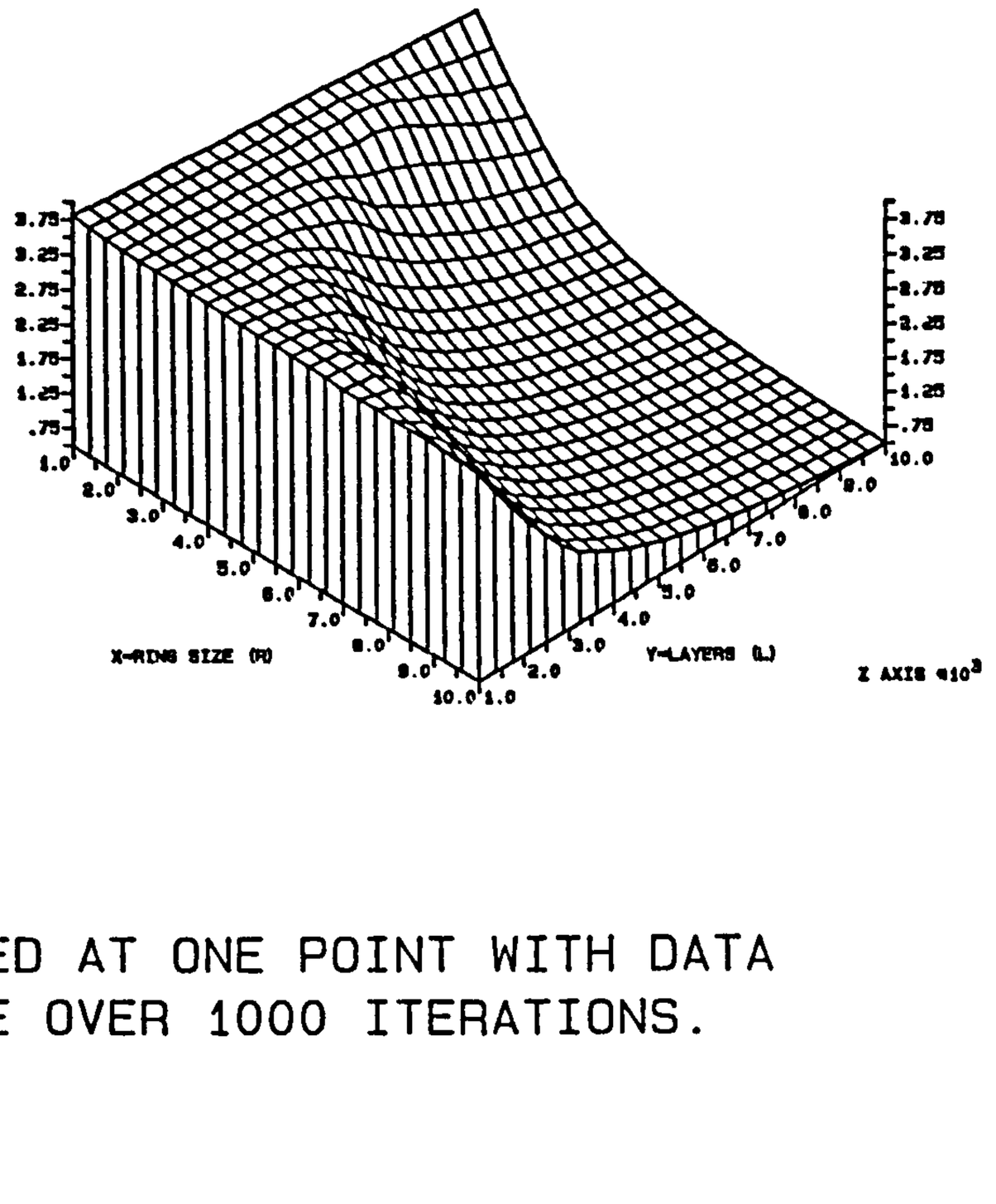
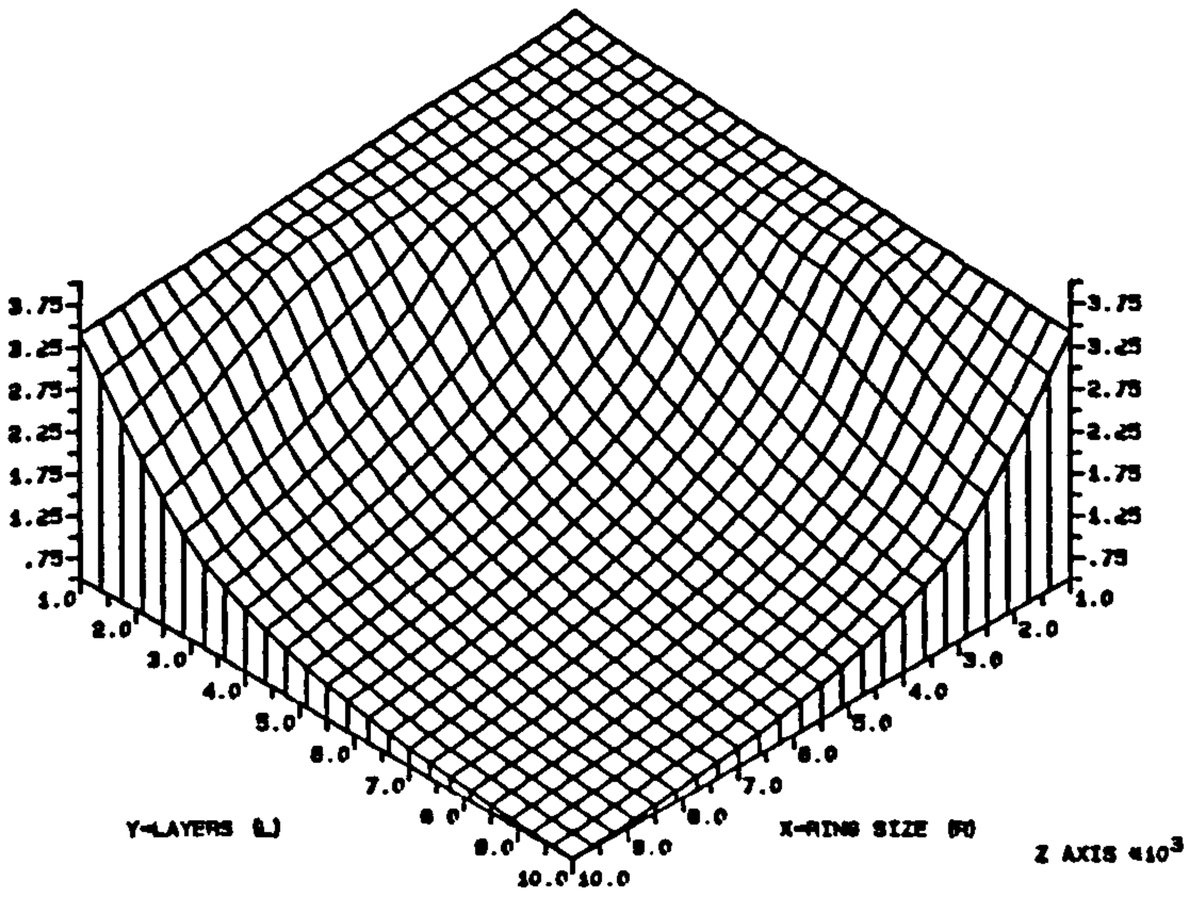
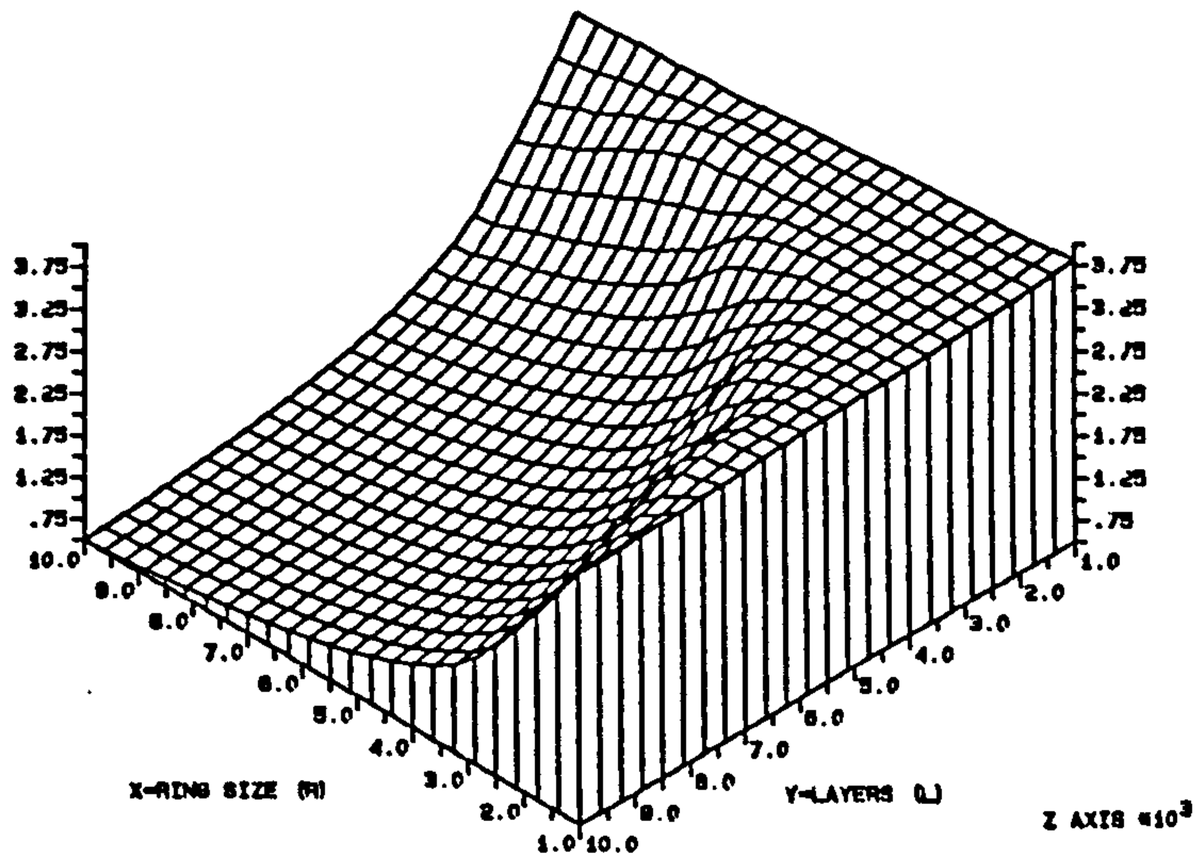


FIG. 5A16.3 HOMOGENEOUS COMMS3 FED AT ONE POINT WITH DATA OF TYPE R100. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.





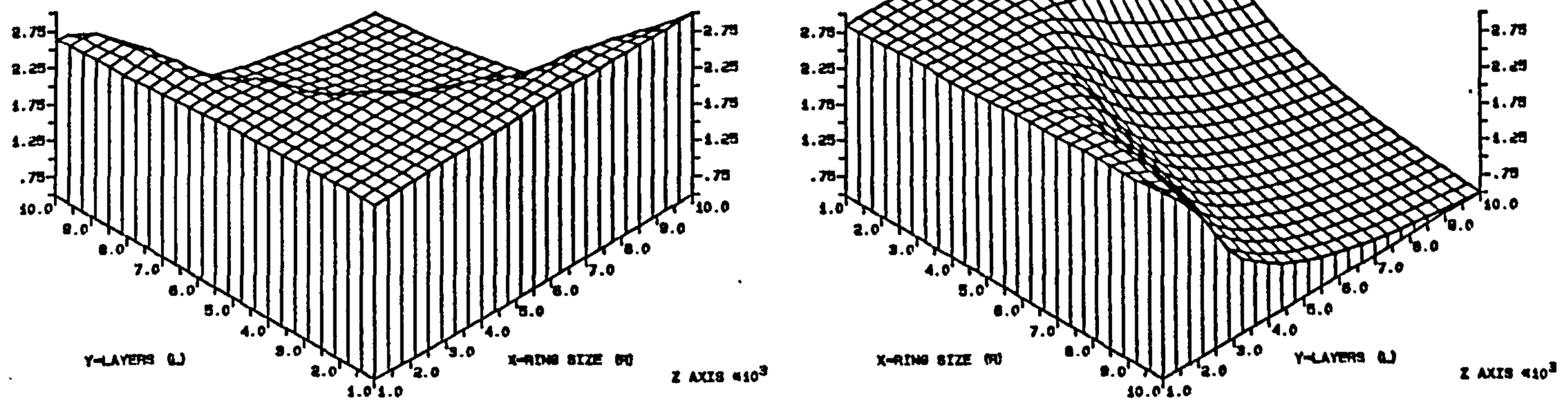
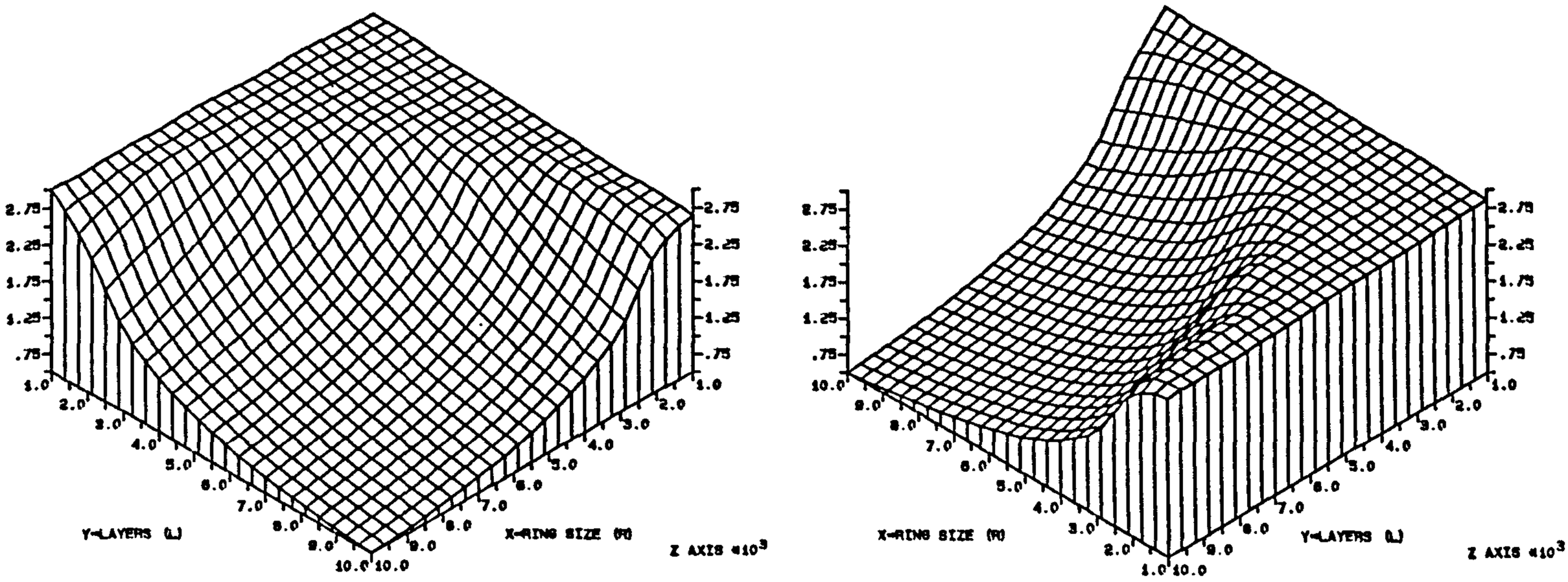


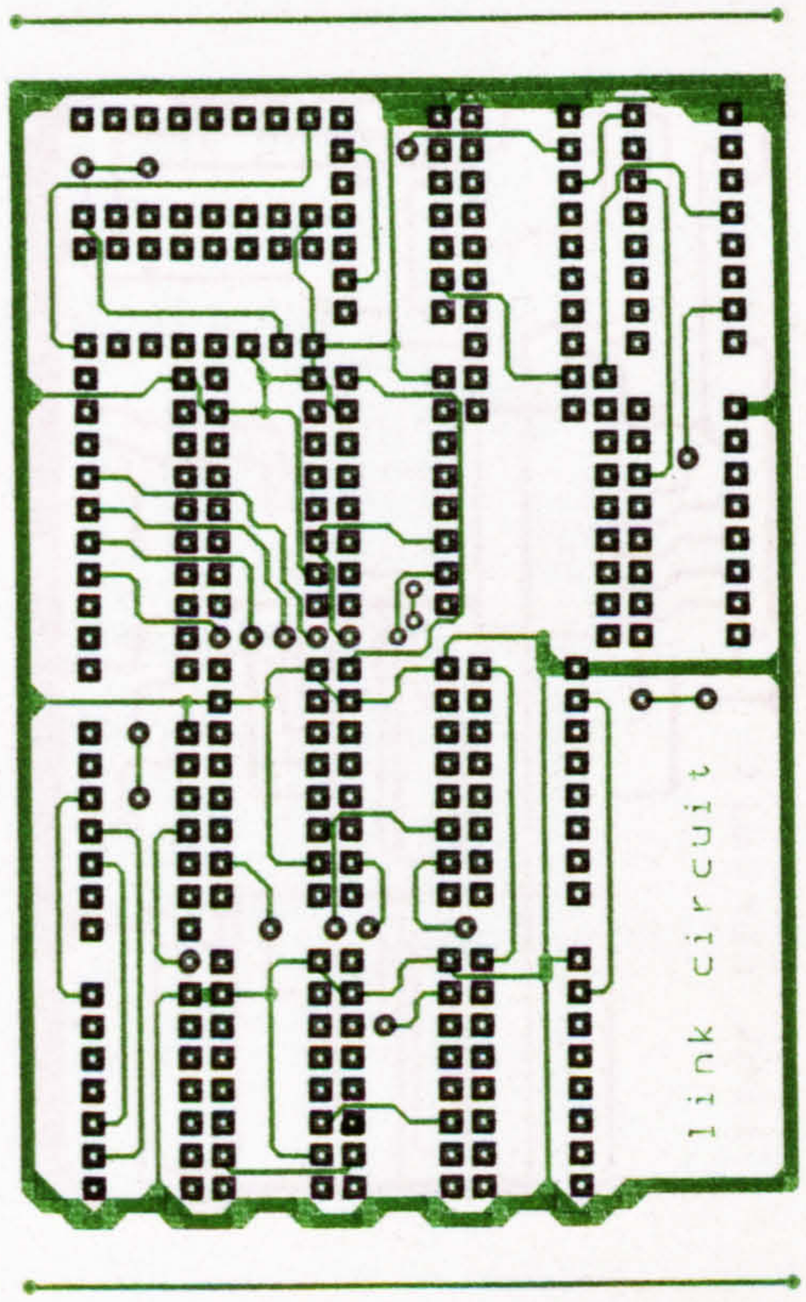
FIG. 5A16.4 HOMOGENEOUS COMMS4 FED AT ONE POINT WITH DATA OF TYPE R100. Z=MEAN WTP PER NODE OVER 1000 ITERATIONS.

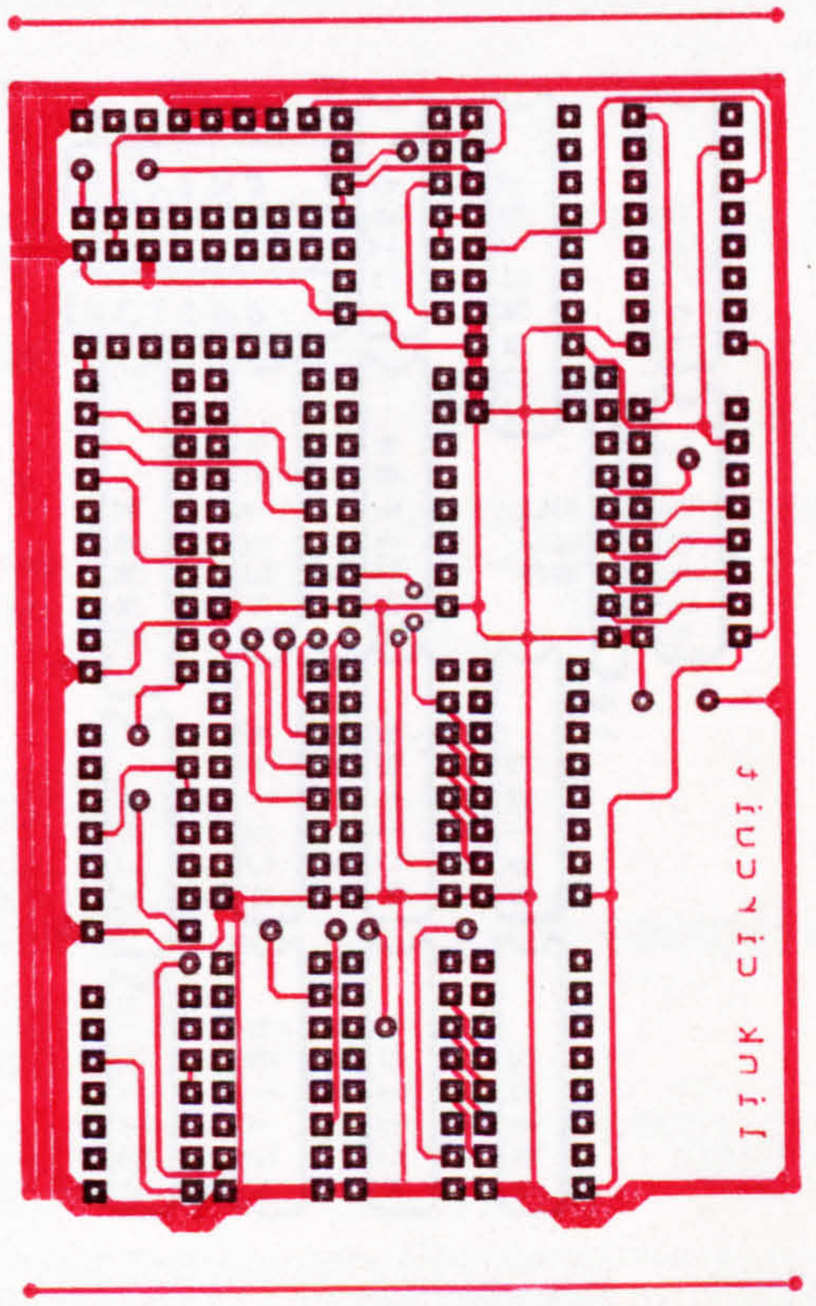


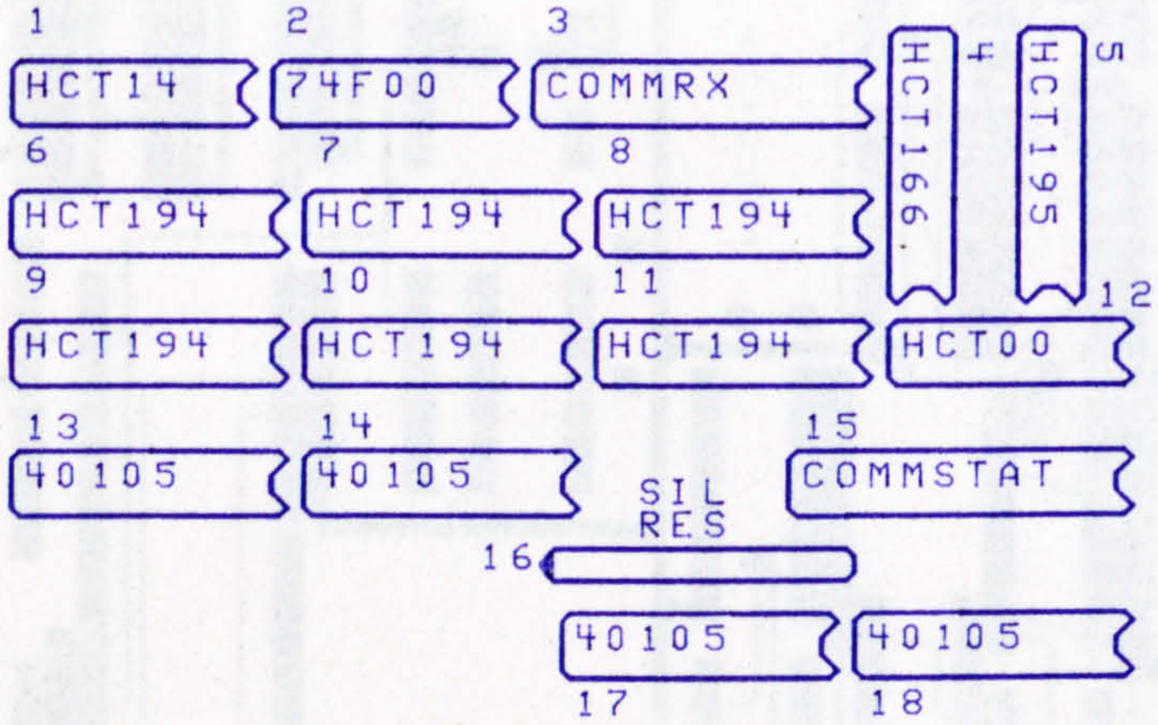
6 Printed Circuit Board Designs

Link Circuit

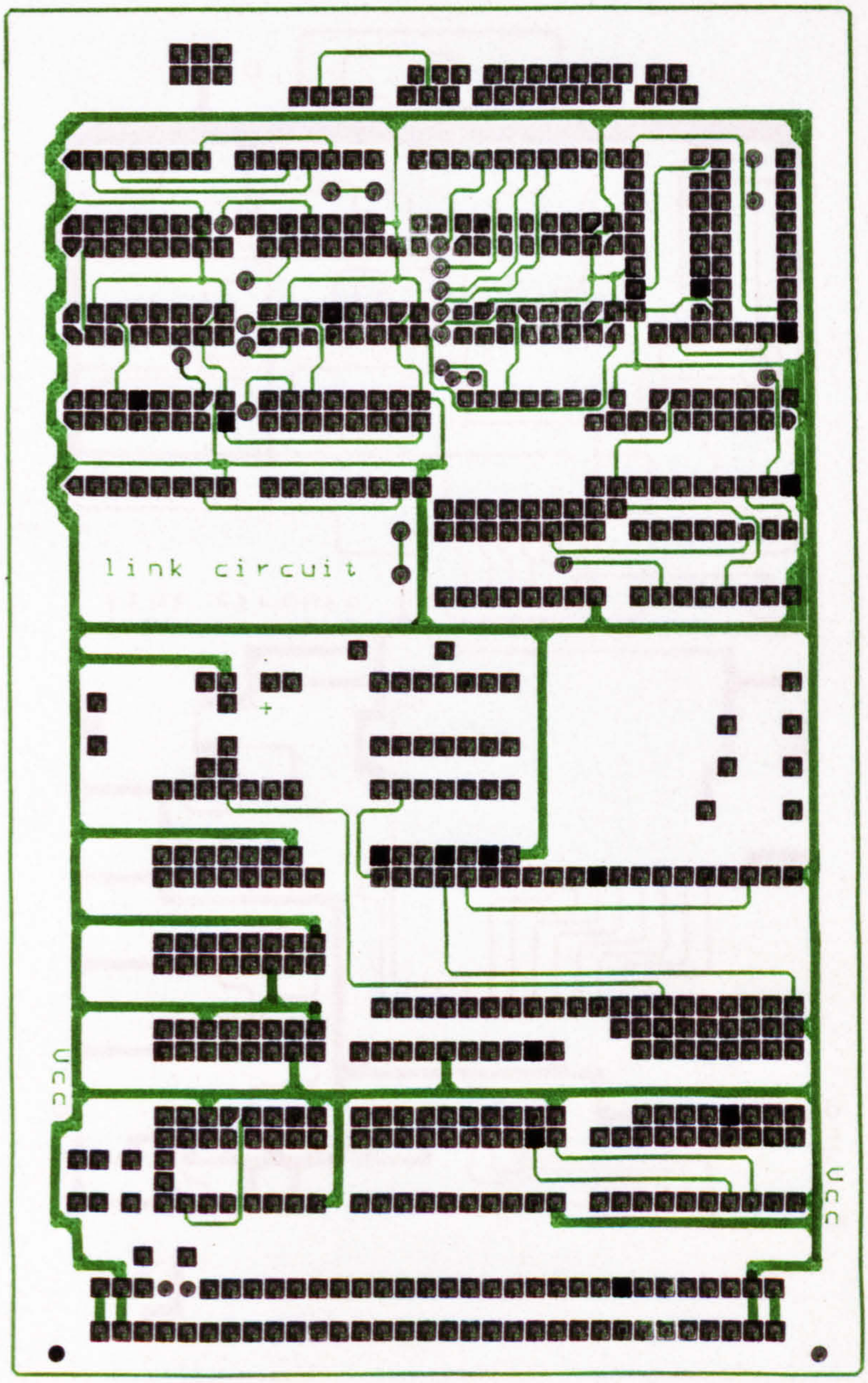
Component Layer

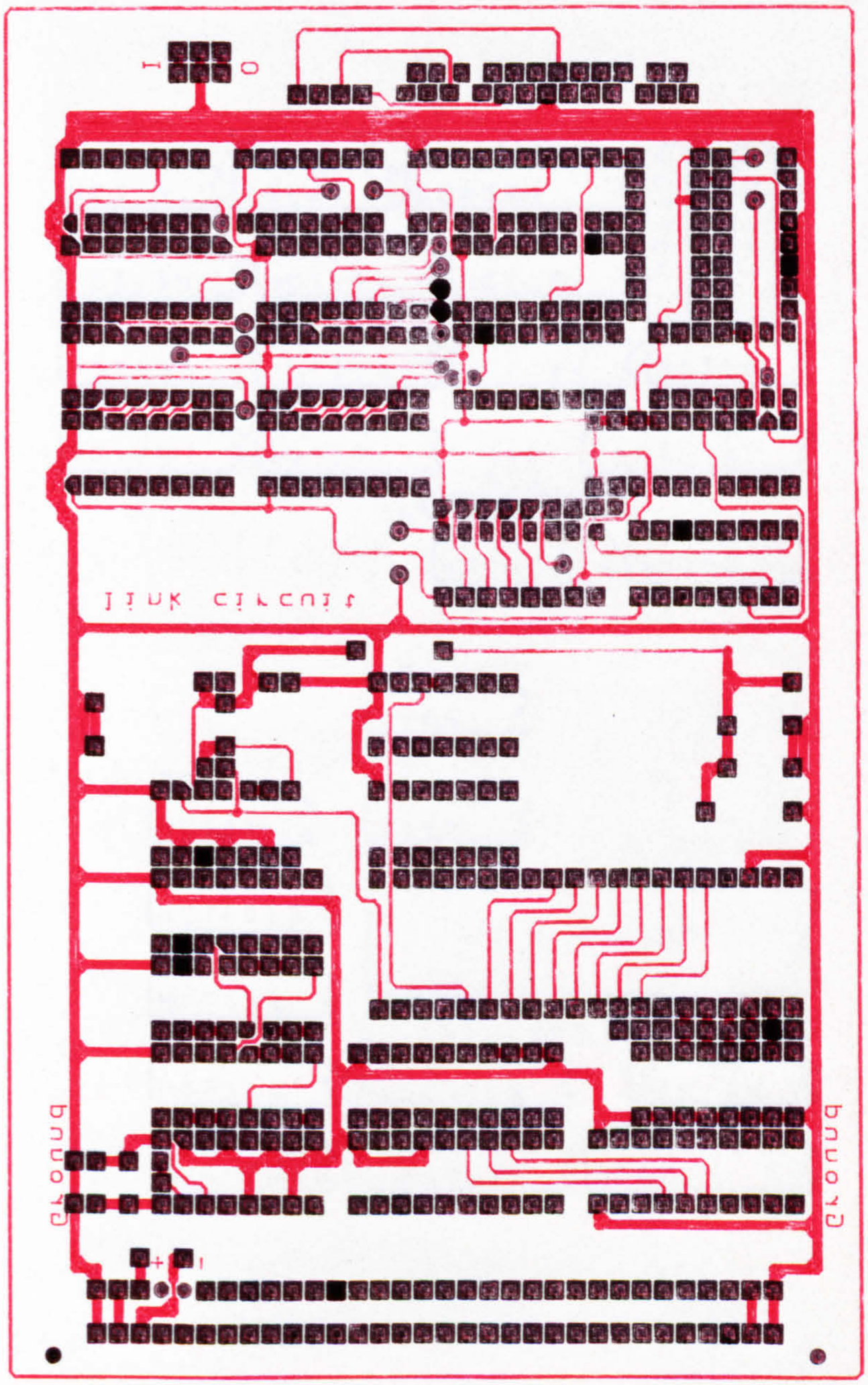


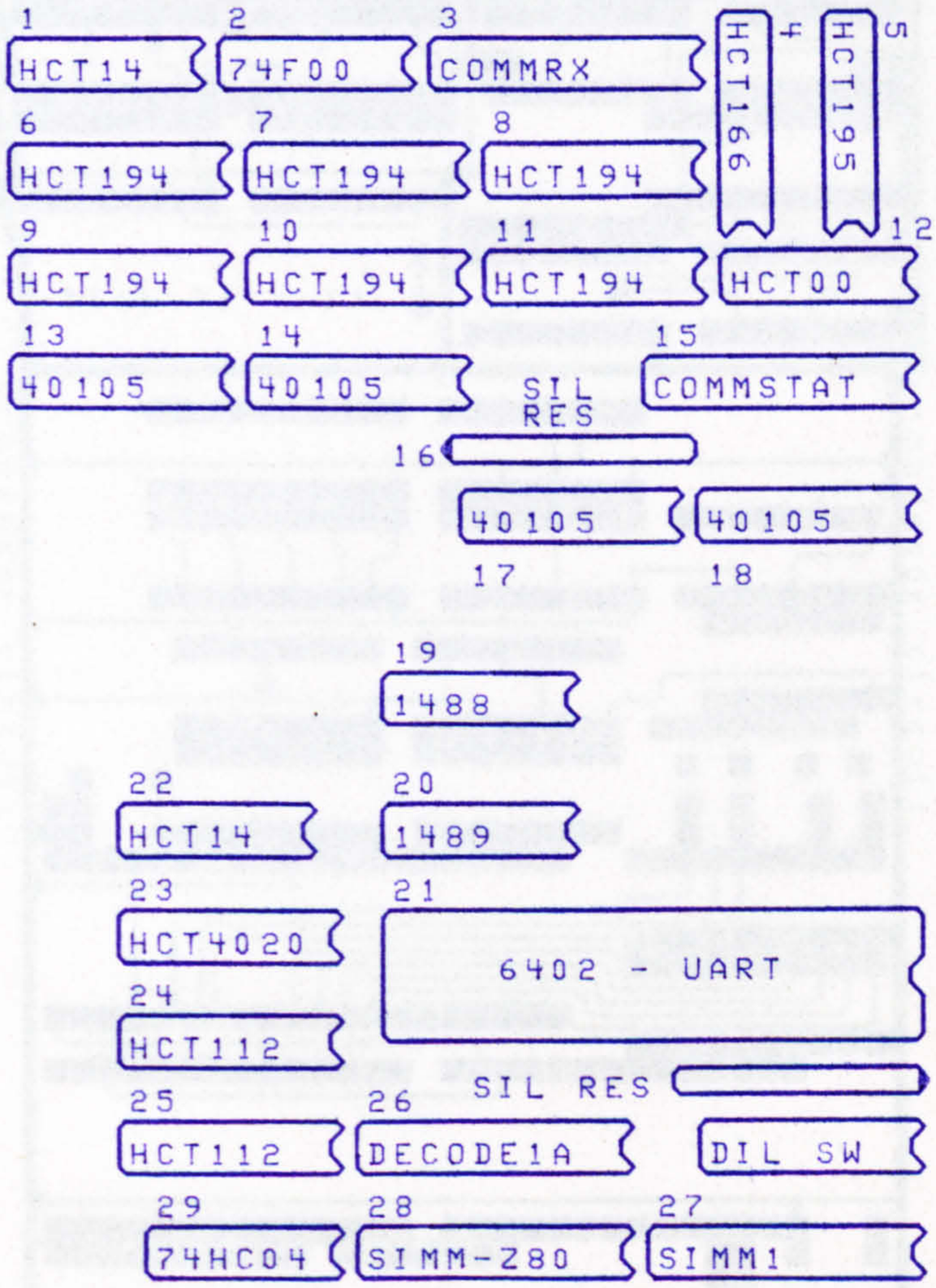



1
HCT14
2
74F00
3
COMMRX
4
HCT166
5
HCT195
6
7
8
9
HCT194
10
HCT194
11
HCT194
12
HCT194
13
HCT194
14
HCT194
15
HCT00
40105
40105
SIL RES
COMMSTAT
16
40105
40105
17
18

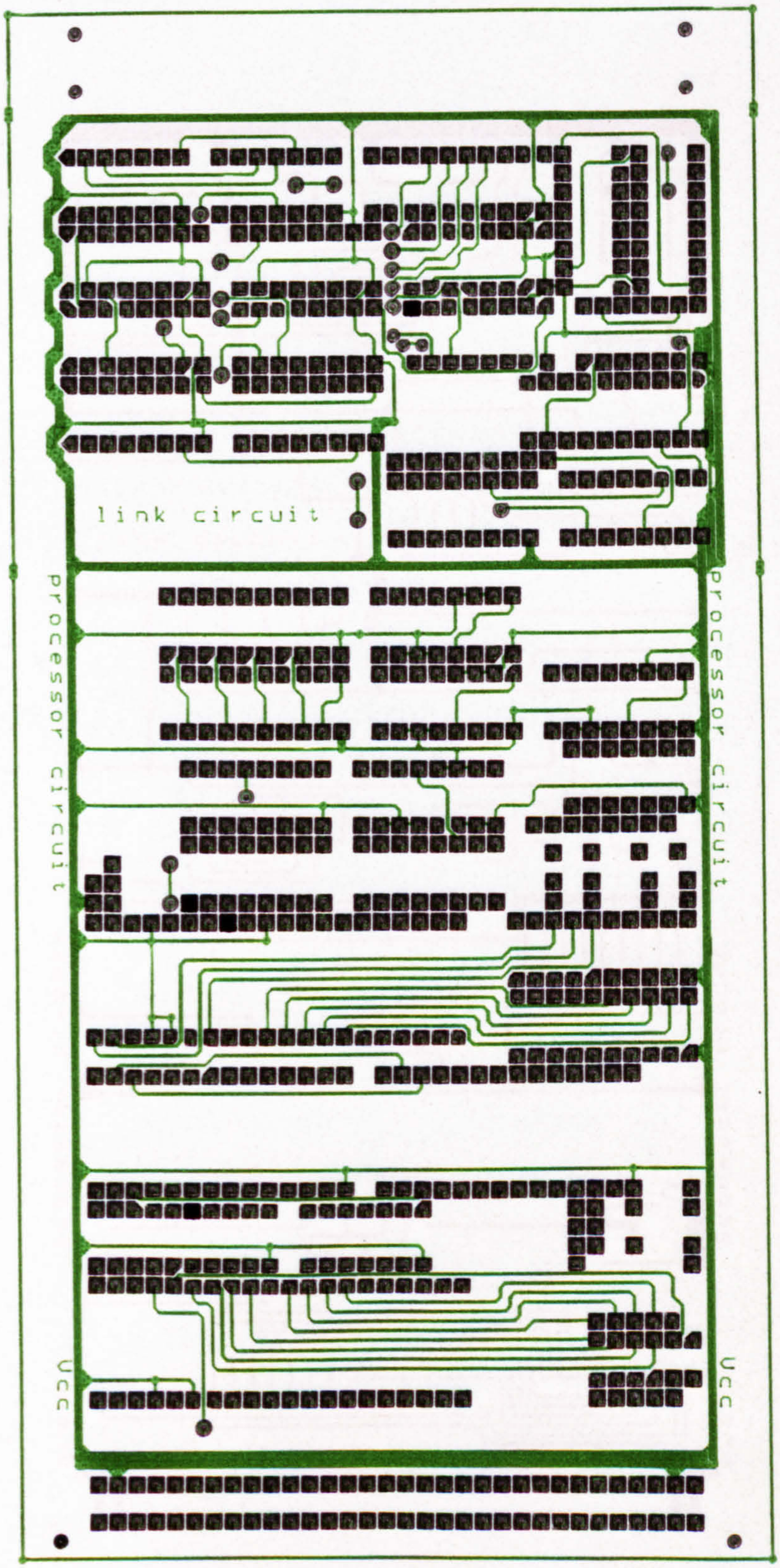
Component Layer

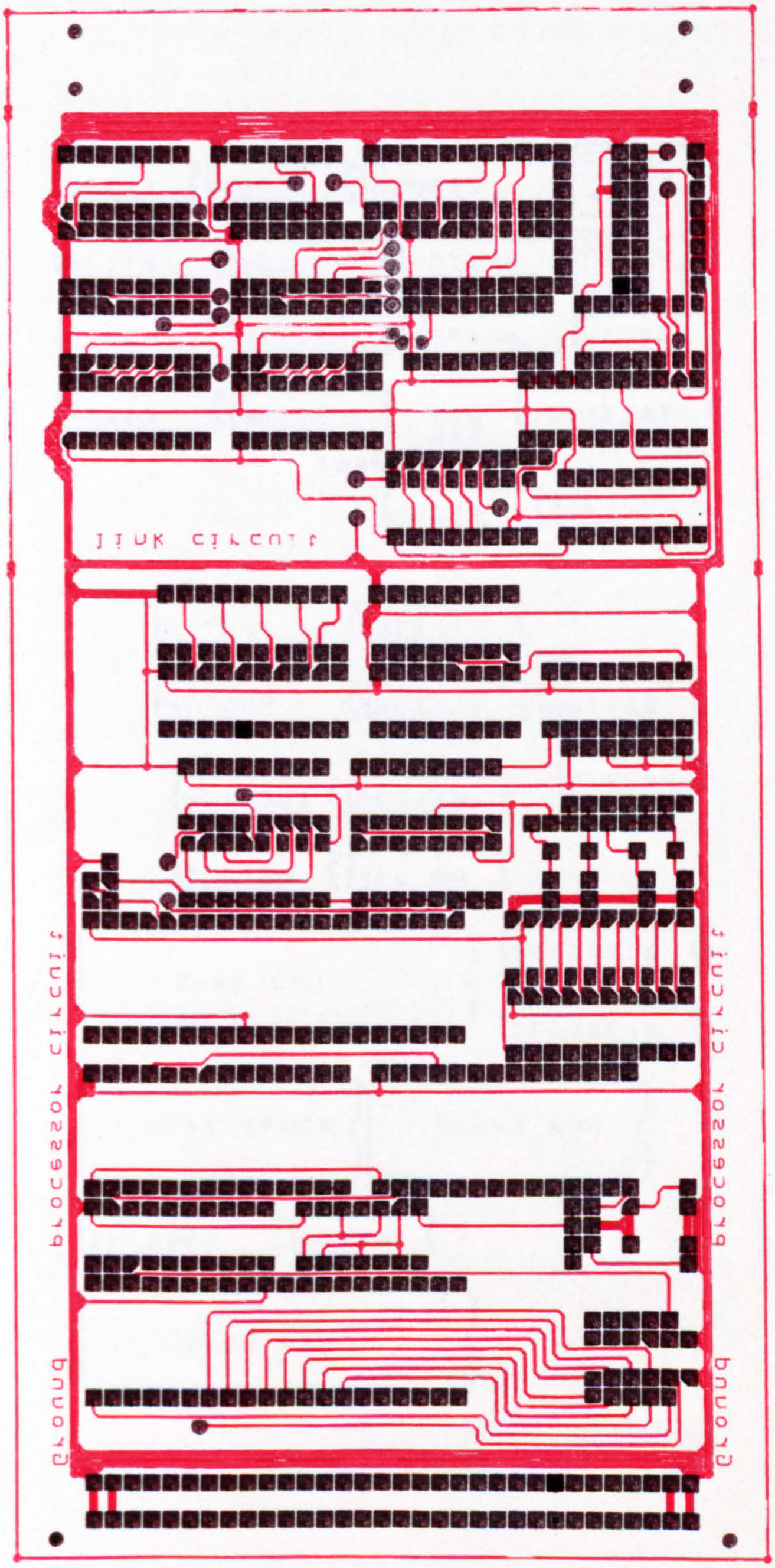


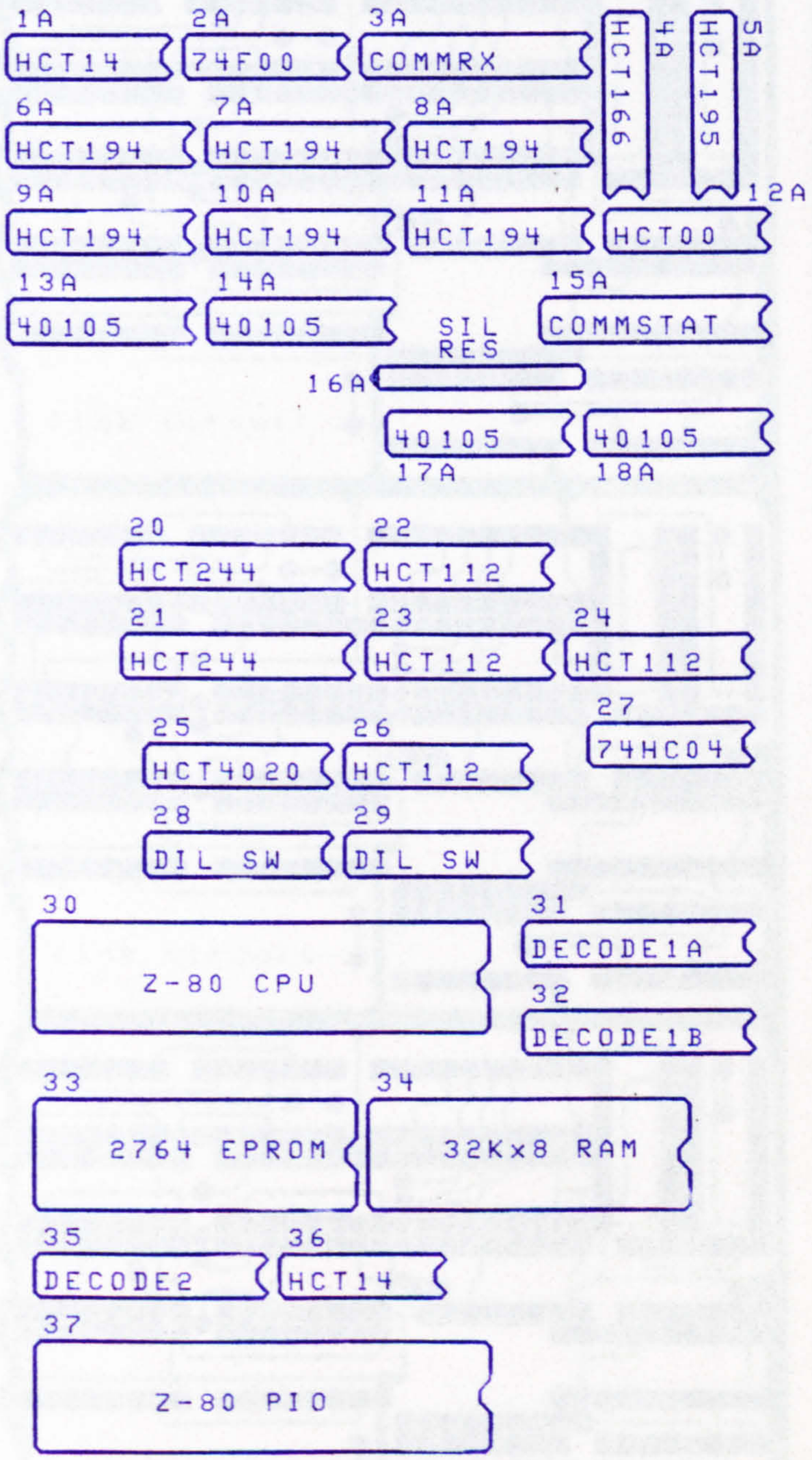

I
O
Logic circuit
Ground
Ground
+
-


1
HCT14
2
74F00
3
COMMRX
4
HCT166
5
HCT195
6
HCT194
7
HCT194
8
HCT194
9
HCT194
10
HCT194
11
HCT194
12
HCT00
13
40105
14
40105
SIL RES
15
COMMSTAT
16
SIL RES
40105
17
40105
18
19
1488
22
HCT14
20
1489
23
HCT4020
21
6402 - UART
24
HCT112
SIL RES
25
HCT112
26
DECODE1A
DIL SW
29
28
27
74HC04
SIMM-Z80
SIMM1

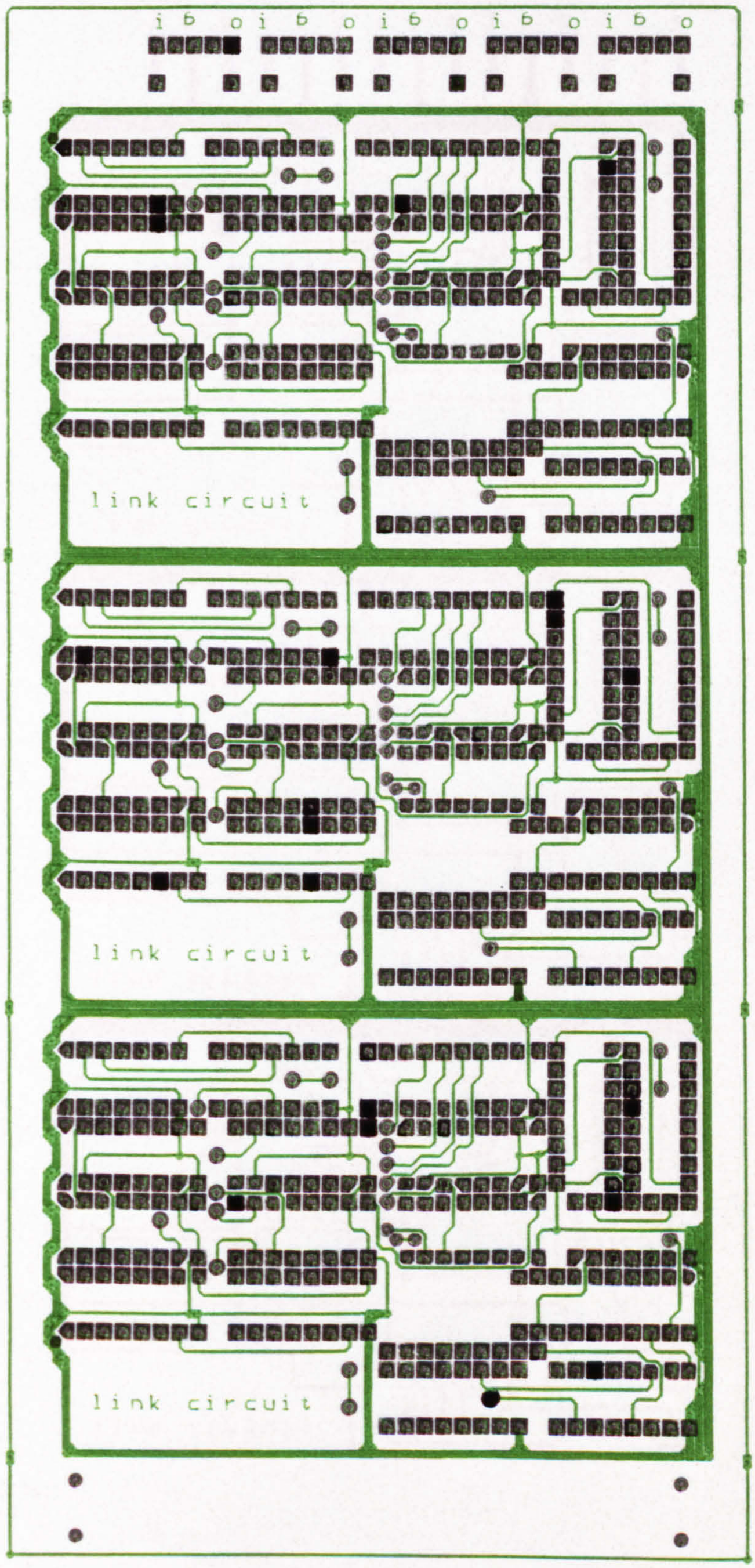
Component Layer



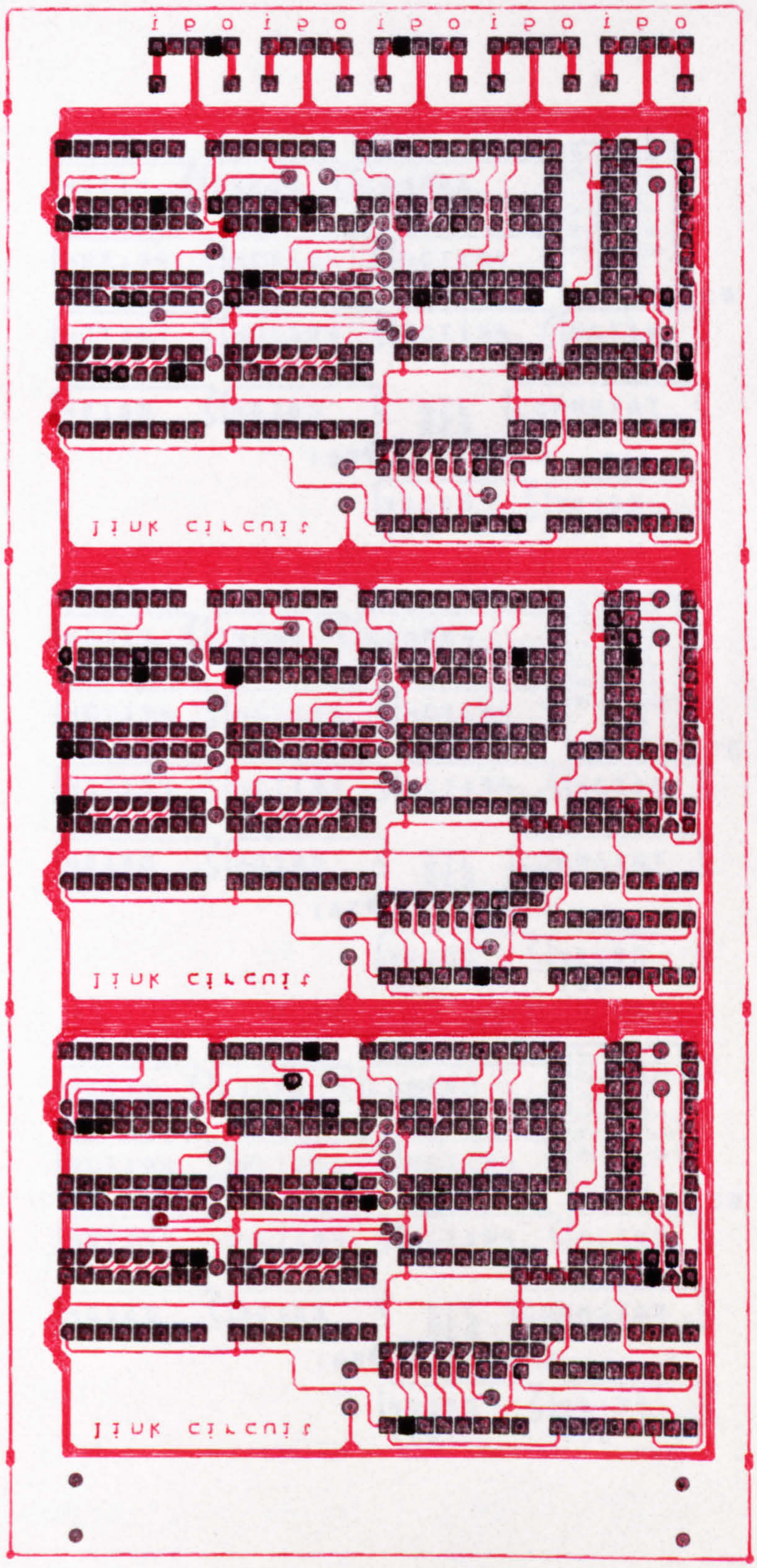

JIUK CIKCUIJ
JIUK CIKCUIJ
GROUND
GROUND

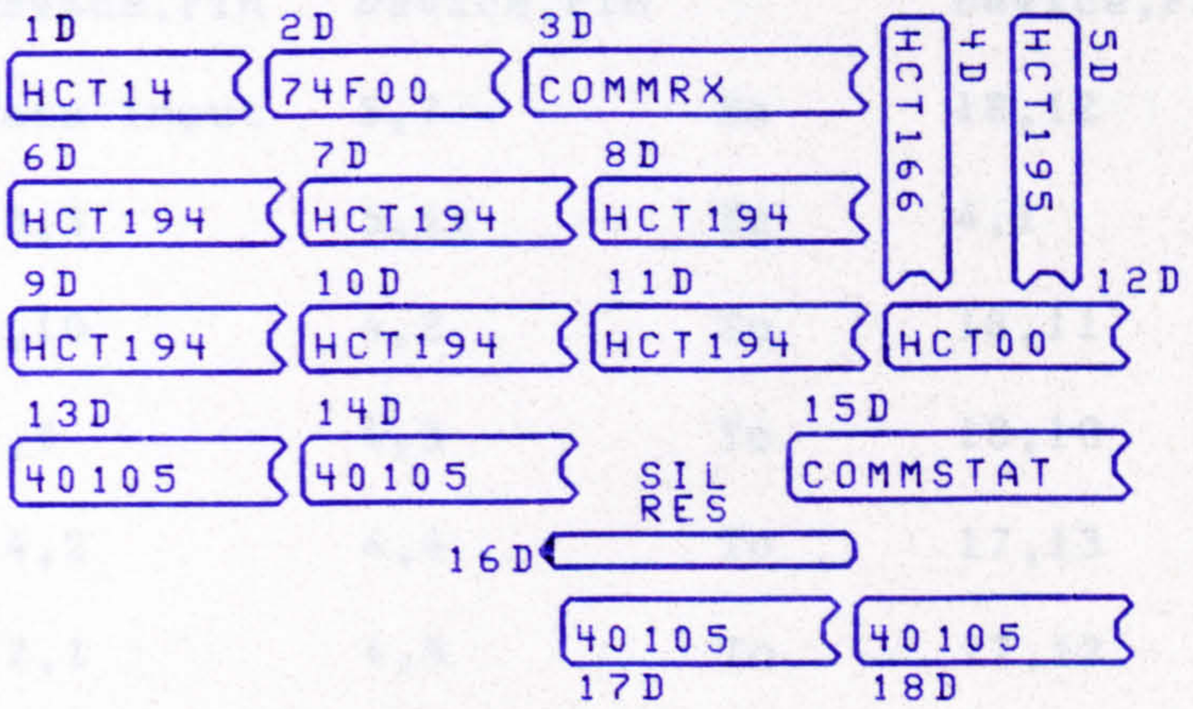

1A
HCT14
2A
74F00
3A
COMMRX
4A
HCT166
5A
HCT195
6A
HCT194
7A
HCT194
8A
HCT194
9A
HCT194
10A
HCT194
11A
HCT194
12A
HCT00
13A
40105
14A
40105
15A
COMMSTAT
SIL
RES
16A
40105
17A
40105
18A
20
HCT244
22
HCT112
21
HCT244
23
HCT112
24
HCT112
25
HCT4020
26
HCT112
27
74HC04
28
DIL SW
29
DIL SW
30
Z-80 CPU
31
DECODE1A
32
DECODE1B
33
2764 EPROM
34
32Kx8 RAM
35
DECODE2
36
HCT14
37
Z-80 PIO

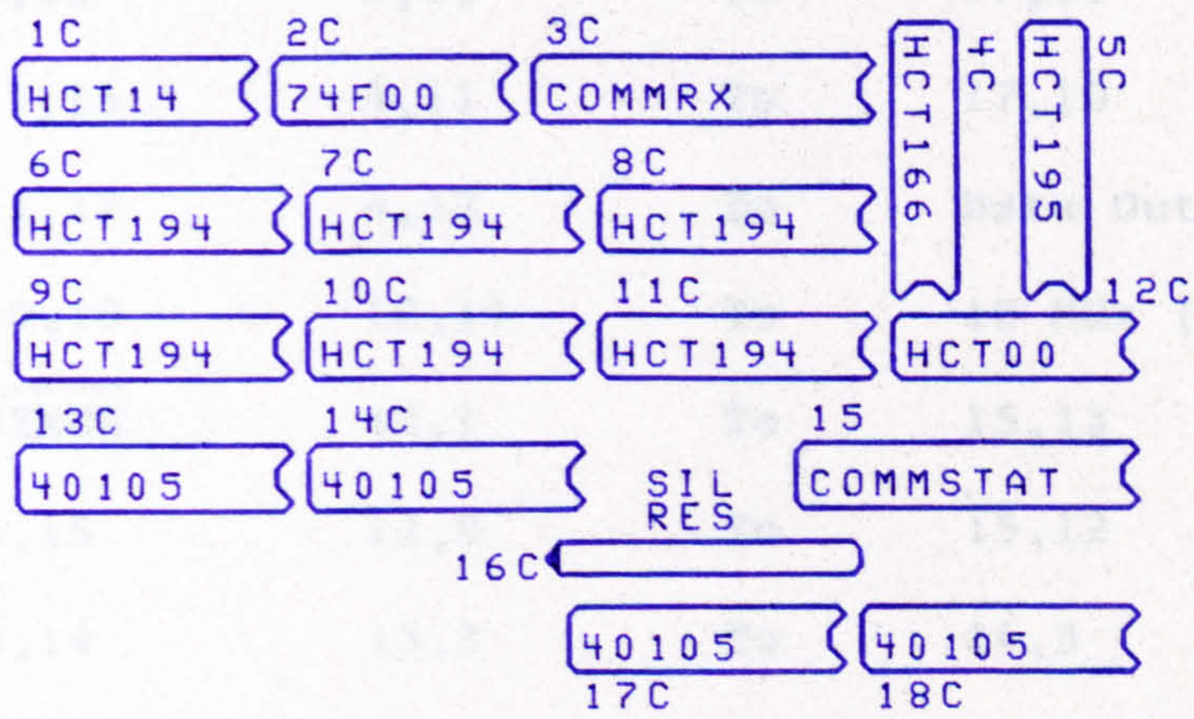
Component Layer

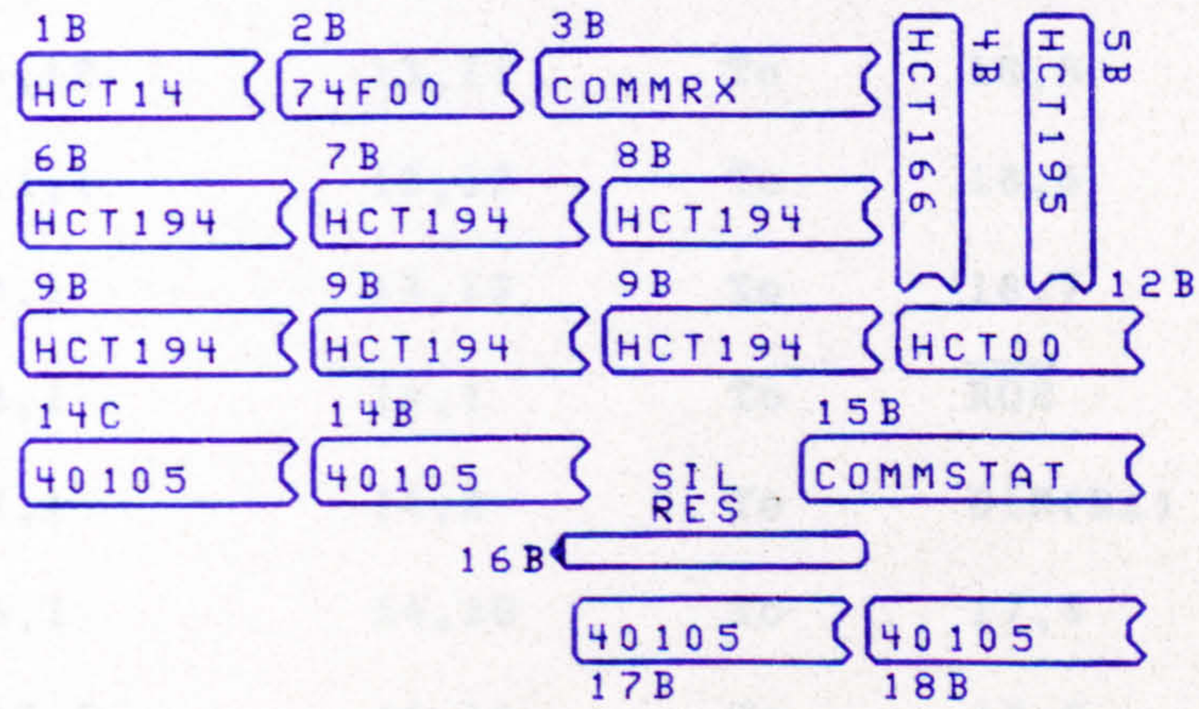


Solder Layer




1D
HCT14
2D
74F00
3D
COMMRX
4D
HCT166
5D
HCT195
6D
HCT194
7D
HCT194
8D
HCT194
9D
HCT194
10D
HCT194
11D
HCT194
12D
HCT00
13D
40105
14D
40105
15D
COMMSTAT
SIL
RES
16D
40105
17D
40105
18D


1C
HCT14
2C
74F00
3C
COMMRX
4C
HCT166
5C
HCT195
6C
HCT194
7C
HCT194
8C
HCT194
9C
HCT194
10C
HCT194
11C
HCT194
12C
HCT00
13C
40105
14C
40105
15
COMMSTAT
SIL
RES
16C
40105
17C
40105
18C


1B
HCT14
2B
74F00
3B
COMMRX
4B
HCT166
5B
HCT195
6B
HCT194
7B
HCT194
8B
HCT194
9B
HCT194
9B
HCT194
9B
HCT194
HCT00
14C
40105
14B
40105
15B
COMMSTAT
SIL
RES
16B
40105
17B
40105
18B

APPENDIX 7

7 Wire Wrap Connections To Complete a Link Circuit

From Device,Pin		To Device,Pin	From Device,Pin		To Device,Pin
1,5	To	Data Input	5,7	To	18,12
1,2	To	13,3	5,12	To	4,1
1,3	To	2,10	4,2	To	18,11
1,4	To	2,9	4,3	To	18,10
1,9	To	14,2	4,4	To	17,13
1,13	To	17,1	4,5	To	17,12
1,12	To	4,12	4,10	To	17,11
2,1	To	11,13	4,11	To	17,10
2,2	To	11,12	4,13	To	Data Output
2,3	To	10,10	10,11	To	10 MHz (PA)
2,6	To	$\overline{\text{SYNCH}}$	12,1	To	15,13
3,8	To	6,15	12,9	To	15,12
3,9	To	6,14	13,3	To	14,3
3,13	To	6,13	13,10	To	18,4
3,14	To	6,12	13,11	To	18,5
3,19	To	11,1	13,12	To	18,6
10,1	To	9,1	13,13	To	18,7
3,18	To	8,1	14,1	To	$\overline{\text{ROE}}$
8,1	To	7,1	14,2	To	DIR(Rx)
7,1	To	6,1	14,10	To	17,4
3,17	To	12,5	14,11	To	17,5
3,12	To	12,13	14,12	To	17,6
5,6	To	18,13	14,13	To	17,7

From		To	From		To
Device,Pin		Device,Pin	Device,Pin		Device,Pin
14,14	To	DOR(Rx)	16,8	To	18,12
14,15	To	RSO	17,2	To	DIR(Tx)
15,1	To	10 MHz	17,9	To	RESET
15,2	To	$\overline{\text{RESET}}$	17,14	To	DOR(Tx)
15,6	To	14,2	18,3	To	TSI
16,7	To	18,11			

APPENDIX 8

8 Wire Wrap Connections To Complete an RS-232C to Inmos Interface

From Device,Pin		To Device,Pin	From Device,Pin		To Device,Pin
19,1	To	-12V	21,23	To	28,16
19,2	To	27,19	22,2	To	15,2
19,3	To	D-conn,20	22,2	To	27,2
19,6	To	D-conn,2	22,4	To	17,9
19,14	To	+12V	23,6	To	28,8
20,1	To	D-conn,5	23,9	To	28,6
20,3	To	27,3	23,10	To	25,13
20,4	To	D-conn,3	23,11	To	28,19
21,4	To	28,18	24,5	To	27,1
21,5	To	13,10	25,14	To	2,6
21,6	To	13,11	25,9	To	10,11
21,7	To	13,12	25,5	To	15,1
21,8	To	13,13	26,5	To	28,15
21,9	To	14,10	26,7	To	28,13
21,10	To	14,11	26,8	To	28,12
21,11	To	14,12	26,9	To	28,14
21,12	To	14,13	26,19	To	18,3
21,17	To	24,9	26,17	To	14,1
21,18	To	28,17	26,18	To	14,15
21,19	To	27,5	27,6	To	14,14
21,22	To	27,4	27,7	To	17,2

APPENDIX 9

9 Wire Wrap Connections To Complete a Processor Board

From Device,Pin		To Device,Pin	From Device,Pin		To Device,Pin
36,4	To	17A,9	30,22	To	35,6
17A,9	To	17B,9	30,19	To	31,11
17B,9	To	17C,9	30,19	To	35,8
17C,9	To	17D,9	30,20	To	35,9
36,10	To	15A,2	30,20	To	37,36
15A,2	To	15B,2	30,21	To	35,7
15B,2	To	15C,2	30,21	To	37,35
15C,2	To	15D,2	30,22	To	35,6
36,8	To	30,26	30,27	To	32,6
25,7	To	28,3	30,27	To	35,11
25,6	To	28,2	30,27	To	37,37
25,5	To	28,1	30,16	To	37,23
25,4	To	29,8	35,12	To	25,11
25,3	To	29,7	35,13	To	33,22
25,2	To	29,6	35,14	To	33,20
25,1	To	29,5	35,17	To	34,27
30,6	To	37,25	35,18	To	34,22
27,12	To	23,1	35,19	To	34,20
30,21	To	35,7	33,18	To	34,18
30,21	To	37,35	33,19	To	34,19

From Device,Pin		To Device,Pin	From Device,Pin		To Device,Pin
30,14	To	33,11	30,7	To	20,12
30,15	To	33,12	30,9	To	20,14
30,12	To	33,13	30,10	To	20,16
30,8	To	33,15	30,13	To	20,18
30,7	To	33,16	20,3	To	14A,13
30,9	To	33,17	20,5	To	14A,12
30,10	To	33,18	20,7	To	14A,11
30,13	To	33,19	20,9	To	14A,10
30,7	To	37,39	20,12	To	13A,13
30,8	To	37,40	20,14	To	13A,12
30,9	To	37,38	20,16	To	13A,11
30,10	To	37,3	20,18	To	13A,10
30,12	To	37,1	14A,13	To	14B,13
30,13	To	37,2	14A,12	To	14B,12
30,14	To	37,19	14A,11	To	14B,11
30,15	To	37,20	14A,10	To	14B,10
30,14	To	20,3	13A,13	To	13B,13
30,15	To	20,5	13A,12	To	13B,12
30,12	To	20,7	13A,11	To	13B,11
30,8	To	20,9	13A,10	To	13B,10

From		To	From		To
Device,Pin		Device,Pin	Device,Pin		Device,Pin
14B,13	To	14C,13	30,34	To	33,6
14B,12	To	14C,12	30,35	To	33,5
14B,11	To	14C,11	30,36	To	33,4
14B,10	To	14C,10	30,37	To	33,3
13B,13	To	13C,13	30,38	To	33,25
13B,12	To	13C,12	30,39	To	33,24
13B,11	To	13C,11	30,40	To	33,21
13B,10	To	13C,10	30,1	To	33,23
14C,13	To	14D,13	30,2	To	33,2
14C,12	To	14D,12	33,10	To	34,10
14C,11	To	14D,11	33,9	To	34,9
14C,10	To	14D,10	33,8	To	34,8
13C,13	To	13D,13	33,7	To	34,7
13C,12	To	13D,12	33,6	To	34,6
13C,11	To	13D,11	33,5	To	34,5
13C,10	To	13D,10	33,4	To	34,4
30,30	To	33,10	33,3	To	34,3
30,31	To	33,9	33,25	To	34,25
30,32	To	33,8	33,24	To	34,24
30,33	To	33,7	33,21	To	34,21

From Device,Pin		To Device,Pin	From Device,Pin		To Device,Pin
33,23	To	34,23	32,19	To	18C,3
33,2	To	34,2	20,1	To	32,12
30,3	To	34,26	20,2	To	14D,2
30,4	To	34,1	20,4	To	14C,2
30,1	To	35,1	20,6	To	14B,2
30,2	To	35,2	20,8	To	14A,2
30,3	To	35,3	20,11	To	17D,2
30,4	To	35,4	20,13	To	17C,2
30,5	To	35,5	20,15	To	17B,2
31,14	To	14B,1	20,17	To	17A,2
31,15	To	14B,15	21,1	To	31,12
31,16	To	18B,3	21,2	To	17D,14
31,17	To	14A,1	21,4	To	17C,14
31,18	To	14A,15	21,6	To	17B,14
31,19	To	18A,3	21,8	To	17A,14
32,14	To	14D,1	21,11	To	14D,14
32,15	To	14D,15	21,13	To	14C,14
32,16	To	18D,3	21,15	To	14B,14
32,17	To	14C,1	21,17	To	14A,14
32,18	To	14C,15	22,15	To	2A,6

From Device,Pin		To Device,Pin	From Device,Pin		To Device,Pin
22,14	To	2B,6	24,5	To	15A,1
23,15	To	2C,6	24,5	To	15B,1
23,14	To	2D,6	24,9	To	15C,1
22,5	To	10A,11	24,9	To	15D,1
22,9	To	10B,11	37,4	To	30,37
23,5	To	10C,11	37,5	To	30,35
23,9	To	10D,11	37,6	To	30,36

APPENDIX 10

10 Pal Designs Used in the Multiprocessor Hardware Design

```
PARTNO     00001 ;
NAME       COMMRX;
DATE       13/01/87 ;
REV        01 ;
DESIGNER   G.A.Lester ;
COMPANY    University Of Salford ;
ASSEMBLY   00002 ;
           LOCATION  00003 ;

/*************************************************************/
/*Logic for use in TRANSPUTER compatible link.               */
/*detects acknowledge, detects data and generates the reset  */
/*for the receive shift registers.                           */
/*Final version.          (Different pins to prototype. -rev 00)*/
/*************************************************************/
/*  Allowable Target Device Types:p1618                      */
/*************************************************************/

/**  Inputs  **/

PIN 1          = !r          ;      /* reset signal                    */
PIN 2          = d0          ;      /* data from shift register.       */
PIN 3          = d1          ;      /*                                 */
PIN 4          = d2          ;      /*                                 */
PIN 5          = d3          ;      /*                                 */
PIN 6          = d4          ;      /*                                 */
PIN 7          = d5          ;      /*                                 */
PIN 8          = d6          ;      /*                                 */
PIN 9          = d7          ;      /*                                 */
PIN 13         = d8          ;      /*                                 */
PIN 14         = d9          ;      /*                                 */
PIN 11         = !dire       ;      /* DIR(falling)Edge from rx buffer*/

/**  Outputs  **/

PIN 19         = !mr1        ;      /*Master Reset 1                   */
PIN 18         = !mr2        ;      /*Master Reset 2                   */
PIN 17         = !arxd       ;      /*Acknowledge received             */
PIN 12         = !dd         ;      /*Data bits   Detected             */
PIN 15         = ack         ;      /*Acknowledge bits                 */
PIN 16         = dat         ;      /*Data bits                        */

/**  Logic Equations  **/

mr1  = r # dire ;

mr2  = r # arxd # (d0 & d1) ;

arxd = r # ack & !(d7 # d6 # d5 # d4 # d3 # d2 # d1 # d0) ;

dd   =     dat & !(d7 # d6 # d5 # d4 # d3 # d2 # d1 # d0) ;
```

```
ack  = d8 & !d9 ;

dat  = d8 &  d9 ;
```

```
PARTNO     00002 ;
NAME       COMMSTAT ;
DATE       09/01/87 ;
REV        01 ;
DESIGNER   G.A.Lester ;
COMPANY    University Of Salford ;
ASSEMBLY   00002 ;
LOCATION   00015 ;

/*************************************************************/
/*The state machine for use in the TRANSPUTER compatible links.   */
/*Used in conjunction with commrx.                               */
/*Generates Acknowledge and data cycles conditional on the inputs.*/
/*************************************************************/
/*  Allowable Target Device Types:p16r4                          */
/*************************************************************/

/**  Inputs  **/

PIN 1          = clk       ;      /* 2*5mhz clock                 */
PIN 2          = !r        ;      /* reset line                   */
PIN 3          = arqd      ;      /* ack required before tx of data */
PIN 4          = drxd      ;      /* data received                */
PIN 5          = !te       ;      /* empty from tx fifo buffer    */
PIN 6          = !rf       ;      /* full from rx fifo buffer     */

/**  Outputs  **/

PIN 12         = !asent    ;      /* ack sent                     */
PIN 13         = !tso      ;      /* Transmitter Shift Out        */
PIN 14         = !q0       ;      /* state var                    */
PIN 15         = !q1       ;      /* state var                    */
PIN 16         = !q2       ;      /* state var                    */
PIN 17         = !q3       ;      /* state var                    */
PIN 18         = !toe      ;      /* Transmitter Output Enable    */
PIN 19         = !pe       ;      /* parallel (load) enable to s.r. */

/** Declarations and Intermediate Variable Definitions **/

Field count = [q3,q2,q1,q0] ;

$define s0 'd'0
$define s1 'd'1
$define s2 'd'3
$define s3 'd'2
$define s4 'd'6
$define s5 'd'7
$define s6 'd'5
$define s7 'd'4
$define s8 'd'12
$define s9 'd'13
$define s10 'd'15
$define s11 'd'14
$define s12 'd'10
$define s13 'd'11
$define s14 'd'9
```

```
$define s15 'd'8


/**  Logic Equations  **/


count.d =
            r & count:s0  & s0
      #     r & count:s1  & s0
      #     r & count:s2  & s0
      #     r & count:s3  & s0
      #     r & count:s4  & s0
      #     r & count:s5  & s0
      #     r & count:s6  & s0
      #     r & count:s7  & s0
      #     r & count:s8  & s0
      #     r & count:s9  & s0
      #     r & count:s10 & s0
      #     r & count:s11 & s0
      #     r & count:s12 & s0
      #     r & count:s13 & s0
      #     r & count:s14 & s0
      #     r & count:s15 & s0
      #    !r & count:s1  & s2
      #    !r & count:s3  & s4
      #    !r & count:s4  & s5
      #    !r & count:s5  & s6
      #    !r & count:s6  & s7
      #    !r & count:s7  & s8
      #    !r & count:s8  & s9
      #    !r & count:s9  & s10
      #    !r & count:s10 & s11
      #    !r & count:s11 & s12
      #    !r & count:s12 & s13
      #    !r & count:s0 & s13
      #    !r & !rf & drxd & count:s2 & s1
      #    !r & !rf & drxd & count:s13 & s1
      #    !r & !arqd & !te & count:s2 & s3
      #    !r & (rf # !drxd) & (!arqd & !te) & count:s13 & s3
      #    !r & (te # arqd) & (!drxd # rf) & count:s2 & s13
      #    !r & (te # arqd) & (!drxd # rf) & count:s13 & s13 ;

asent = count:s2 # count:s0 ;

tso   = count:s5 ;

toe   = !(count:s1 # count:s2) ;

pe    = count:s1 # count:s3 ;
```

```
PARTNO    00003 ;
NAME      DECODE1A ;
DATE      08/01/87 ;
REV       00 ;
DESIGNER  G.A.Lester ;
COMPANY   University Of Salford ;
ASSEMBLY  00002 ;
LOCATION  00031 ;

/*************************************************************/
/* The decode functions required to interface the TRANSPUTER type */
/* links to a Z-80 system.                                   */
/* Generates the gate signal for an additional latch.        */
/*************************************************************/
/*  Allowable Target Device Types:16L8                       */
/*************************************************************/

/**  Inputs  **/

PIN 1         = a4          ;      /* address line a4          */
PIN 2         = a3          ;      /* address line a3          */
PIN 3         = a2          ;      /* address line a2          */
PIN 4         = a1          ;      /* address line a1          */
PIN 5         = a0          ;      /* address line a0          */
PIN 6         = !m1         ;      /* m1 cycle line            */
PIN 7         = !rd         ;      /* read line                */
PIN 8         = !wr         ;      /* write line               */
PIN 9         = !iorq       ;      /* iorq line                */
PIN 11        = !mreq       ;      /* mreq line                */

/**  Outputs  **/

PIN 12        = !elatch     ;      /* Latch (buffer empty signals)  */
PIN 13        = interm      ;      /* intermediate variable    */
PIN 14        = !roeB       ;      /* Rx Output Enable B       */
PIN 15        = rsoB        ;      /* Rx Shift Out B           */
PIN 16        = tsiB        ;      /* Ts Shift In B            */
PIN 17        = !roeA       ;      /* Rx Output Enable A       */
PIN 18        = rsoA        ;      /* Rx Shift Out A           */
PIN 19        = tsiA        ;      /* Tx Shift In A            */

/** Declarations and Intermediate Variable Definitions **/

Field addr = [a4,a3,a2,a1,a0] ;

/**  Logic Equations  **/

elatch  =  a4 & a0 & iorq & !m1 & rd ;

interm  = iorq & !m1 ;

rsoA    = !a4 & a0 & iorq & !m1 & rd ;

roeA    = !a4 & a0 & iorq & !m1 & rd ;

tsiA    = !a4 & a0 & iorq & !m1 & wr ;
```

```
rsoB    = !a4 & a1 & iorq & !m1 & rd ;

roeB    = !a4 & a1 & iorq & !m1 & rd ;

tsiB    = !a4 & a1 & iorq & !m1 & wr ;
```

```
PARTNO     00004 ;
NAME       DECODE1B ;
DATE       08/01/87 ;
REV        00 ;
DESIGNER   G.A.Lester ;
COMPANY    University Of Salford ;
ASSEMBLY   00002 ;
LOCATION   00032 ;

/*************************************************************/
/* The decode functions required to interface the TRANSPUTER style*/
/* links to a Z-80 system.                                   */
/* Generates the gate signal for an additional Latch         */
/*************************************************************/
/*  Allowable Target Device Types:16L8                       */
/*************************************************************/

/**  Inputs  **/

PIN 1          = a4          ;      /* address line a4        */
PIN 2          = a3          ;      /* address line a3        */
PIN 3          = a2          ;      /* address line a2        */
PIN 4          = a1          ;      /* address line a1        */
PIN 5          = a0          ;      /* address line a0        */
PIN 6          = !m1         ;      /* m1 cycle line          */
PIN 7          = !rd         ;      /* read line              */
PIN 8          = !wr         ;      /* write line             */
PIN 9          = !iorq       ;      /* iorq line              */
PIN 11         = !mreq       ;      /* mreq line              */

/**  Outputs  **/

PIN 12         = !flatch     ;      /* Latch (buffer full signals) */
PIN 13         = interm      ;      /* intermediate variable  */
PIN 14         = !roeD       ;      /* Rx Output Enable D     */
PIN 15         = rsoD        ;      /* Rx Shift Out D         */
PIN 16         = tsiD        ;      /* Ts Shift In D          */
PIN 17         = !roeC       ;      /* Rx Output Enable C     */
PIN 18         = rsoC        ;      /* Rx Shift Out C         */
PIN 19         = tsiC        ;      /* Tx Shift In C          */

/** Declarations and Intermediate Variable Definitions **/

Field addr = [a4,a3,a2,a1,a0] ;

/**  Logic Equations  **/

flatch  =  a4 & a1 & iorq & !m1 & rd ;

interm  = iorq & !m1 ;

rsoC    = !a4 & a2 & iorq & !m1 & rd ;

roeC    = !a4 & a2 & iorq & !m1 & rd ;

tsiC    = !a4 & a2 & iorq & !m1 & wr ;
```

```
rsoD    = !a4 & a3 & iorq & !m1 & rd ;

roeD    = !a4 & a3 & iorq & !m1 & rd ;

tsiD    = !a4 & a3 & iorq & !m1 & wr ;
```

```
PARTNO     00005 ;
NAME       DECODE2 ;
DATE       13/03/87 ;
REV        00 ;
DESIGNER   G.A.Lester ;
COMPANY    University Of Salford ;
ASSEMBLY   XXXX2 ;
LOCATION   XXX35 ;

/*************************************************************/
/* Memory decoding for Z-80 system.                          */
/* Also provides interrupt acknowledge signal.               */
/*                                                           */
/*************************************************************/
/*  Allowable Target Device Types: 16L8                      */
/*************************************************************/

/**  Inputs  **/

PIN 1          = a11        ;      /* address line a11        */
PIN 2          = a12        ;      /* address line a12        */
PIN 3          = a13        ;      /* address line a13        */
PIN 4          = a14        ;      /* address line a14        */
PIN 5          = a15        ;      /* address line a15        */
PIN 6          = !wr        ;      /* write line              */
PIN 7          = !rd        ;      /* read line               */
PIN 8          = !mreq      ;      /* mreq line               */
PIN 9          = !iorq      ;      /* iorq line               */
PIN 11         = !m1        ;      /* m1 cycle line           */

/**  Outputs  **/

PIN 12         = intack     ;      /* Interrupt acknowledge   */
PIN 13         = !romrd     ;      /* rom read                */
PIN 14         = !romce     ;      /* rom chip enable         */
PIN 17         = !ramwr     ;      /* ram write               */
PIN 18         = !ramrd     ;      /* ram read                */
PIN 19         = !ramce     ;      /* ram chip enable         */

/** Declarations and Intermediate Variable Definitions **/

/**  Logic Equations  **/

intack = m1 & iorq ;

romrd = mreq & !a15 & rd ;

romce = mreq & !a15 ;

ramwr = mreq & a15 & wr ;

ramrd = mreq & a15 & rd ;

ramce = mreq & a15 ;
```

```
PARTNO     00006 ;
NAME       SIMM1 ;
DATE       08/05/86 ;
REV        00 ;
DESIGNER   G.A.Lester ;
COMPANY    University Of Salford ;
ASSEMBLY   00001 ;
LOCATION   00027 ;

/*************************************************************/
/*The state machine to generate a pseudo Z80 I/O read/write    */
/*cycle simulation.  To be used in conjunction with simmZ80.   */
/*Used to drive the TRANSPUTER compatible links.               */
/*************************************************************/
/*  Allowable Target Device Types:p16r6                        */
/*************************************************************/

/**  Inputs  **/

PIN 1          = clk       ;      /* 5MHz clock (10 MHz ? )          */
PIN 2          = !r        ;      /* reset                           */
PIN 3          = !cts      ;      /* clear to send                   */
PIN 4          = tbre      ;      /* trans buffer requires extra     */
PIN 5          = dr        ;      /* data ready                      */
PIN 6          = !e        ;      /* empty                           */
PIN 7          = !f        ;      /* full                            */

/**  Outputs  **/

PIN 19         = !dtr      ;      /* data terminal ready             */
PIN 18         = !q0       ;      /* state var                       */
PIN 17         = !q1       ;      /* state var                       */
PIN 16         = !q2       ;      /* state var                       */
PIN 15         = !q3       ;      /* state var                       */
PIN 14         = !q4       ;      /* state var                       */
PIN 12         = etc       ;      /* empty or !tbre or !cts          */

/** Declarations and Intermediate Variable Definitions **/

etc = (e # !tbre # !cts);

Field count = [q4,q3,q2,q1,q0] ;

$define s0 'd'0
$define s1 'd'1
$define s2 'd'2
$define s3 'd'3
$define s4 'd'4
$define s5 'd'5
$define s6 'd'6
$define s7 'd'7
$define s8 'd'8
$define s9 'd'9
$define s10 'd'10
$define s11 'd'11
$define s12 'd'12
$define s13 'd'13
```

```
$define s14 'd'14
$define s15 'd'15
$define s16 'd'16
$define s17 'd'17
$define s18 'd'18
$define s19 'd'19
$define s20 'd'20
$define s21 'd'21

/**  Logic Equations  **/

count.d=
          r & count:s0  & s0
#   r & count:s1  & s0
#   r & count:s2  & s0
#   r & count:s3  & s0
#   r & count:s4  & s0
#   r & count:s5  & s0
#   r & count:s6  & s0
#   r & count:s7  & s0
#   r & count:s8  & s0
#   r & count:s9  & s0
#   r & count:s10 & s0
#   r & count:s11 & s0
#   r & count:s12 & s0
#   r & count:s13 & s0
#   r & count:s14 & s0
#   r & count:s15 & s0
#   r & count:s16 & s0
#   r & count:s17 & s0
#   r & count:s18 & s0
#   r & count:s19 & s0
#   r & count:s20 & s0
#   r & count:s21 & s0
# !r & count:s1  & s2
# !r & count:s2  & s3
# !r & count:s3  & s4
# !r & count:s4  & s5
# !r & count:s5  & s6
# !r & count:s6  & s7
# !r & count:s7  & s8
# !r & count:s8  & s9
# !r & count:s9  & s10
# !r & count:s10 & s11
# !r & count:s11 & s12
# !r & count:s14 & s15
# !r & count:s15 & s16
# !r & count:s16 & s17
# !r & count:s17 & s18
# !r & count:s18 & s19
# !r & count:s19 & s20
# !r & count:s20 & s21
# !r & count:s21 & s0
# !r & (!f & dr) & count:s0  & s1
# !r & (f # !dr) & !e & tbre & cts & count:s0 & s14
# !r & (f # !dr) & etc &  count:s0 & s0
# !r & !etc & count:s12 & s14
# !r & etc & count:s12 & s0  ;
```

```
dtr = !dr ;
```

```
PARTNO     00007 ;
NAME       SIMMZ80 ;
DATE       08/01/87 ;
REV        00 ;
DESIGNER   G.A.Lester ;
COMPANY    University Of Salford ;
ASSEMBLY   00000 ;
LOCATION   00000 ;

/*************************************************************/
/*The state machine to generate a pseudo Z80 I/O read/write  */
/*cycle simulation.  To be used in conjunction with simm1.   */
/*                                                           */
/*************************************************************/
/*  Allowable Target Device Types:p1618                      */
/*************************************************************/

/**  Inputs  **/

PIN 1          = !q0       ;      /* state var              */
PIN 2          = !q1       ;      /* state var              */
PIN 3          = !q2       ;      /* state var              */
PIN 4          = !q3       ;      /* state var              */
PIN 5          = !q4       ;      /* state var              */
PIN 6          = clk0      ;      /* from clk cct           */
PIN 7          = clk1      ;      /* from clk cct           */
PIN 8          = clk6      ;      /* from clk cct           */

/**  Outputs  **/

PIN 19         = clko      ;      /* to clk cct             */
PIN 18         = rrd       ;      /* received register disable */
PIN 17         = !drr      ;      /* data received reset    */
PIN 16         = !tbrl     ;      /* trans buffer register load */
PIN 15         = addr      ;      /* status line            */
PIN 14         = !iorq     ;      /* I/O request line       */
PIN 13         = !rd       ;      /* read data strobe       */
PIN 12         = !wr       ;      /* write data strobe      */

/** Declarations and Intermediate Variable Definitions **/

Field count = [q4,q3,q2,q1,q0] ;

$define s0 'd'0
$define s1 'd'1
$define s2 'd'2
$define s3 'd'3
$define s4 'd'4
$define s5 'd'5
$define s6 'd'6
$define s7 'd'7
$define s8 'd'8
$define s9 'd'9
$define s10 'd'10
$define s11 'd'11
$define s12 'd'12
$define s13 'd'13
```

```
$define s14 'd'14
$define s15 'd'15
$define s16 'd'16
$define s17 'd'17
$define s18 'd'18
$define s19 'd'19
$define s20 'd'20
$define s21 'd'21

/**  Logic Equations  **/

clko =clk6 &  clk0 ;

rrd  =!(count:s3 # count:s4 # count:s5 # count:s6
      # count:s7 # count:s8 # count:s9);

drr  =count:s10 ;

tbrl =count:s17 # count:s18 # count:s19 ;

addr =!(count:s2  # count:s10 # count:s11 # count:s12 # count:s13
      # count:s14 # count:s0  # count:s1);

iorq = count:s4  # count:s5  # count:s6  # count:s7  # count:s8
     # count:s16 # count:s17 # count:s18 # count:s19 # count:s20 ;

rd   = count:s16 # count:s17 # count:s18 # count:s19 # count:s20 ;

wr   = count:s4  # count:s5  # count:s6  # count:s7  # count:s8 ;
/*************************************************************/
/*                  PROGRAMS COMPILED USING:                 */
/* CUPL 2.1, ASSISTED TECHNOLOGY, DIVISION OF PERSONAL CAD   */
/* SYSTEMS INC., 1290 PARKMOOR AVE, SAN JOSE, CA.            */
/*************************************************************/
```

APPENDIX 11

1 Files Used with the Eco-C Compiler for Multiprocessing

```
.Z80
RAMTOP     EQU 9FFFH
STACKSIZE  EQU 01FFH
OSSTACK    EQU RAMTOP
S1         EQU OSSTACK - (STACKSIZE/2)
S2         EQU S1 - STACKSIZE
S3         EQU S2 - STACKSIZE
SMAIN      EQU S3 - STACKSIZE
TERMINATE  EQU 0028H
SPROC      EQU 0030H

           ENTRY BUFFERS

           EXTRN     PROC1              ;PROCESS ENTRIES OBTAINED
           EXTRN     PROC2              ;EXTERNALLY
           EXTRN     PROC3
           EXTRN     MAIN

           DSEG
CLOCK::    DEFW 0000
RUNNING::
           DEFB 01
NPROC::    DEFB 08
STACKLIST::                             ;A MAXIMUM OF FOUR PROCESSES
           DEFW 0101H
           DEFW 0202H
           DEFW 0303H
           DEFW 0404H
USERNPROC::                             ;STATIC VARS SHOULD BE LOADED AFTER
BUFFERS    EQU USERNPROC +1             ;THIS POINT

           CSEG
$INITA::
           DI
           LD SP,OSSTACK                ;SETS OPERATING SYSTEM STACK
           LD HL,0000                   ;INITIALISES PSEUDO CLOCK
           LD (CLOCK),HL
           LD A,(USERNPROC)             ;SETS THE RUNNING PROCESS AND
           LD (NPROC),A                 ;NPROC TO THE VALUE CONTAINED
           LD A,01                      ;IN USERNPROC
           LD (RUNNING),A
           LD HL,2
           LD DE,TERMINATE              ;SETS UP STACKS AND PROCLIST FOR
           LD SP,SMAIN                  ;EACH PROCESS
           PUSH DE
           LD BC,MAIN
           PUSH BC
           CALL PUTREG
           LD (STACKLIST+06),SP
           LD SP,S3                     ;IF ANY PROCESS RETURNS THEN
           PUSH DE                      ;THAT PROCESS WILL TERMINATE
```

```
LD BC,PROC3                    ;BY ENTERING A NON-BUSY WAIT
PUSH BC
CALL PUTREG
LD (STACKLIST+04),SP
LD SP,S2
PUSH DE
LD BC,PROC2
PUSH BC
CALL PUTREG
LD (STACKLIST+02),SP
LD SP,S1
PUSH DE
LD HL,02
EI
JP PROC1

PUTREG:
        POP IY
        PUSH AF
        EX AF,AF'
        PUSH AF
        PUSH BC
        PUSH DE
        PUSH HL
        EXX
        PUSH BC
        PUSH DE
        PUSH HL
        PUSH IX
        PUSH IY
        PUSH IY
        RET
        END $INITA
```

```
        LD HL,RUNNING                   ;FIND THE SPACE TO SAVE SP
        LD C,(HL)
        LD B,0
        ADD HL,BC
        ADD HL,BC
        EX DE,HL
        LD HL,0000
        ADD HL,SP                       ;SAVES STACK POINTER
        EX DE,HL
        LD (HL),E
        INC HL
        LD (HL),D
        LD HL,RUNNING                   ;TEST WHICH PROCESS IS RUNNING
        LD A,(HL)
        INC HL
        LD B,(HL)
        CP B
        JP NZ,NOTLAST
        DEC HL                          ;IF RUNNING = NPROC
        LD (HL),1                       ;SET RUNNING TO 01
        INC HL
        INC HL
        JR RESTORE                      ;RESTORE REGISTERS FOR PROC1
NOTLAST:
        DEC HL                          ;IF RUNNING LESS THAN NPROC
        INC (HL)                        ;INCREMENT RUNNING
        INC A                           ;INCREMENT A TO MATCH (A=RUNNING)
        LD C,A                          ;SET HL TO POINT TO SP FOR THE
        LD B,0                          ;NEW PROCESS
        ADD HL,BC
        ADD HL,BC
RESTORE:
        LD E,(HL)                       ;RESTORE REGISTERS USING THE VALUE
        INC HL                          ;OF SP STORED AT (HL)
        LD D,(HL)
        EX DE,HL
        LD SP,HL
        POP IY
        POP IX
        POP HL
        POP DE
        POP BC
        EXX
        POP HL
        POP DE
        POP BC
        POP AF
        EX AF,AF'
        POP AF
        EI                              ;RE-ENABLE INTERRUPTS AND
        RETI                            ;RETURN TO THE RUNNING PROCESS
                                        ;NON-OPERATING SYSTEM ROUTINES
                                        ;INITIALISATION ETC
SETPIO:                                 ;SET UP THE PIO (INACTIVE)
        LD A,0FFH
        OUT (20H),A
        OUT (20H),A
        OUT (60H),A
```

```
        OUT (60H),A
        LD A,17H
        OUT (20H),A
        OUT (60H),A
        LD A,0FFH
        OUT (20H),A
        OUT (60H),A
        RET

WAITPROG:                               ;WAIT FOR AND OFFER OF A PROGRAM
        IN A,(91H)
        AND A
        JR Z,WAITPROG
WPROM1: LD C,01
WPROM2: IN A,(91H)
        AND C
        JR Z,WPROM3
        SET 7,C
        IN A,(C)
        CP PROGOFFER
        JR Z,INPROG
        RES 7,C
WPROM3: RLC C
        LD A,C
        CP 10
        JR NZ,WPROM2
        JR WPROM1

INPROG: LD A,C                          ;SET C' TO THE ADDRESS OF THE CREATOR
        EXX
        LD C,A
        EXX
        LD A,GITM                       ;ACKNOWLEDGE THE OFFER OF A PROGRAM
        CALL OUTPUT

RLOOP1: CALL INPUT                      ;LOOP TO READ IN THE PROGRAM
        CP OFFH
        JR Z,ROM1
        CALL SEND
        JR RLOOP1
ROM1:   CALL INPUT
        CP OFFH
        JR NZ,ROM2
        CALL SEND
        JR RLOOP1
ROM2:   CP 00                           ;SETS MEMORY LOAD AND ADDRESS
        JR NZ,ROM3                      ;TO START AT
        CALL INPUT
        LD L,A
        CALL INPUT
        LD H,A
        LD IY,MEM
        JR RLOOP1
ROM3:   CP 01                           ;PUSHES A VALUE ONTO THE STACK
        JR NZ, ROM4
        CALL INPUT
        LD E,A
        CALL INPUT
```

```
        LD D,A
        PUSH DE
        JR RLOOP1
ROM4:   CP 02                       ;SETS THE STACK POINTER
        JR NZ, ROM5
        EX DE,HL
        CALL INPUT
        LD L,A
        CALL INPUT
        LD H,A
        LD SP,HL
        EX DE,HL
        JR RLOOP1
ROM5:   CP 03                       ;SETS I/O FORWARD AND ADDRESS
        JR NZ,ROM6                  ;HANDSHAKING ONLY GOOD WITH LINK ADDR
        CALL INPUT
        EXX
        LD C,A
        EXX
        LD IY,IO
        JR RLOOP1
ROM6:   CP 04                       ;PERFORMS A 'RET' INSTRUCTION
        JR NZ,ROM7
        RET
ROM7:   CP 05                       ;EXITS FROM THE PROGRAMMING LOOP
        JR NZ,ROM8
        JR REXIT1
ROM8:   CP 06                       ;STORES HIGHEST VALUE TO FORWARD
        JR NZ,ROM9                  ;IN DE'
        EXX
        CALL INPUT
        LD E,A
        CALL INPUT
        LD D,A
        EXX
        JR RLOOP1
ROM9:   CP 07                       ;STORES LOWEST VALUE TO FORWARD
        JR NZ,ROMA                  ;IN HL'
        EXX
        CALL INPUT
        LD L,A
        CALL INPUT
        LD H,A
        EXX
ROMA:   JP RLOOP1                   ;END OF LOOP TO READ IN PROGRAM
REXIT1: JR OFFERPROG                ;EXIT FROM LOOP

INPUT:  IN A,(91H)                  ;C=I/O ADDRESS TO READ FROM
        AND C                       ;A:=(C)
        AND OFH                     ;THE SUBROUTINE WAITS UNTIL A VALUE
        JR Z,INPUT                  ;IS AVAILABLE
        IN A,(C)
        RET

OUTPUT: EX AF,AF'                   ;C'=I/O ADDRESS TO WRITE TO
RETEST: LD B,OFH
WAITBIT:DJNZ WAITBIT
        IN A,(92H)                  ;(C'):=A
```

```
        AND C                       ;THE SUBROUTINE WAITS UNTIL THE OUTPUT
        AND OFH                     ;CHANNEL HAS ROOM FOR THE VALUE
        JR Z,RETEST
        EX AF,AF'
        OUT (C),A
        RET

SEND:   JP (IY)                     ;IY=ROUTINE TO CALL

MEM:    LD (HL),A                   ;A=VALUE TO SEND:HL=ADDRESS TO SEND TO
        INC HL                      ;(HL):=A:INC HL
        RET                         ;THE VALUE IS WRITTEN TO MEMORY
IO:     EXX                         ;A=VALUE TO SEND:C'=ADDRESS TO SEND TO
        CALL OUTPUT                 ;(C'):=A
        EXX                         ;THE VALUE IS SENT I/O
        RET

OFFERPROG:
        LD C,81H
        LD A,C
RLOOP2: EXX
        CP C                        ;C' CONTAINS THE ADDRESS OF THE CREATOR
        EXX
        JP Z,NEXT2
        LD A,PROGOFFER              ;SEND PROGOFFER TO (C)
        CALL OUTPUT
        LD B,0                      ;MAXIMUM TIMEOUT DELAY
        CALL AWAITANY
        INC B
        DJNZ NOTTIMEOUT             ;IF B=0 ON EXIT THEN TIMOUT OCCURED
        JR TIMEOUT
NOTTIMEOUT:                         ;IF B<>0 ON EXIT THEN NO TIMEOUT
        CP GITM                     ;CHECK TO SEE IF RESPONSE IS
        JR NZ,NOTHANKS              ;GIVE IT TO ME
        EXX
        PUSH HL
        PUSH DE
        EXX
        POP DE
        POP HL
        LD A,OFFH                   ;FF 00 L' H'
        CALL OUTPUT                 ;SETS LOAD ADDRESS
        LD A,00
        CALL OUTPUT
        LD A,L
        CALL OUTPUT
        LD A,H
        CALL OUTPUT
        LD A,OFFH                   ;FF 02 SP
        CALL OUTPUT                 ;SETS STACK POINTER
        LD A,02
        CALL OUTPUT
        LD HL,0000
        ADD HL,SP
        LD A,L
        CALL OUTPUT
        LD A,H
        CALL OUTPUT
```

```
        LD A,OFFH                       ;FF 01 (SP)
        CALL OUTPUT                     ;PUSHES A VALUE ON THE STACK
        LD A,01                         ;USED AS A STARTING ADDRESS
        CALL OUTPUT
        POP HL
        PUSH HL
        LD A,L
        CALL OUTPUT
        LD A,H
        CALL OUTPUT
        LD A,OFFH                       ;FF 06 E' D'
        CALL OUTPUT                     ;SETS HIGHEST ADDRESS TO FORWARD
        LD A,06
        CALL OUTPUT
        LD A,E
        CALL OUTPUT
        LD A,D
        CALL OUTPUT
        LD A,OFFH                       ;FF 07 L' H'
        CALL OUTPUT                     ;SETS LOWEST ADDRESS TO FORWARD
        LD A,07
        CALL OUTPUT
        EXX
        PUSH HL
        EXX
        POP HL
        LD A,L
        CALL OUTPUT
        LD A,H
        CALL OUTPUT
        CALL TCLOOP
        LD A,OFFH                       ;FF 05
        CALL OUTPUT                     ;CAUSES AN EXIT FROM THE PROGRAMMING
        LD A,05                         ;LOOP
        CALL OUTPUT

TIMEOUT:
NOTHANKS:
NEXT2:  LD A,C
        AND OFH
        RLCA
        CP 10H
        JR Z,REXIT2
        OR 80H
        LD C,A
        JP RLOOP2
REXIT2: JR STARTUP

AWAITANY:
        IN A,(91H)
        AND C
        AND OFH
        JR NZ,FOUNDANY
        DJNZ AWAITANY
        RET
FOUNDANY:
        IN A,(C)
        RET
```

```
TCLOOP: LD A,(HL)
        CP OFFH
        JR NZ,SINGLE
        CALL OUTPUT
SINGLE: CALL OUTPUT
        INC HL
        LD A,H
        CP D
        JR NZ,TCLOOP
        LD A,L
        CP E
        JR NZ,TCLOOP
        RET

STARTUP:
        RET                          ;PERFORMS A 'RET' TO THE PROGRAM
                                     ;POSSIBLY STRIP ALL PROGOFFERS?
        END $RESET
```

APPENDIX 12

12 Communication Functions: Byte Wise Versions

```
        .Z80
        SPROC EQU 30H
        EXTRN BUFFERS

;FUNCTION INT RTBYTE(PROC,LINK)
;PARAMETERS EXPECTED AS INTEGERS AND RETURNS INTEGER
;RECEIVES A BYTE, ONLY TESTS, DOES NOT WAIT IF NO VALUE AVAILABLE
;IF A BYTE IS WAITING THEN THE FUNCTION RETURNS THE VALUE RECEIVED
;IF THERE IS NO BYTE THEN A VALUE GREATER THAN 255 IS RETURNED

        CSEG
RTBYTE::
        ADD HL,SP
        PUSH HL
        LD HL,0004
        ADD HL,SP
        LD A,(HL)                       ;A:=PROC
        AND A
        JR NZ,RTINT
        LD HL,04 + 02                   ;EXTERNAL COMMUNICATION
        ADD HL,SP
        LD C,(HL)                       ;C:=LINK ADDRESS
RT1:    DI                              ;DI-PREVENTS INTERFERENCE
        IN A,(91H)                      ;TEST THE BUFFER FLAGS
        AND C
        JR NZ,RT2                       ;IF NO VALUE THEN
        EI                              ;EI-AGAIN
        LD HL,0100H
        JP $RTNI##                      ;RETURN 0100H
RT2:    SET 7,C                         ;IF VALUE THEN
        IN L,(C)                        ;RECEIVE VALUE
        EI                              ;EI-AGAIN
        LD H,00
        JP $RTNI##                      ;RETURN VALUE
RTINT:                                  ;INTERNAL COMMUNICATION
        LD HL,04 + 02
        ADD HL,SP
        LD C,(HL)                       ;C:=LINK ADDRESS
        LD B,A                          ;B:=PROC
        LD HL,BUFFERS - 09H
        LD DE,0009H
RT3:    ADD HL,DE                       ;SKIP OVER BUFFERS
        DJNZ RT3
        INC HL
RT4:    DI                              ;DI-PREVENTS INTERFERENCE
        LD A,(HL)                       ;TEST BUFFER FLAGS
        AND C
        JR Z,RT5                        ;IF NO VALUE THEN
        EI                              ;EI-AGAIN
        LD HL,0100H
        JP $RTNI##                      ;RETURN 0100H
```

```
RT5:    LD A,C                  ;IF VALUE THEN
        XOR (HL)                ;ADJUST FLAGS
        LD (HL),A
        LD A,C
RT6:    INC HL                  ;SET HL TO POINT TO BUFFER LOCATION
        RRA
        JR NC,RT6
        LD L,(HL)               ;RECEIVE VALUE
        EI                      ;EI-AGAIN
        LD H,00
        JP $RTNI##              ;RETURN VALUE
        END RTBYTE
```

```
        .Z80
        SPROC EQU 30H
        EXTRN BUFFERS

;FUNCTION INT RWBYTE(PROC,LINK)
;PARAMETERS EXPECTED AS INTEGERS AND RETURNS INTEGER
;RECEIVES A BYTE, WAITS IF NO VALUE AVAILABLE
;RETURNS THE VALUE RECIEVED

        CSEG
RWBYTE::
        ADD HL,SP
        PUSH HL
        LD HL,0004
        ADD HL,SP
        LD A,(HL)                   ;A:=PROC
        AND A
        JR NZ,RWINT
        LD HL,04+02                 ;EXTERNAL COMMUNICATION
        ADD HL,SP
        LD C,(HL)                   ;C:=LINK ADDRESS
RW1:    DI                          ;DI-PREVENTS INTERFERENCE
        IN A,(91H)                  ;TEST THE BUFFER FLAGS
        AND C
        JR NZ,RW2                   ;IF NO VALUE THEN
        EI                          ;EI-AGAIN
        RST SPROC                   ;RELINQUISH TIME SLICE
        JR RW1                      ;TRY AGAIN
RW2:    SET 7,C                     ;IF SPACE THEN
        IN L,(C)                    ;RECEIVE VALUE
        EI                          ;EI-AGAIN
        LD H,00
        JP $RTNI##                  ;RETURN VALUE
RWINT:                              ;INTERNAL COMMUNICATION
        LD HL,04+02
        ADD HL,SP
        LD C,(HL)                   ;C:=LINK ADDRESS
        LD B,A                      ;B:=PROC
        LD HL,BUFFERS - 09H
        LD DE,0009H
RW3:    ADD HL,DE                   ;SKIP OVER BUFFERS
        DJNZ RW3
        INC HL
RW4:    DI                          ;DI-PREVENTS INTERFERENCE
        LD A,(HL)                   ;TEST BUFFER FLAGS
        AND C
        JR Z,RW5                    ;IF NO VALUE THEN
        EI                          ;EI-AGAIN
        RST SPROC                   ;RELINQUISH TIME SLICE
        JR RW4                      ;TRY AGAIN
RW5:    LD A,C                      ;IF VALUE THEN
        XOR (HL)                    ;ADJUST FLAGS
        LD (HL),A
        LD A,C
RW6:    INC HL                      ;SET HL TO POINT TO BUFFER LOCATION
        RRA
```

```
JR NC,RW6
LD L,(HL)                    ;RECEIVE VALUE
EI                           ;EI-AGAIN
LD H,00
JP $RTNI##                   ;RETURN VALUE
END RWBYTE
```

```
        .Z80
        SPROC EQU 30H
        EXTRN BUFFERS

;FUNCTION INT STBYTE(PROC,LINK,VALUE)
;PARAMETERS EXPECTED AS INTEGERS AND RETURNS INTEGER
;SENDS A BYTE, TESTS FOR SPACE, DOES NOT WAIT IF NO SPACE AVAILABLE
;IF THERE IS SPACE A BYTE IS SENT AND A VALUE LESS THAN 255 RETURNED
;IF THERE IS NO SPACE A VALUE GREATER THEN 255 IS RETURNED

        CSEG
STBYTE::
        ADD HL,SP
        PUSH HL
        LD HL,0004H
        ADD HL,SP
        LD A,(HL)                   ;A:=PROC
        AND A
        JR NZ,STINT
        LD HL,04+02                 ;EXTERNAL COMMUNICATION
        ADD HL,SP
        LD C,(HL)                   ;C:=LINK ADDRESS
ST1:    DI                          ;DI-PREVENTS INTERFERENCE
        IN A,(92H)                  ;TEST THE BUFFER FLAGS
        AND C
        JR NZ,ST2                   ;IF NO SPACE THEN
        EI                          ;EI-AGAIN
        LD HL,0100H
        JP $RTNI##                  ;RETURN 0100H
ST2:    SET 7,C                     ;IF SPACE THEN
        LD HL,04 + 04
        ADD HL,SP
        OUTI                        ;SEND VALUE
        EI                          ;EI-AGAIN
        LD HL,0000H
        JP $RTNI##                  ;RETURN 0000
STINT:                              ;INTERNAL COMMUNICATION
        LD HL,04 + 02
        ADD HL,SP
        LD C,(HL)                   ;C:=LINK ADDRESS
        LD B,A                      ;B:=PROC
        LD HL,BUFFERS - 09H
        LD DE,0009H
ST3:    ADD HL,DE                   ;SKIP OVER BUFFERS
        DJNZ ST3
        INC HL
ST4:    DI                          ;DI-PREVENTS INTERFERENCE
        LD A,(HL)                   ;TEST BUFFER FLAGS
        AND C
        JR NZ, ST5                  ;IF NO SPACE THEN
        EI                          ;EI-AGAIN
        LD HL,0100H
        JP $RTNI##                  ;RETURN 0100H
ST5:    LD A,C                      ;IF SPACE THEN
        XOR (HL)                    ;ADJUST FLAGS
        LD (HL),A
```

```
        LD A,C
ST6:    INC HL                          ;SET HL TO POINT TO BUFFER LOCATION
        RRA
        JR NC,ST6
        EX DE,HL
        LD HL,04 + 04
        ADD HL,SP
        LDI                             ;COPY VALUE INTO BUFFER
        EI                              ;EI-AGAIN
        LD HL,0000H
        JP $RTNI##                      ;RETURN 0000
        END STBYTE
```

```
        .Z80
        SPROC EQU 30H
        EXTRN BUFFERS

;FUNCTION VOID SWBYTE(PROC,LINK,VAL)
;PARAMETERS EXPECTED AS INTEGERS AND RETURNS VOID
;SENDS A BYTE, WAITS UNTIL DESTINATION FREE IF BUSY

        CSEG
SWBYTE::
        ADD HL,SP
        PUSH HL
        LD HL,0004
        ADD HL,SP
        LD A,(HL)                       ;A:=PROC
        AND A
        JR NZ,SWINT
        LD HL,04+02                     ;EXTERNAL COMMUNICATION
        ADD HL,SP
        LD C,(HL)                       ;C:=LINK ADDRESS
SW1:    DI                              ;DI-PREVENTS INTERFERENCE
        IN A,(92H)                      ;TEST THE BUFFER FLAGS
        AND C
        JR NZ,SW2                       ;IF NO SPACE THEN
        EI                              ;EI-AGAIN
        RST SPROC                       ;RELINQUISH TIME SLICE
        JR SW1                          ;TRY AGAIN
SW2:    SET 7,C                         ;IF SPACE THEN
        LD HL,04 + 04
        ADD HL,SP
        OUTI                            ;SEND VALUE
        EI                              ;EI-AGAIN
        JP $RTNV##
SWINT:                                  ;INTERNAL COMMUNICATION
        LD HL,04 + 02
        ADD HL,SP
        LD C,(HL)                       ;C:=LINK ADDRESS
        LD B,A                          ;B:=PROC
        LD HL,BUFFERS - 09H
        LD DE,0009H
SW3:    ADD HL,DE                       ;SKIP OVER BUFFERS
        DJNZ SW3
        INC HL
SW4:    DI                              ;DI-PREVENTS INTERFERENCE
        LD A,(HL)                       ;TEST BUFFER FLAGS
        AND C
        JR NZ,SW5                       ;IF NO SPACE THEN
        EI                              ;EI-AGAIN
        RST SPROC                       ;RELINQUISH TIME SLICE
        JR SW4                          ;TRY AGAIN
SW5:    LD A,C                          ;IF SPACE THEN
        XOR (HL)                        ;ADJUST FLAGS
        LD (HL),A
        LD A,C
SW6:    INC HL                          ;SET HL TO BUFFER LOCATION
        RRA
```

```
JR NC,SW6
EX DE,HL
LD HL,04 + 04
ADD HL,SP
LDI                                ;COPY VALUE INTO BUFFER
EI                                 ;EI-AGAIN
JP $RTNV##
END SWBYTE
```

```
/* static definitions for inter-process communication buffer
   and flags.
   definition for process proc1
single byte version for use with rtbyte, stbyte, rwbyte, swbyte
*/
        static short lflg1 = 255 ;
        static short buff1[8] = {0,0,0,0,0,0,0,0} ;
```

APPENDIX 13

13 Communication Functions: Fifo Buffered Versions

```
        .Z80
        SPROC EQU 30H
        EXTRN BUFFERS

;FUNCTION INT RTBYTE(PROC,LINK)
;PARAMETERS EXPECTED AS INTEGERS AND RETURNS INTEGER
;RECEIVES A BYTE, ONLY TESTS, DOES NOT WAIT IF NO VALUE AVAILABLE
;IF A BYTE IS WAITING THEN THE FUNCTION RETURNS THE VALUE RECEIVED
;IF THERE IS NO BYTE THEN A VALUE GREATER THAN 255 IS RETURNED
;VERSION FOR USE WITH FIFO BUFFERING

        CSEG
RTBYTE::
        ADD HL,SP
        PUSH HL
        LD HL,0004
        ADD HL,SP
        LD A,(HL)                       ;A:=PROC
        AND A
        JR NZ,RTINTL
        LD HL,04+02                     ;EXTERNAL COMMUNICATION
        ADD HL,SP
        LD C,(HL)                       ;C:=LINK ADDRESS
RTL1:   DI                              ;DI-PREVENTS INTERFERENCE
        IN A,(91H)                      ;TEST BUFFER FLAGS
        AND C
        JR NZ,RTL2                      ;IF NO VALUE THEN
        EI                              ;EI-AGAIN
        LD HL,0100H
        JP $RTNI##                      ;RETURN 100H
RTL2:   SET 7,C                         ;IF VALUE THEN
        IN L,(C)                        ;RECEIVE VALUE
        EI                              ;EI-AGAIN
        LD H,00
        JP $RTNI##                      ;RETURN VALUE
RTINTL:                                 ;INTERNAL COMMUNICATION
        LD HL,04+02
        ADD HL,SP
        LD C,(HL)                       ;C:=LINK ADDRESS
        LD B,A                          ;B:=PROC
        LD HL,BUFFERS
        LD DE,0012H
        DJNZ RTL3                       ;IF PROC=1 THEN
        JR RTL6                         ;GO STRAIGHT ON
RTL3:   PUSH BC                         ;ELSE
        LD B,(HL)                       ;SKIP OVER OTHER PROCESSES BUFFERS
        INC HL
RTL4:   ADD HL,DE
        DJNZ RTL4
RTL5:   POP BC
        DJNZ RTL3
```

```
RTL6:   INC HL
RTL7:   BIT 0,C                 ;FIND BOTTOM OF LINK STRUCTURE
        JR NZ,RTL8
        ADD HL,DE
        RRC C
        JR RTL7
RTL8:   DI                      ;DI-PREVENTS INTERFERENCE
        LD B,(HL)               ;B:=HEAD
        INC HL
        LD A,(HL)               ;A:=TAIL
RTL9:   CP B                    ;TEST FOR BUFFER EMPTY (HEAD=TAIL)
        JR NZ,RTLA              ;IF NO VALUE THEN
        EI                      ;EI-AGAIN
        LD HL,0100H
        JP $RTNI##              ;RETURN 0100H
RTLA:   DI                      ;DI-PREVENTS INTERFERENCE
        DEC HL                  ;IF VALUE THEN
        LD A,(HL)
        LD D,00
        LD E,A
        INC A                   ;INCREMENT HEAD
        CP 11H                  ;MODULO 16
        JR C,RTLB
        LD A,01
RTLB:   LD (HL),A
        INC HL
        ADD HL,DE               ;SET HL TO POINT TO HEAD ADDRESS
        LD E,(HL)
        EI                      ;EI-AGAIN
        EX DE,HL
        JP $RTNI##              ;RETURN VALUE
        END RTBYTE
```

```
          .Z80
          SPROC EQU 30H
          EXTRN BUFFERS

;FUNCTION INT RWBYTE(PROC,LINK)
;PARAMETERS EXPECTED AS INTEGERS AND RETURNS INTEGER
;RECEIVES A BYTE, WAITS IF NO VALUE AVAILABLE
;RETURNS THE VALUE RECEIVED
;VERSION FOR USE WITH FIFO BUFFERING

          CSEG
RWBYTE::
          ADD HL,SP
          PUSH HL
          LD HL,0004
          ADD HL,SP
          LD A,(HL)                    ;A:=PROC
          AND A
          JR NZ,RWINTL
          LD HL,04+02                  ;EXTERNAL COMMUNICATION
          ADD HL,SP
          LD C,(HL)                    ;C:=LINK ADDRESS
RWL1:     DI                           ;DI-PREVENTS INTERFERENCE
          IN A,(91H)                   ;TEST BUFFER FLAGS
          AND C
          JR NZ,RWL2                   ;IF NO VALUE THEN
          EI                           ;EI-AGAIN
          RST SPROC                    ;RELINQUISH TIME SLICE
          JR RWL1                      ;TRY AGAIN
RWL2:     SET 7,C                      ;IF VALUE THEN
          IN L,(C)                     ;RECEIVE VALUE
          EI                           ;EI-AGAIN
          LD H,00
          JP $RTNI##                   ;RETURN VALUE
RWINTL:                                ;INTERNAL COMMUNICATION
          LD HL,04+02
          ADD HL,SP
          LD C,(HL)                    ;C:=LINK ADDRESS
          LD B,A                       ;B:=PROC
          LD HL,BUFFERS
          LD DE,0012H
          DJNZ RWL3                    ;IF PROC=1 THEN
          JR RWL6                      ;GO STRAIGHT ON
RWL3:     PUSH BC                      ;ELSE
          LD B,(HL)                    ;SKIP OVER OTHER PROCESSES BUFFERS
          INC HL
RWL4:     ADD HL,DE
          DJNZ RWL4
RWL5:     POP BC
          DJNZ RWL3
RWL6:     INC HL
RWL7:     BIT 0,C                      ;FIND BOTTOM OF LINK STRUCTURE
          JR NZ,RWL8
          ADD HL,DE
          RRC C
          JR RWL7
```

```
RWL8:   DI                      ;DI-PREVENTS INTERFERENCE
        LD B,(HL)               ;B:=HEAD
        INC HL
        LD A,(HL)               ;A:=TAIL
RWL9:   CP B                    ;TEST FOR BUFFER EMPTY (HEAD=TAIL)
        JR NZ,RWLA              ;IF NO VALUE THEN
        EI                      ;EI-AGAIN
        RST SPROC               ;RELINQUISH TIME SLICE
        DEC HL
        JR RWL8                 ;TRY AGAIN
RWLA:   DI                      ;DI-PREVENTS INTERFERENCE
        DEC HL                  ;IF VALUE THEN
        LD A,(HL)
        LD D,00
        LD E,A
        INC A                   ;INCREMENT HEAD
        CP 11H                  ;MODULO 16
        JR C,RWLB
        LD A,01
RWLB:   LD (HL),A
        INC HL
        ADD HL,DE               ;SET HL TO POINT TO HEAD ADDRESS
        LD E,(HL)
        EI                      ;EI-AGAIN
        EX DE,HL
        JP $RTNI##              ;RETURN VALUE
        END RWBYTE
```

```
        .Z80
        SPROC EQU 30H
        EXTRN BUFFERS

;FUNCTION INT STBYTE(PROC,LINK,VALUE)
;PARAMETERS EXPECTED AS INTEGERS AND RETURNS INTEGER
;SENDS A BYTE, TESTS FOR SPACE, DOES NOT WAIT IF NO SPACE AVAILABLE
;IF THERE IS SPACE A BYTE IS SENT AND A VALUE LESS THAN 255 RETURNED
;IF THERE IS NO SPACE A VALUE GREATER THEN 255 IS RETURNED
;VERSION FOR USE WITH FIFO BUFFERING

        CSEG
STBYTE::
        ADD HL,SP
        PUSH HL
        LD HL,0004
        ADD HL,SP
        LD A,(HL)                   ;A:=PROC
        AND A
        JR NZ,STINTL
        LD HL,04+02                 ;EXTERNAL COMMUNICATION
        ADD HL,SP
        LD C,(HL)                   ;C:=LINK ADDRESS
STL1:   DI                          ;DI-PREVENTS INTERFERENCE
        IN A,(92H)                  ;TEST THE BUFFER FLAGS
        AND C
        JR NZ,STL2                  ;IF NO SPACE THEN
        EI                          ;EI-AGAIN
        LD HL,0100H
        JP $RTNI##                  ;RETURN 0100H
STL2:   SET 7,C                     ;IF SPACE THEN
        LD HL,04+04
        ADD HL,SP
        OUTI                        ;SEND VALUE
        EI                          ;EI-AGAIN
        LD HL,0000H
        JP $RTNI##                  ;RETURN 0000
STINTL:                             ;INTERNAL COMMUNICATION
        LD HL,04+02
        ADD HL,SP
        LD C,(HL)                   ;C:=LINK ADDRESS
        LD B,A                      ;B:=PROC
        LD HL,BUFFERS
        LD DE,0012H
        DJNZ STL3                   ;IF PROC=1 THEN
        JR STL6                     ;GO STRAIGHT ON
STL3:   PUSH BC                     ;ELSE
        LD B,(HL)                   ;SKIP OVER OTHER PROCESSES BUFFERS
        INC HL
STL4:   ADD HL,DE
        DJNZ STL4
STL5:   POP BC
        DJNZ STL3
STL6:   INC HL
STL7:   BIT 0,C                     ;FIND BOTTOM OF LINK STRUCTURE
        JR NZ,STL8
```

```
        ADD HL,DE
        RRC C
        JR STL7
STL8:   DI                          ;DI-PREVENTS INTERFERENCE
        LD B,(HL)                   ;B:=HEAD
        INC HL
        LD A,(HL)                   ;A:=TAIL
        LD D,00
        LD E,A
        INC A                       ;INCREMENT TAIL
        CP 11H                      ;MODULO 10H
        JR C,STL9
        LD A,01
STL9:   CP B                        ;TEST FOR BUFFER FULL (HEAD=TAIL+1)
        JR NZ,STLA                  ;IF NO SPACE THEN
        EI                          ;EI-AGAIN
        LD HL,0100H
        JP $RTNI##                  ;RETURN 0100H
STLA:   DI                          ;DI-PREVENTS INTERFERENCE
        LD (HL),A
        ADD HL,DE                   ;SET HL TO POINT TO BUFFER LOCATION
        EX DE,HL
        LD HL,04+04
        ADD HL,SP
        LDI                         ;COPY VALUE INTO THE BUFFER
        EI                          ;EI-AGAIN
        LD HL,0000
        JP $RTNI##                  ;RETURN 0000H
        END STBYTE
```

```
        .Z80
        SPROC EQU 30H
        EXTRN BUFFERS

;FUNCTION INT STBYTE(PROC,LINK,VALUE)
;PARAMETERS EXPECTED AS INTEGERS AND RETURNS INTEGER
;SENDS A BYTE, TESTS FOR SPACE, DOES NOT WAIT IF NO SPACE AVAILABLE
;IF THERE IS SPACE A BYTE IS SENT AND A VALUE LESS THAN 255 RETURNED
;IF THERE IS NO SPACE A VALUE GREATER THEN 255 IS RETURNED
;VERSION FOR USE WITH FIFO BUFFERING

        CSEG
STBYTE::
        ADD HL,SP
        PUSH HL
        LD HL,0004
        ADD HL,SP
        LD A,(HL)                   ;A:=PROC
        AND A
        JR NZ,STINTL
        LD HL,04+02                 ;EXTERNAL COMMUNICATION
        ADD HL,SP
        LD C,(HL)                   ;C:=LINK ADDRESS
STL1:   DI                          ;DI-PREVENTS INTERFERENCE
        IN A,(92H)                  ;TEST THE BUFFER FLAGS
        AND C
        JR NZ,STL2                  ;IF NO SPACE THEN
        EI                          ;EI-AGAIN
        LD HL,0100H
        JP $RTNI##                  ;RETURN 0100H
STL2:   SET 7,C                     ;IF SPACE THEN
        LD HL,04+04
        ADD HL,SP
        OUTI                        ;SEND VALUE
        EI                          ;EI-AGAIN
        LD HL,0000H
        JP $RTNI##                  ;RETURN 0000
STINTL:                             ;INTERNAL COMMUNICATION
        LD HL,04+02
        ADD HL,SP
        LD C,(HL)                   ;C:=LINK ADDRESS
        LD B,A                      ;B:=PROC
        LD HL,BUFFERS
        LD DE,0012H
        DJNZ STL3                   ;IF PROC=1 THEN
        JR STL6                     ;GO STRAIGHT ON
STL3:   PUSH BC                     ;ELSE
        LD B,(HL)                   ;SKIP OVER OTHER PROCESSES BUFFERS
        INC HL
STL4:   ADD HL,DE
        DJNZ STL4
STL5:   POP BC
        DJNZ STL3
STL6:   INC HL
STL7:   BIT 0,C                     ;FIND BOTTOM OF LINK STRUCTURE
        JR Z,STL8
```

```
        ADD HL,DE
        RRC C
        JR STL7
STL8:   DI                          ;DI-PREVENTS INTERFERENCE
        LD B,(HL)                   ;B:=HEAD
        INC HL
        LD A,(HL)                   ;A:=TAIL
        LD D,00
        LD E,A
        INC A                       ;INCREMENT TAIL
        CP 11H                      ;MODULO 10H
        JR C,STL9
        LD A,01
STL9:   CP B                        ;TEST FOR BUFFER FULL (HEAD=TAIL+1)
        JR NZ,STLA                  ;IF NO SPACE THEN
        EI                          ;EI-AGAIN
        LD HL,0100H
        JP $RTNI##                  ;RETURN 0100H
STLA:   DI                          ;DI-PREVENTS INTERFERENCE
        LD (HL),A
        ADD HL,DE                   ;SET HL TO POINT TO BUFFER LOCATION
        EX DE,HL
        LD HL,04+04
        ADD HL,SP
        LDI                         ;COPY VALUE INTO THE BUFFER
        EI                          ;EI-AGAIN
        LD HL,0000
        JP $RTNI##                  ;RETURN 0000H
        END STBYTE
```

```
        .Z80
        SPROC EQU 30H
        EXTRN BUFFERS

;FUNCTION VOID SWBYTE(PROC,LINK,VALUE)
;PARAMETERS EXPECTED AS INTEGERS AND RETURNS VOID
;SENDS A BYTE, WAITS UNTIL DESTINATION FREE IF BUSY
;VERSION FOR USE WITH FIFO BUFFERING

        CSEG
SWBYTE::
        ADD HL,SP
        PUSH HL
        LD HL,0004
        ADD HL,SP
        LD A,(HL)                       ;A:=PROC
        AND A
        JR NZ,SWINTL
        LD HL,04+02                     ;EXTERNAL COMMUNICATION
        ADD HL,SP
        LD C,(HL)                       ;C:=LINK ADDRESS
SWL1:   DI                              ;DI-PREVENTS INTERFERENCE
        IN A,(92H)                      ;TEST THE BUFFER FLAGS
        AND C
        JR NZ,SWL2                      ;IF NO SPACE THEN
        EI                              ;EI-AGAIN
        RST SPROC                       ;RELINQUISH TIME SLICE
        JR SWL1                         ;TRY AGAIN
SWL2:   SET 7,C                         ;IF SPACE THEN
        LD HL,04+04
        ADD HL,SP
        OUTI                            ;SEND VALUE
        EI                              ;EI-AGAIN
        JP $RTNV##
SWINTL:                                 ;INTERNAL COMMUNICATION
        LD HL,04+02
        ADD HL,SP
        LD C,(HL)                       ;C:=LINK ADDRESS
        LD B,A                          ;B:=PROC
        LD HL,BUFFERS
        LD DE,0012H
        DJNZ SWL3                       ;IF PROC=1 THEN
        JR SWL6                         ;GO STRAIGHT ON
SWL3:   PUSH BC                         ;ELSE
        LD B,(HL)                       ;SKIP OVER OTHER PROCESSES BUFFERS
        INC HL
SWL4:   ADD HL,DE
        DJNZ SWL4
SWL5:   POP BC
        DJNZ SWL3
SWL6:   INC HL
SWL7:   BIT 0,C                                 ;FIND BOTTOM OF LINK STRUCTURE
        JR NZ,SWL8
        ADD HL,DE
        RRC C
        JR SWL7
```

```
SWL8:   DI                      ;DI-PREVENTS INTERFERENCE
        LD B,(HL)               ;B:=HEAD
        INC HL
        LD A,(HL)               ;A:=TAIL
        LD D,00
        LD E,A
        INC A                   ;INCREMENT TAIL
        CP 11H                  ;MODULO 10H
        JR C,SWL9
        LD A,01
SWL9:   CP B                    ;TEST FOR BUFFER FULL (HEAD=TAIL+1)
        JR NZ,SWLA              ;IF NO SPACE THEN
        EI                      ;EI-AGAIN
        RST SPROC               ;RELINQUISH TIME SLICE
        DEC HL
        JR SWL8                 ;TRY AGAIN
SWLA:   DI                      ;DI-PREVENTS INTERFERENCE
        LD (HL),A
        ADD HL,DE               ;SET HL TO POINT TO BUFFER LOCATION
        EX DE,HL
        LD HL,04+04
        ADD HL,SP
        LDI                     ;COPY VALUE INTO THE BUFFER
        EI                      ;EI-AGAIN
        JP $RTNV##
        END SWBYTE
```

```
/* static definitions for inter-process communication buffer and
head and tail byte.
definition for process proc1
multiple byte version for use with rtfifo, stfifo, rwfifo, swfifo
*/
        static short nlink1    = 01 ;
        static short q1h1      = 01 ;
        static short q1t1      = 01 ;
        static short bf1p1[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0} ;
```

APPENDIX 14

14 Programs Used to Interface to the System

The following function was written by Dr C. A. Owen and is reproduced by kind permission.

```
        TITLE   ECO-C BIOS entry
        .Z80

CCALL::
        ;Address of CALL,AF,BC,DE and HL passed in
        ADD     HL,SP
        PUSH    HL
        LD      HL,4
        ADD     HL,SP           ;Pointer address
        LD      A,(HL)
        INC     HL
        LD      H,(HL)
        LD      L,A             ;HL = pointer address
        PUSH    HL              ;Address of call
        POP     IX              ;Use IX as pointer
        LD      L,(IX+0)
        LD      H,(IX+1)        ;HL = address to call
        PUSH    IX              ;(Need to recover this later)
        LD      DE,EXIT
        PUSH    DE              ;Set return point
        PUSH HL                 ;RET to it later
        LD      A,(IX+3)
        LD      C,(IX+4)
        LD      B,(IX+5)
        LD      E,(IX+6)
        LD      D,(IX+7)
        LD      L,(IX+8)
        LD      H,(IX+9)
        RET                     ;jumps to routine
EXIT:
        ;Comes here after routine called
        POP     IX              ;Recover pointer
        LD      (IX+9),H
        LD      (IX+8),L
        LD      (IX+7),D
        LD      (IX+6),E
        LD      (IX+5),B
        LD      (IX+4),C
        PUSH    AF
        POP     HL              ;To get flags
        LD      (IX+3),A
        LD      (IX+2),L        ;Flag register
        JP      $RTNV##

        END
```

```
/* stdio.h containing the required definitions for the */
/* Function CCALL */
/******************** some system definitions ********************/

                #define NULL    0
                #define FALSE   0
                #define TRUE    1
                #define EOF     (-1)
                #define CPMEOF  (0x1a)

/****************** the structure for allocations **************/

                typedef int     ALIGN;

                union   header  {
                        struct  {
                                union   header  *ptr;
                                unsigned        size;
                        }s;
                        ALIGN   x;
                };

                typedef union header HEADER;


/********************** the iob structure ************************/

                struct  iobbuf{
                        int     _cnt;
                        char    *_ptr;
                        char    *_base;
                        int     _flag;
                        int     _fd;
                        };

                #define _NFILE  6
                                                /* maximum number of files */

                extern  int     _nfile;         /* this variable yields the
                                                    number of files set in
                                                    the library */

                typedef struct  iobbuf  FILE;

                extern  FILE    _iob[];

/************************** the iob definitions ********************/

                #define stdin   &_iob[0]
                #define stdout  &_iob[1]
                #define stderr  &_iob[2]
                #define stdlst  &_iob[3]

                #define _READ   01
                                                /* file open for reading */
                #define _WRITE  02
                                                /* file open for writing */
                #define _UNBUF  04
```

```
#define _BBUF   010                     /* file is unbuffered */

                                        /* a big buffer was
                                           allocated */
#define _EOF    020
                                        /* EOF has occurred on this
                                           file */
#define _ERR    040
                                        /* error has occurred on this
                                            file */
#define _WFLAG  0100
                                        /* buffer has been written
                                           to */
#define  BFLAG  0200
                                        /* if true then binary read/
                                           write */
#define _BSIZE 512
                                        /* buffer size for files */
#define SECSIZE 128
                                        /* size of a sector */

/********************* setjmp and longjmp defs ************************/

        struct  _env{
                char    *rtnadr;
                char    *oldstk;
                };

        typedef struct  _env    *jmp_buf;
        typedef struct  _env    jmp_env;

/*****************************************************************/

typedef struct
        {void (*call)();
         char flag;                     /*flag register*/
         char A;
         unsigned int BC;
         unsigned int DE;
         unsigned int HL;
        } BIOSREG;

typedef struct
        {void (*call)();
         char flag;
         char A;
         char C;
         char B;
         char E;
         char D;
         unsigned int HL;
        } CALLREG;

typedef struct
        {
        unsigned int spt;
        char bsh;
```

```
        char blm;
        char exm;
        unsigned int dsm;
        unsigned int drm;
        unsigned int alv;
        unsigned int cks;
        unsigned int offs;
        char psh;
        char phm;
        char drive;
        void (*pdisc)();
        char tracks;
        char sectors;
        char firsts;
        char skew;
        char drvcha;
        char medcha;
        char gap1;
        char s1,s2,s3,s4,s5;
        } XDPB;

typedef struct
        {
        char *trans;
        int s0;
        int s1;
        int s2;
        char *dirbuf;
        XDPB *xdpb;
        unsigned int csv, alv;
        } DPH;

typedef struct {char x; void (*addr)();} *BIOS;

/**********************BIOS entry point definitions**********************/

#define BOOT    0
#define WBOOT   1
#define CONST   2
#define CONIN   3
#define CONOUT  4
#define LIST    5
#define PUNCH   6
#define READER  7
#define HOME    8
#define SELDSK  9
#define SETTRK  10
#define SETSEC  11
#define SETDMA  12
#define READ    13
#define WRITE   14
#define LISTST  15
#define SECTRAN 16
#define CONOST  17
#define AUXIST  18
#define AUXOST  19
#define DEVTBL  20
#define DEVINI  21
```

```
#define DRVTBL 22
#define MULTIO 23
#define FLUSH 24
#define MOVE 25
#define TIME 26
#define SELMEM 27
#define SETBNK 28
#define XMOVE 29

/*****************************************************************************/
```

```
(*****************************************************************)
(*                          COM TO BINARY                        *)
(* ASSEMBLES A BINARY FILE FROM A *.COM FILE                     *)
(* INTENDED FOR CONSTRUCTION OF FILES TO BE DOWN LOADED TO SYSTEM *)
(* TAKES A *.COM FILE AND STRIPS OFF THE FIRST 384 BYTES TO REMOVE THE *)
(* LOADER ADDED BY LINK-80, REPLACES THIS WITH AN APPROPRIATE HEADER *)
(* AND THEN GOES ON TO REPLACE ALL OCCURENCES OF                 *)
(* FF WITH FF FF TO ALLOW THESE TO BE DOWN LOADED TO A Z-80 SYSTEM *)
(* COMPATIBLE WITH THE LOADER INSTALLED IN ROM2                  *)
(*****************************************************************)
MODULE ComToBinary;

FROM Files IMPORT
     Close, Create, EOF, FILE, Open, ReadByte, WriteByte;

FROM InOut IMPORT
     ReadString, WriteHex, WriteString;

FROM Strings IMPORT
     CAPS;

FROM SYSTEM IMPORT
     BYTE;

FROM Terminal IMPORT
     ReadChar, WriteChar, WriteLn;


  PROCEDURE Initialise(VAR infile,outfile:FILE);
  VAR
    filename: ARRAY CHAR OF CHAR;
    finished:BOOLEAN;
BEGIN (* Initialise *)
(* OPENS THE *.COM (INPUT) AND *.BIN (OUTPUT) FILES *)
REPEAT
WriteString('What filename is the input file');
WriteLn;
WriteString('Stored under? ');
WriteLn;
ReadString(filename);
IF Open(infile,filename) THEN
  finished:=TRUE;
ELSE
  finished:=FALSE;
END;
UNTIL finished;
  REPEAT
  WriteString('What filename should the binary file');
  WriteLn;
  WriteString('be stored under?');
  WriteLn;
  ReadString(filename);
  Create(outfile,filename);
  IF Open(outfile,filename) THEN
    finished:=TRUE;
  ELSE
    finished:=FALSE;
  END;
```

```
UNTIL finished;
END Initialise;


PROCEDURE WriteHeader(outfile:FILE);

  PROCEDURE HextoByte(h1,h2:CHAR;VAR bin:BYTE);

    PROCEDURE ChartoInt(ch:CHAR):CARDINAL;
    BEGIN (* ChartoInt *)
    (* CONVERTS A HEXADECIMAL CHARACTER INTO A DECIMAL VALUE *)
    IF ((ORD('a')<=ORD(ch)) & (ORD(ch)<=ORD('f'))) THEN
      ch:=CHAR(ORD(ch)+ORD('A')-ORD('a'));
    END;
    IF ((ORD('A')<=ORD(ch)) & (ORD(ch)<=ORD('F'))) THEN
      RETURN ORD(ch)-ORD('A')+10;
    ELSE
      RETURN ORD(ch)-ORD('0');
    END;
    END ChartoInt;

  BEGIN (* HextoByte *)
  (* CONVERTS A PAIR OF HEXADECIMAL CHARACTERS INTO A DECIMAL VALUE *)
  bin:=BYTE(16*ChartoInt(h1)+ChartoInt(h2));
  END HextoByte;

VAR
  h1,h2:CHAR;
  low,high:BYTE;
BEGIN (* WriteHeader *)
(* WRITES A HEADER TO THE *.BIN FILE CONTAINING VARIOUS INFORMATION *)
(* FOR THE LOADER *)
(* Send a program offer byte *)
WriteByte(outfile,170);
(* Address to load the program into *)
WriteByte(outfile,255);
WriteByte(outfile,00);
WriteByte(outfile,00);
WriteByte(outfile,128);
(* Value to set the Stack Pointer to *)
WriteByte(outfile,255);
WriteByte(outfile,02);
WriteByte(outfile,255);
WriteByte(outfile,255);
(* Start address of the program *)
WriteByte(outfile,255);
  WriteByte(outfile,01);
  WriteString('What is the Start address in Hexadecimal?');
  WriteLn;
  ReadChar(h1);WriteChar(h1);
  ReadChar(h2);WriteChar(h2);
  HextoByte(h1,h2,high);
  ReadChar(h1);WriteChar(h1);
  ReadChar(h2);WriteChar(h2);
  HextoByte(h1,h2,low);
  WriteLn;
  WriteHex(CARDINAL(256*CARDINAL(high)+CARDINAL(low)),4);
  WriteLn;
```

```
WriteByte(outfile,low);
WriteByte(outfile,high);
(* highest address to forward *)
WriteByte(outfile,255);
WriteByte(outfile,06);
WriteString('What is the Highest address to forward?');
WriteLn;
ReadChar(h1);WriteChar(h1);
ReadChar(h2);WriteChar(h2);
HextoByte(h1,h2,high);
ReadChar(h1);WriteChar(h1);
ReadChar(h2);WriteChar(h2);
HextoByte(h1,h2,low);
WriteLn;
WriteHex(CARDINAL(256*CARDINAL(high)+CARDINAL(low)),4);
WriteLn;
WriteByte(outfile,low);
WriteByte(outfile,high);
(* Lowest address to forward *)
(* This is automatically set to 8000H since this is the bottom *)
(* of RAM *)
WriteByte(outfile,255);
WriteByte(outfile,07);
WriteByte(outfile,00);
WriteByte(outfile,128);
END WriteHeader;


PROCEDURE CopyFile(infile,outfile:FILE);
VAR
  bin:BYTE;
  c:INTEGER;
  filename:ARRAY CHAR OF CHAR;
BEGIN (* CopyFile *)
(* TRANSFERS THE *.COM FILE TO THE *.BIN FILE *)
(* WITH SUITABLE MODIFICATIONS *)
(* Strips off 'waste characters' put there by the loader *)
FOR c:=1 TO 128 DO
  ReadByte(infile,bin);
END;
(* Copies the rest of the file replacing FF by FF FF *)
REPEAT
ReadByte(infile,bin);
IF (CARDINAL(bin)=255) THEN
  WriteByte(outfile,255);
END;
  WriteByte(outfile,bin);
  UNTIL EOF(infile);
  WriteString('File Processed and Copied');
  WriteLn;
  END CopyFile;


  PROCEDURE WriteTail(outfile:FILE);
  VAR
    bin:BYTE;
  BEGIN (* WriteTail *)
  (* Exit from programming and offer program *)
```

```
  WriteByte(outfile,255);
  WriteByte(outfile,05);
  END WriteTail;


VAR
  i:CHAR;
  infile,outfile:FILE;
BEGIN (* ComToBinary *)
WriteString('*.com to *.bin conversion');WriteLn;
WriteString('produces files compatible with ROM2.MAC');WriteLn;WriteLn;
Initialise(infile,outfile);
WriteHeader(outfile);
CopyFile(infile,outfile);
WriteTail(outfile);
Close(infile);
Close(outfile);
END ComToBinary.
```

```
(*****************************************************************)
(*                          BINTOHEX                             *)
(* PROGRAM TO CONVERT A BINARY FILE INTO HEX ON THE TERMINAL SCREEN *)
(* INTENDED TO CHECK THE OUTPUT OF COM2BIN.MOD                   *)
(*****************************************************************)
MODULE BintoHex ;

FROM Files IMPORT
     Close, EOF, FILE, Open, ReadByte;

FROM InOut  IMPORT
     ReadString, WriteHex, WriteLn, WriteString;

FROM SYSTEM IMPORT
     BYTE;

VAR
  bin:BYTE;
  binfile:FILE;
  count:INTEGER;
  filename:ARRAY CHAR OF CHAR;
  i:INTEGER;

BEGIN (* BintoHex *)
WriteLn;
WriteString('What is the filename of the binary file?');
WriteLn;
ReadString(filename);
IF Open(binfile,filename) THEN
  count:=0000;
  REPEAT
  WriteHex(CARDINAL(count),4);
  FOR i:=1 TO 16 DO
    IF NOT (EOF(binfile)) THEN
      ReadByte(binfile,bin);
      WriteString('  ');
      WriteHex(CARDINAL(bin),2);
      count:=count+1;
    END;
  END;
  WriteLn;
  UNTIL EOF(binfile);
  Close(binfile);
ELSE
  WriteString('File Not Opened');
END;
END BintoHex.
```

15 Programs for the Cylindrical Homogeneous Processor

```
#include <stdio.h>

#define BIOS 1
#define ACK  255          /* Value used as acknowledgement of receipt */
#define HEAD 254          /* Head of a data or results frame */

struct j { short x; void(*addr)(); };
typedef struct j *TABLE;

TABLE bios, *p;

BIOSREG biosr;
CALLREG callr;

static int ack  = ACK;
static int sack = 00;

main()
/* Driver to supply events to the ring and to
   receive processed events from the ring     */
{
/* Arrays to hold incoming and outgoing events */
int frami[5];
int framo[5];
/* Total sent/received of each event type */
/* Not used in this version */
int sent[26];
int back[26];
int scount = 0;
/* Characters used as events */
static char sequence[10] = { 'a','g','b','c','d','e','f','i','h','j' } ;
int len;
int *pointer;
int i;
void ccall();
p = (TABLE *) BIOS, bios = (TABLE) ( ( (int) *p) - 3);
/* Determines the addresses for the status test routines */
callr.call=&bios[CONST];
ccall(&callr);
pointer=callr.HL;
pointer+=20;
/* Sets up a standard frame */
framo[0] = HEAD;
framo[1] = 01;
for (;;)
  {
  callr.call = pointer[0];
  ccall(&callr);
  if (callr.A == 0xFF)
    {
    callr.call = &bios[READER];
    ccall(&callr);
```

```
if (callr.A != HEAD)
  ack = ACK;
else if (callr.A == HEAD)
        {
        /* Take the incoming results */
        ccall(&callr);
        len = callr.A;
        for (i = 0;i < len;++i)
          {
          ccall(&callr);
          /* Print the received character */
          printf("%1c",callr.A);
          }
        /* Send an acknowledge for the data */
             callr.call = &bios[PUNCH];
             callr.C = ACK;
             ccall (&callr);
             }
    }
/* If the previous data has been acknowledged then */
/* send another one */
  if (ack)
    {
    ack = 00;
    callr.call = &bios[PUNCH];
    framo[2]=sequence[scount++];
    if (scount == 10) scount=0;
    ++sent[framo[2] - 'a'];
    for (i=0;i < 3;++i)
      {
      callr.C = framo[i];
      ccall(&callr);
      }
    }
  }
}
```

```
#include <stdio.h>

#define BIOS 1
#define ACK  255          /* Value used as acknowledgement of receipt */
#define HEAD 254          /* Head of a data or results frame */
#define UPDATES 500

struct j { short x; void(*addr)(); };
typedef struct j *TABLE;

TABLE bios, *p;

BIOSREG biosr;
CALLREG callr;
```

```
static int ack  = ACK;
static int sack = 00;

main()
/* Driver to supply events to the ring and to
   receive processed events from the ring     */
{
/* Arrays to hold incoming and outgoing events */
int frami[5];
int framo[5];
int scount = 0;
int bcount = 0;
/* Totals sent/received of each event type */
int sent[10];
int back[10];
```

```
/* The characters used as events */
static char sequence[10] = { 'a','g','b','c','d','e','f','i','h','j' } ;
int len;
int *pointer;
int i;
void ccall();
p = (TABLE *) BIOS, bios = (TABLE) ( ( (int) *p) - 3);
/* Determines the addresses for the status test routines */
callr.call=&bios[CONST];
ccall(&callr);
pointer=callr.HL;
pointer+=20;
/* Sets up a standard frame */
framo[0] = HEAD;
framo[1] = 01;
/* Initialises the sent and back arrays to zeros */
/* And prints the headings of the event types */
for (i = 0;i < 10;++i)
  {
  printf("  %1cs  %1cb",sequence[i],sequence[i]);
  sent[i] = 0;back[i] = 0;
  }
printf("\n");
for (;;)
  {
  callr.call = pointer[0];
  ccall(&callr);
  if (callr.A == 0xFF)
```

```
{
callr.call = &bios[READER];
ccall(&callr);
if (callr.A != HEAD)
  ack = ACK;
else if (callr.A == HEAD)
        {
        /* Take the incoming result */
        ccall(&callr);
        len = callr.A;
        for (i = 0;i < len;++i)
          {
          ccall(&callr);
          frami[i] = callr.A;
          }
        /* Send an acknowledgement for the data */
        callr.call = &bios[PUNCH];
        callr.C = ACK;
        ccall(&callr);
        /* Update the totals */
        ++bcount;
        ++back[frami[0] - 'A'];
        }
    }
/* If the previous data has been acknowledged then */
/* send another one */
  if (ack)
    {
    ack = 00;
    callr.call = &bios[PUNCH];
    framo[2]=sequence[scount++];
    if (scount == 10) scount=0;
    ++sent[framo[2] - 'a'];
    for (i=0;i < 3;++i)
      {
      callr.C = framo[i];
      ccall(&callr);
      }
    }
/* After UPDATES events have been received print a */
/* Summary of the totals so far */
  if (bcount>UPDATES)
    {
    bcount = 0;
    for (i = 0;i < 10;++i)
      {
      printf("%4d",sent[i]);
      printf("%4d",back[i]);
      }
    printf("\n");
    }
  }
}
```

```
#define ACK  255          /* Value used as acknowledgement of receipt */
#define HEAD 254          /* Head of a data or results frame */

static short usernproc = 3;

#include "proc1fb.h"
#include "proc2fb.h"

extern int  rtbyte();
extern int  stbyte();
extern int  rwbyte();
extern void swbyte();

/* Shared variables for handshaking */
/* Acknowledge required before any more data */
/* (or results) can be sent */
static short ack1 = ACK;
static short ack2 = ACK;
static short ack4 = ACK;
static short ack8 = ACK;

/* Send an acknowledge as soon as possible */
static short sack1 = 00;
static short sack2 = 00;
static short sack4 = 00;
static short sack8 = 00;

/* Processor ready for input */
static short pip = 01;

proc1()
/* Data Routing Process */
{
int rdata = 0;
int ndata = 0;
int len;
int i;
/* COMMS 1 BASED ALGORITHM */
for (;;)
  {
  do
    {
    if (!ndata)
      {
      test1:
      switch(ndata = rtbyte(0,1))
      {
      case ACK: ack1 = ACK;
                goto test1;
      case HEAD:break;
      default:  ndata = 00;
                break;
      }
    }

  if (!rdata)
    {
    test4:
```

```
  switch(rdata = rtbyte(0,4))
    {
    case ACK: ack4 = ACK;
              goto test4;
    case HEAD:break;
    default:  rdata = 00;
              break;
    }
  }

if (ndata && pip)
  {
  pip = 0;
  swbyte(1,1,HEAD);
  swbyte(1,1,(len = rwbyte(0,1)));
  for (i=0;i < len;++i)
    swbyte(1,1,rwbyte(0,1));
  sack1 = 01;
  ndata = 00;
  }
else if (rdata && pip)
       {
       pip = 0;
       swbyte(1,1,HEAD);
       swbyte(1,1,(len = rwbyte(0,4)));
       for (i=0;i < len;++i)
         swbyte(1,1,rwbyte(0,4));
       sack4 = 01;
       rdata = 00;
       }

if (ack2 && ndata)
  {
  ack2 = 00;
  swbyte(0,2,HEAD);
  swbyte(0,2,(len = rwbyte(0,1)));
  for (i = 0;i < len;++i)
    swbyte(0,2,rwbyte(0,1));
  sack1 = 01;
  ndata = 00;
  }
else if (ack2 && rdata)
       {
       ack2 = 00;
       swbyte(0,2,HEAD);
       swbyte(0,2,(len = rwbyte(0,4)));
       for (i = 0;i < len;++i)
         swbyte(0,2,rwbyte(0,4));
       sack4 = 01;
       rdata = 00;
       }

if (ack8 && ndata)
  {
  ack8 = 00;
  swbyte(0,8,HEAD);
  swbyte(0,8,(len = rwbyte(0,1)));
  for (i = 0;i < len;++i)
```

```
      swbyte(0,8,rwbyte(0,1));
    sack1 = 01;
    ndata = 00;
    }
else if (ack8 && rdata)
        {
        ack8 = 00;
        swbyte(0,8,HEAD);
        swbyte(0,8,(len = rwbyte(0,4)));
        for (i = 0;i < len;++i)
          swbyte(0,8,rwbyte(0,4));
        sack4 = 01;
        rdata = 00;
        }

      if (sack2) {sack2 = 00;swbyte(0,2,ACK);}
      if (sack8) {sack8 = 00;swbyte(0,8,ACK);}
      /* Doesn't want to work without this loop in */
      }
    while (ndata || rdata);
    }
}

proc2()
/* Results routing process */
{
int rresu = 0;
int nresu = 0;
int len2;
int i2;
for (;;)
  {
  do
    {
    if (!rresu)
      {
      test2:
      switch(rresu = rtbyte(0,2))
        {
        case ACK: ack2 = ACK;
                  goto test2;
        case HEAD:break;
        default:  rresu = 00;
                  break;
        }
      }

    if (!nresu)
      {
      test8:
      switch(nresu = rtbyte(0,8))
        {
        case ACK: ack8 = ACK;
                  goto test8;
        case HEAD:break;
        default:  nresu = 00;
                  break;
        }
```

```
   }

if (ack1 && (rtbyte(2,1) == HEAD))
   {
   ack1 = 00;
   swbyte(0,1,HEAD);
   swbyte(0,1,(len2 = rwbyte(2,1)));
   for (i2=0;i2 < len2;++i2)
     swbyte(0,1,rwbyte(2,1));
   }
else if (ack4 && (rtbyte(2,1) == HEAD))
        {
        ack4 = 00;
        swbyte(0,4,HEAD);
        swbyte(0,4,(len2 = rwbyte(2,1)));
        for (i2=0;i2 < len2;++i2)
          swbyte(0,4,rwbyte(2,1));
        }

if (rresu && ack1)
   {
   ack1 = 00;
   swbyte(0,1,HEAD);
   swbyte(0,1,(len2 = rwbyte(0,2)));
   for (i2=0;i2 < len2;++i2)
     swbyte(0,1,rwbyte(0,2));
   sack2 = 01;
   rresu = 00;
   }
else if (rresu && ack4)
        {
        ack4 = 00;
        swbyte(0,4,HEAD);
        swbyte(0,4,(len2 = rwbyte(0,2)));
        for (i2=0;i2 < len2;++i2)
          swbyte(0,4,rwbyte(0,2));
        sack2 = 01;
        rresu = 00;
        }

if (nresu && ack1)
   {
   ack1 = 00;
   swbyte(0,1,HEAD);
   swbyte(0,1,(len2 = rwbyte(0,8)));
   for (i2=0;i2 < len2;++i2)
     swbyte(0,1,rwbyte(0,8));
   sack8 = 01;
   nresu = 00;
   }
else if (nresu && ack4)
        {
        ack4 = 00;
        swbyte(0,4,HEAD);
        swbyte(0,4,(len2 = rwbyte(0,8)));
        for (i2=0;i2 < len2;++i2)
          swbyte(0,4,rwbyte(0,8));
        sack8 = 01;
```

```
            nresu = 00;
            }
    if (sack1) {sack1 = 00;swbyte(0,1,ACK);}
    if (sack4) {sack4 = 00;swbyte(0,4,ACK);}
    /* Doesn't want to work without this loop in */
    }
  while(rresu || nresu);
  }
}

proc3()
/* Event Processing Process */
{
int frami[5];
int framo[5];
int i3,len3;
for (;;)
  {
  /* Waits for a data packet */
  do
    len3=rwbyte(1,1);
  while(len3 != HEAD);
  len3=rwbyte(1,1);
  for(i3=0;i3 < len3;++i3)
    frami[i3]=rwbyte(1,1);
  /* Kills off a big chunk of processing time */
  for(i3=0;i3 < 1000;++i3)
    ;;
  /* Changes case of characters to show somethings been done */
  for(i3=0;i3 < len3;++i3)
    framo[i3]=frami[i3]+('A'-'a');
  pip = 01;
  /* Sends out finished results */
  swbyte(2,1,HEAD);
  swbyte(2,1,len3);
  for(i3=0;i3 < len3;++i3)
    swbyte(2,1,framo[i3]);
  }
}
```

```
#define ACK  255          /* Value used as acknowledgement of receipt */
#define HEAD 254          /* Head of a data or results frame */

static short usernproc = 3;

#include "proc1fb.h"
#include "proc2fb.h"

extern int  rtbyte();
extern int  stbyte();
extern int  rwbyte();
extern void swbyte();

/* Shared variables for handshaking */
/* Acknowledge required before more data */
/* (or results) can be sent */
static short ack1 = ACK;
static short ack2 = ACK;
static short ack4 = ACK;
static short ack8 = ACK;

/* Send an Acknowledge as soon as possible */
static short sack1 = 00;
static short sack2 = 00;
static short sack4 = 00;
static short sack8 = 00;

/* Processor ready for input */
static short pip = 01;

proc1()
/* Data Routing Process */
{
int ndata = 0;
int rdata = 0;
int len;
int i;
/* COMMS 2 BASED ALGORITHM */
for (;;)
  {
  do
    {
    if (!rdata)
      {
      test4:
      switch(rdata = rtbyte(0,4))
        {
      case ACK: ack4 = ACK;
                goto test4;
      case HEAD:break;
      default:  rdata = 00;
                break;
      }
    }

  if (!ndata)
    {
    test1:
```

```
    switch(ndata = rtbyte(0,1))
      {
      case ACK: ack1 = ACK;
                goto test1;
      case HEAD:break;
      default:  ndata = 00;
                break;
      }
    }

if (rdata && pip)
   {
   pip = 0;
   swbyte(1,1,HEAD);
   swbyte(1,1,(len = rwbyte(0,4)));
   for (i=0;i < len;++i)
     swbyte(1,1,rwbyte(0,4));
   sack4 = 01;
   rdata = 00;
   }
else if (ndata && pip)
        {
        pip = 0;
        swbyte(1,1,HEAD);
        swbyte(1,1,(len = rwbyte(0,1)));
        for (i=0;i < len;++i)
          swbyte(1,1,rwbyte(0,1));
        sack1 = 01;
        ndata = 00;
        }

if (ack2 && rdata)
   {
   ack2 = 00;
   swbyte(0,2,HEAD);
   swbyte(0,2,(len = rwbyte(0,4)));
   for (i = 0;i < len;++i)
     swbyte(0,2,rwbyte(0,4));
   sack4 = 01;
   rdata = 00;
   }
else if (ack2 && ndata)
        {
        ack2 = 00;
        swbyte(0,2,HEAD);
        swbyte(0,2,(len = rwbyte(0,1)));
        for (i = 0;i < len;++i)
          swbyte(0,2,rwbyte(0,1));
        sack1 = 01;
        ndata = 00;
        }

    if (ack8 && rdata)
      {
      ack8 = 00;
      swbyte(0,8,HEAD);
      swbyte(0,8,(len = rwbyte(0,4)));
      for (i = 0;i < len;++i)
```

```
      swbyte(0,8,rwbyte(0,4));
    sack4 = 01;
    rdata = 00;
    }
else if (ack8 && ndata)
        {
        ack8 = 00;
        swbyte(0,8,HEAD);
        swbyte(0,8,(len = rwbyte(0,1)));
        for (i = 0;i < len;++i)
          swbyte(0,8,rwbyte(0,1));
        sack1 = 01;
        ndata = 00;
        }
      if (sack2) {sack2 = 00;swbyte(0,2,ACK);}
      if (sack8) {sack8 = 00;swbyte(0,8,ACK);}
      /* Doesn't want to work without this loop in */
      }
    while (rdata || ndata);
    }
}

proc2()
/* Results routing process */
{
int nresu = 0;
int rresu = 0;
int len2;
int i2;
for (;;)
{
do
  {
  if (!rresu)
    {
    test2:
    switch(rresu = rtbyte(0,2))
      {
      case ACK: ack2 = ACK;
                goto test2;
      case HEAD:break;
      default:  rresu = 00;
                break;
      }
    }
  if (!nresu)
    {
    test8:
    switch(nresu = rtbyte(0,8))
      {
      case ACK: ack8 = ACK;
                goto test8;
      case HEAD:break;
      default:  nresu = 00;
                break;
      }
```

```
   }

if (ack4 && (rtbyte(2,1) == HEAD))
   {
   ack4 = 00;
   swbyte(0,4,HEAD);
   swbyte(0,4,(len2 = rwbyte(2,1)));
   for (i2=0;i2 < len2;++i2)
     swbyte(0,4,rwbyte(2,1));
   }
else if (ack1 && (rtbyte(2,1) == HEAD))
        {
        ack1 = 00;
        swbyte(0,1,HEAD);
        swbyte(0,1,(len2 = rwbyte(2,1)));
        for (i2=0;i2 < len2;++i2)
          swbyte(0,1,rwbyte(2,1));
        }

if (rresu && ack4)
   {
   ack4 = 00;
   swbyte(0,4,HEAD);
   swbyte(0,4,(len2 = rwbyte(0,2)));
   for (i2=0;i2 < len2;++i2)
     swbyte(0,4,rwbyte(0,2));
   sack2 = 01;
   rresu = 00;
   }
else if (rresu && ack1)
        {
        ack1 = 00;
        swbyte(0,1,HEAD);
        swbyte(0,1,(len2 = rwbyte(0,2)));
        for (i2=0;i2 < len2;++i2)
          swbyte(0,1,rwbyte(0,2));
        sack2 = 01;
        rresu = 00;
        }

if (nresu && ack4)
   {
   ack4 = 00;
   swbyte(0,4,HEAD);
   swbyte(0,4,(len2 = rwbyte(0,8)));
   for (i2=0;i2 < len2;++i2)
     swbyte(0,4,rwbyte(0,8));
   sack8 = 01;
   nresu = 00;
   }
else if (nresu && ack1)
        {
        ack1 = 00;
        swbyte(0,1,HEAD);
        swbyte(0,1,(len2 = rwbyte(0,8)));
        for (i2=0;i2 < len2;++i2)
          swbyte(0,1,rwbyte(0,8));
        sack8 = 01;
```

```
            nresu = 00;
            }
      if (sack4) {sack4 = 00;swbyte(0,4,ACK);}
      if (sack1) {sack1 = 00;swbyte(0,1,ACK);}
      /* Doesn't seem to want to work without this loop in */
      }
    while(rresu || nresu);
    }
}

proc3()
/* Event Processing Process */
{
int frami[5];
int framo[5];
int i3,len3;
for (;;)
  {
  /* Waits for a data packet */
  do
    len3=rwbyte(1,1);
  while(len3 != HEAD);
  len3=rwbyte(1,1);
  for(i3=0;i3 < len3;++i3)
    frami[i3]=rwbyte(1,1);
  /* Kills a big chunk of processing time */
  for(i3=0;i3 < 1000;++i3)
    ;;
  /* Changes case to show somethings been done */
  for(i3=0;i3 < len3;++i3)
    framo[i3]=frami[i3]+('A'-'a');
  pip = 01;
  /* Sends out finished result */
  swbyte(2,1,HEAD);
  swbyte(2,1,len3);
  for(i3=0;i3 < len3;++i3)
    swbyte(2,1,framo[i3]);
  }
}
```

```
#define ACK  255          /* Value used as acknowledgement of receipt */
#define HEAD 254          /* Head of a data or results frame */

static short usernproc = 3;

#include "proc1fb.h"
#include "proc2fb.h"

extern int  rtbyte();
extern int  stbyte();
extern int  rwbyte();
extern void swbyte();

/* Shared variables for handshaking */
/* Acknowledge required before any more data */
/* (or results) can be sent */
static short ack1 = ACK;
static short ack2 = ACK;
static short ack4 = ACK;
static short ack8 = ACK;

/* Send an acknowledge as soon as possible */
static short sack1 = 00;
static short sack2 = 00;
static short sack4 = 00;
static short sack8 = 00;

/* Processor read for input */
static short pip = 01;

proc1()
/* Data Routing Process */
{
int rdata = 0;
int ndata = 0;
int len;
int i;
/* COMMS 3 BASED ALGORITHM */
for (;;)
  {
  do
    {
    if (!ndata)
      {
      test1:
      switch(ndata = rtbyte(0,1))
      {
      case ACK: ack1 = ACK;
                goto test1;
      case HEAD:break;
      default:  ndata = 00;
                break;
      }
    }

  if (!rdata)
    {
    test4:
```

```
   switch(rdata = rtbyte(0,4))
     {
     case ACK: ack4 = ACK;
               goto test4;
     case HEAD:break;
     default:  rdata = 00;
               break;
     }
   }

if (ndata && pip)
   {
   pip = 0;
   swbyte(1,1,HEAD);
   swbyte(1,1,(len = rwbyte(0,1)));
   for (i=0;i < len;++i)
     swbyte(1,1,rwbyte(0,1));
   sack1 = 01;
   ndata = 00;
   }
else if (rdata && pip)
        {
        pip = 0;
        swbyte(1,1,HEAD);
        swbyte(1,1,(len = rwbyte(0,4)));
        for (i=0;i < len;++i)
          swbyte(1,1,rwbyte(0,4));
        sack4 = 01;
        rdata = 00;
        }

if (ack8 && ndata)
   {
   ack8 = 00;
   swbyte(0,8,HEAD);
   swbyte(0,8,(len = rwbyte(0,1)));
  for (i = 0;i < len;++i)
     swbyte(0,8,rwbyte(0,1));
   sack1 = 01;
   ndata = 00;
   }
else if (ack8 && rdata)
        {
        ack8 = 00;
        swbyte(0,8,HEAD);
        swbyte(0,8,(len = rwbyte(0,4)));
        for (i = 0;i < len;++i)
          swbyte(0,8,rwbyte(0,4));
        sack4 = 01;
        rdata = 00;
        }

    if (ack2 && ndata)
      {
      ack2 = 00;
      swbyte(0,2,HEAD);
      swbyte(0,2,(len = rwbyte(0,1)));
      for (i = 0;i < len;++i)
```

```
    swbyte(0,2,rwbyte(0,1));
  sack1 = 01;
  ndata = 00;
  }
else if (ack2 && rdata)
       {
       ack2 = 00;
       swbyte(0,2,HEAD);
       swbyte(0,2,(len = rwbyte(0,4)));
       for (i = 0;i < len;++i)
         swbyte(0,2,rwbyte(0,4));
       sack4 = 01;
       rdata = 00;
       }

    if (sack8) {sack8 = 00;swbyte(0,8,ACK);}
    if (sack2) {sack2 = 00;swbyte(0,2,ACK);}
    /* Doesn't want to work without this loop in */
    }
  while (ndata || rdata);
  }
}

proc2()
/* Results routing process */
{
int nresu = 0;
int rresu = 0;
int len2;
int i2;
for (;;)
  {
  do
    {
    if (!nresu)
      {
      test8:
      switch(nresu = rtbyte(0,8))
        {
        case ACK: ack8 = ACK;
                  goto test8;
        case HEAD:break;
        default:  nresu = 00;
                  break;
        }
      }

    if (!rresu)
      {
      test2:
      switch(rresu = rtbyte(0,2))
        {
        case ACK: ack2 = ACK;
                  goto test2;
        case HEAD:break;
        default:  rresu = 00;
                  break;
        }
```

```
   }

if (ack1 && (rtbyte(2,1) == HEAD))
   {
   ack1 = 00;
   swbyte(0,1,HEAD);
   swbyte(0,1,(len2 = rwbyte(2,1)));
   for (i2=0;i2 < len2;++i2)
     swbyte(0,1,rwbyte(2,1));
   }
else if (ack4 && (rtbyte(2,1) == HEAD))
         {
         ack4 = 00;
         swbyte(0,4,HEAD);
         swbyte(0,4,(len2 = rwbyte(2,1)));
         for (i2=0;i2 < len2;++i2)
           swbyte(0,4,rwbyte(2,1));
         }

if (nresu && ack1)
   {
   ack1 = 00;
   swbyte(0,1,HEAD);
   swbyte(0,1,(len2 = rwbyte(0,8)));
   for (i2=0;i2 < len2;++i2)
     swbyte(0,1,rwbyte(0,8));
   sack8 = 01;
   nresu = 00;
   }
else if (nresu && ack4)
         {
         ack4 = 00;
         swbyte(0,4,HEAD);
         swbyte(0,4,(len2 = rwbyte(0,8)));
         for (i2=0;i2 < len2;++i2)
           swbyte(0,4,rwbyte(0,8));
         sack8 = 01;
         nresu = 00;
         }

if (rresu && ack1)
   {
   ack1 = 00;
   swbyte(0,1,HEAD);
   swbyte(0,1,(len2 = rwbyte(0,2)));
   for (i2=0;i2 < len2;++i2)
     swbyte(0,1,rwbyte(0,2));
   sack2 = 01;
   rresu = 00;
   }
else if (rresu && ack4)
         {
         ack4 = 00;
         swbyte(0,4,HEAD);
         swbyte(0,4,(len2 = rwbyte(0,2)));
         for (i2=0;i2 < len2;++i2)
           swbyte(0,4,rwbyte(0,2));
         sack2 = 01;
```

```
            rresu = 00;
            }
    if (sack1) {sack1 = 00;swbyte(0,1,ACK);}
    if (sack4) {sack4 = 00;swbyte(0,4,ACK);}
    /* Doesn't want to work without this loop in */
    }
  while(nresu || rresu);
  }
}

proc3()
/* Event Processing Process */
{
int frami[5];
int framo[5];
int i3,len3;
for (;;)
  {
  /* Waits for a data packet */
  do
    len3=rwbyte(1,1);
  while(len3 != HEAD);
  len3=rwbyte(1,1);
  for(i3=0;i3 < len3;++i3)
    frami[i3]=rwbyte(1,1);
  /* Kills off a big chunk of processing time */
  for(i3=0;i3 < 1000;++i3)
    ;;
  /* Changes case of characters to show somethings been done */
  for(i3=0;i3 < len3;++i3)
    framo[i3]=frami[i3]+('A'-'a');
  pip = 01;
  /* Sends out finished result */
  swbyte(2,1,HEAD);
  swbyte(2,1,len3);
  for(i3=0;i3 < len3;++i3)
    swbyte(2,1,framo[i3]);
  }
}
```

```
#define ACK  255          /* Value used as acknowledgement of receipt */
#define HEAD 254          /* Head of data or results frame */

static short usernproc = 3;

#include "proc1fb.h"
#include "proc2fb.h"

extern int  rtbyte();
extern int  stbyte();
extern int  rwbyte();
extern void swbyte();

/* Shared variables for handshaking */
/* Acknowledge required before any more data */
/* (or results) can be sent */
static short ack1 = ACK;
static short ack2 = ACK;
static short ack4 = ACK;
static short ack8 = ACK;

/* Send an acknowledge as soon as possible */
static short sack1 = 00;
static short sack2 = 00;
static short sack4 = 00;
static short sack8 = 00;

/* Processor ready for input */
static short pip = 01;

proc1()
/* Data Routing Process */
{
int ndata = 0;
int rdata = 0;
int len;
int i;
/* COMMS 4 BASED ALGORITHM */
for (;;)
  {
  do
    {
    if (!rdata)
      {
      test4:
      switch(rdata = rtbyte(0,4))
      {
      case ACK: ack4 = ACK;
                goto test4;
      case HEAD:break;
      default:  rdata = 00;
                break;
      }
    }

  if (!ndata)
    {
    test1:
```

```
   switch(ndata = rtbyte(0,1))
     {
     case ACK: ack1 = ACK;
               goto test1;
     case HEAD:break;
     default:  ndata = 00;
               break;
     }
   }

if (rdata && pip)
   {
   pip = 0;
   swbyte(1,1,HEAD);
   swbyte(1,1,(len = rwbyte(0,4)));
   for (i=0;i < len;++i)
     swbyte(1,1,rwbyte(0,4));
   sack4 = 01;
   rdata = 00;
   }
else if (ndata && pip)
        {
        pip = 0;
        swbyte(1,1,HEAD);
        swbyte(1,1,(len = rwbyte(0,1)));
        for (i=0;i < len;++i)
          swbyte(1,1,rwbyte(0,1));
        sack1 = 01;
        ndata = 00;
        }

if (ack8 && rdata)
   {
   ack8 = 00;
   swbyte(0,8,HEAD);
   swbyte(0,8,(len = rwbyte(0,4)));
   for (i = 0;i < len;++i)
     swbyte(0,8,rwbyte(0,4));
   sack4 = 01;
   rdata = 00;
   }
else if (ack8 && ndata)
        {
        ack8 = 00;
        swbyte(0,8,HEAD);
        swbyte(0,8,(len = rwbyte(0,1)));
        for (i = 0;i < len;++i)
          swbyte(0,8,rwbyte(0,1));
        sack1 = 01;
        ndata = 00;
        }

   if (ack2 && rdata)
     {
     ack2 = 00;
     swbyte(0,2,HEAD);
     swbyte(0,2,(len = rwbyte(0,4)));
     for (i = 0;i < len;++i)
```

```
      swbyte(0,2,rwbyte(0,4));
    sack4 = 01;
    rdata = 00;
    }
else if (ack2 && ndata)
        {
        ack2 = 00;
        swbyte(0,2,HEAD);
        swbyte(0,2,(len = rwbyte(0,1)));
        for (i = 0;i < len;++i)
          swbyte(0,2,rwbyte(0,1));
        sack1 = 01;
        ndata = 00;
        }

      if (sack8) {sack8 = 00;swbyte(0,8,ACK);}
      if (sack2) {sack2 = 00;swbyte(0,2,ACK);}
      /* Doesn't want to work without this loop in */
      }
    while (rdata || ndata);
    }
}

proc2()
/* Results routing process */
{
int nresu = 0;
int rresu = 0;
int len2;
int i2;
for (;;)
  {
  do
    {
    if (!nresu)
      {
      test8:
      switch(nresu = rtbyte(0,8))
        {
        case ACK: ack8 = ACK;
                  goto test8;
        case HEAD:break;
        default:  nresu = 00;
                  break;
        }
      }

    if (!rresu)
      {
      test2:
      switch(rresu = rtbyte(0,2))
        {
        case ACK: ack2 = ACK;
                  goto test2;
        case HEAD:break;
        default:  rresu = 00;
                  break;
        }
```

```
  }

if (ack4 && (rtbyte(2,1) == HEAD))
  {
  ack4 = 00;
  swbyte(0,4,HEAD);
  swbyte(0,4,(len2 = rwbyte(2,1)));
  for (i2=0;i2 < len2;++i2)
    swbyte(0,4,rwbyte(2,1));
  }
else if (ack1 && (rtbyte(2,1) == HEAD))
       {
       ack1 = 00;
       swbyte(0,1,HEAD);
       swbyte(0,1,(len2 = rwbyte(2,1)));
       for (i2=0;i2 < len2;++i2)
         swbyte(0,1,rwbyte(2,1));
       }

 if (nresu && ack4)
   {
   ack4 = 00;
   swbyte(0,4,HEAD);
   swbyte(0,4,(len2 = rwbyte(0,8)));
   for (i2=0;i2 < len2;++i2)
     swbyte(0,4,rwbyte(0,8));
   sack8 = 01;
   nresu = 00;
   }
 else if (nresu && ack1)
       {
       ack1 = 00;
       swbyte(0,1,HEAD);
       swbyte(0,1,(len2 = rwbyte(0,8)));
       for (i2=0;i2 < len2;++i2)
         swbyte(0,1,rwbyte(0,8));
       sack8 = 01;
       nresu = 00;
       }

if (rresu && ack4)
  {
  ack4 = 00;
  swbyte(0,4,HEAD);
  swbyte(0,4,(len2 = rwbyte(0,2)));
  for (i2=0;i2 < len2;++i2)
    swbyte(0,4,rwbyte(0,2));
  sack2 = 01;
  rresu = 00;
  }
else if (rresu && ack1)
       {
       ack1 = 00;
       swbyte(0,1,HEAD);
       swbyte(0,1,(len2 = rwbyte(0,2)));
       for (i2=0;i2 < len2;++i2)
         swbyte(0,1,rwbyte(0,2));
       sack2 = 01;
```

```
            rresu = 00;
            }
    if (sack4) {sack4 = 00;swbyte(0,4,ACK);}
    if (sack1) {sack1 = 00;swbyte(0,1,ACK);}
    /* Doesn't want to work without this loop in */
    }
  while(nresu || rresu);
  }
}

proc3()
/* Event Processing Process */
{
int frami[5];
int framo[5];
int i3,len3;
for (;;)
  {
  /* Waits for a data packet */
  do
    len3=rwbyte(1,1);
  while(len3 != HEAD);
  len3=rwbyte(1,1);
  for(i3=0;i3 < len3;++i3)
    frami[i3]=rwbyte(1,1);
  /* Kills off a big chunk of processing time */
  for(i3=0;i3 < 1000;++i3)
    ;;
  /* Changes case of characters to show somethings been done */
  for(i3=0;i3 < len3;++i3)
    framo[i3]=frami[i3]+('A'-'a');
  pip = 01;
  /* Sends out finished result */
  swbyte(2,1,HEAD);
  swbyte(2,1,len3);
  for(i3=0;i3 < len3;++i3)
    swbyte(2,1,framo[i3]);
  }
}
```

```
#define ACK  255          /* Value used as acknowledgement of receipt */
#define HEAD 254          /* Head of a data or results frame */

static short usernproc = 3;

#include "proc1fb.h"
#include "proc2fb.h"

extern int  rtbyte();
extern int  stbyte();
extern int  rwbyte();
extern void swbyte();

/* Shared variables for handshaking */
/* Acknowledge required before any more data */
/* (or results) can be sent */
static short ack1 = ACK;
static short ack2 = ACK;
static short ack4 = ACK;
static short ack8 = ACK;

/* Send an acknowledge as soon as possible */
static short sack1 = 00;
static short sack2 = 00;
static short sack4 = 00;
static short sack8 = 00;

/* Processor read for input */
static short pip = 01;

proc1()
/* Data Routing Process */
{
int rdata = 0;
int ndata = 0;
int len;
int i;
/* COMMS 4 BASED ALGORITHM */
for (;;)
  {
  do
    {
    if (!rdata)
      {
      test4:
      switch(rdata = rtbyte(0,4))
        {
      case ACK: ack4 = ACK;
                goto test4;
      case HEAD:break;
      default:  rdata = 00;
                break;
      }
    }

  if (!ndata)
    {
    test1:
```

```
    switch(ndata = rtbyte(0,1))
      {
      case ACK: ack1 = ACK;
                goto test1;
      case HEAD:break;
      default:  ndata = 00;
                break;
      }
    }

if (rdata && pip)
  {
  pip = 0;
  swbyte(1,1,HEAD);
  swbyte(1,1,(len = rwbyte(0,4)));
  for (i=0;i < len;++i)
    swbyte(1,1,rwbyte(0,4));
  sack4 = 01;
  rdata = 00;
  }
else if (ndata && pip)
       {
       pip = 0;
       swbyte(1,1,HEAD);
       swbyte(1,1,(len = rwbyte(0,1)));
       for (i=0;i < len;++i)
         swbyte(1,1,rwbyte(0,1));
       sack1 = 01;
       ndata = 00;
       }

if (ack8 && rdata)
  {
  ack8 = 00;
  swbyte(0,8,HEAD);
  swbyte(0,8,(len = rwbyte(0,4)));
  for (i = 0;i < len;++i)
    swbyte(0,8,rwbyte(0,4));
  sack4 = 01;
  rdata = 00;
  }
else if (ack8 && ndata)
       {
       ack8 = 00;
       swbyte(0,8,HEAD);
       swbyte(0,8,(len = rwbyte(0,1)));
       for (i = 0;i < len;++i)
         swbyte(0,8,rwbyte(0,1));
       sack1 = 01;
       ndata = 00;
       }

if (ack2 && rdata)
  {
  ack2 = 00;
  swbyte(0,2,HEAD);
  swbyte(0,2,(len = rwbyte(0,4)));
  for (i = 0;i < len;++i)
```

```
      swbyte(0,2,rwbyte(0,4));
    sack4 = 01;
    rdata = 00;
    }
else if (ack2 && ndata)
        {
        ack2 = 00;
        swbyte(0,2,HEAD);
        swbyte(0,2,(len = rwbyte(0,1)));
        for (i = 0;i < len;++i)
          swbyte(0,2,rwbyte(0,1));
        sack1 = 01;
        ndata = 00;
        }

    if (sack8) {sack8 = 00;swbyte(0,8,ACK);}
    if (sack2) {sack2 = 00;swbyte(0,2,ACK);}
    /* Doesn't want to work without this loop in */
    }
  while (rdata || ndata);
  }
}

proc2()
/* Results routing process */
{
int nresu = 0;
int rresu = 0;
int len2;
int i2;
for (;;)
  {
  do
    {
    if (!nresu)
      {
      test8:
      switch(nresu = rtbyte(0,8))
        {
        case ACK: ack8 = ACK;
                  goto test8;
        case HEAD:break;
        default:  nresu = 00;
                  break;
        }
      }

    if (!rresu)
      {
      test2:
      switch(rresu = rtbyte(0,2))
        {
        case ACK: ack2 = ACK;
                  goto test2;
        case HEAD:break;
        default:  rresu = 00;
                  break;
        }
```

```
  }

if (ack4 && (rtbyte(2,1) == HEAD))
  {
  ack4 = 00;
  swbyte(0,4,HEAD);
  swbyte(0,4,(len2 = rwbyte(2,1)));
  for (i2=0;i2 < len2;++i2)
    swbyte(0,4,rwbyte(2,1));
  }
else if (ack1 && (rtbyte(2,1) == HEAD))
       {
       ack1 = 00;
       swbyte(0,1,HEAD);
       swbyte(0,1,(len2 = rwbyte(2,1)));
       for (i2=0;i2 < len2;++i2)
         swbyte(0,1,rwbyte(2,1));
       }

if (nresu && ack4)
  {
  ack4 = 00;
  swbyte(0,4,HEAD);
  swbyte(0,4,(len2 = rwbyte(0,8)));
  for (i2=0;i2 < len2;++i2)
    swbyte(0,4,rwbyte(0,8));
  sack8 = 01;
  nresu = 00;
  }
else if (nresu && ack1)
       {
       ack1 = 00;
       swbyte(0,1,HEAD);
       swbyte(0,1,(len2 = rwbyte(0,8)));
       for (i2=0;i2 < len2;++i2)
         swbyte(0,1,rwbyte(0,8));
       sack8 = 01;
       nresu = 00;
       }

if (rresu && ack4)
  {
  ack4 = 00;
  swbyte(0,4,HEAD);
  swbyte(0,4,(len2 = rwbyte(0,2)));
  for (i2=0;i2 < len2;++i2)
    swbyte(0,4,rwbyte(0,2));
  sack2 = 01;
  rresu = 00;
  }
else if (rresu && ack1)
       {
       ack1 = 00;
       swbyte(0,1,HEAD);
       swbyte(0,1,(len2 = rwbyte(0,2)));
       for (i2=0;i2 < len2;++i2)
         swbyte(0,1,rwbyte(0,2));
       sack2 = 01;
```

```
            rresu = 00;
            }
    if (sack4) {sack4 = 00;swbyte(0,4,ACK);}
    if (sack1) {sack1 = 00;swbyte(0,1,ACK);}
    /* Doesn't want to work without this loop in */
    }
  while(nresu || rresu);
  }
}

proc3()
/* Event Processing Process */
{
int frami[5];
int framo[5];
int i3,len3;
for (;;)
  {
  /* Waits for a data packet */
  do
    len3=rwbyte(1,1);
  while(len3 != HEAD);
  len3=rwbyte(1,1);
  for(i3=0;i3 < len3;++i3)
    frami[i3]=rwbyte(1,1);
  /* Kills off a big chunk of processing time */
  /* Version with weighted processing */
  for(i3=0;i3 < (100*(frami[0]-'a'));++i3)
    ;;
  /* Changes case of characters to show somethings been done */
  for(i3=0;i3 < len3;++i3)
    framo[i3]=frami[i3]+('A'-'a');
  pip = 01;
  /* Sends out finished result */
  swbyte(2,1,HEAD);
  swbyte(2,1,len3);
  for(i3=0;i3 < len3;++i3)
    swbyte(2,1,framo[i3]);
  }
}
```

16 Programs for the Distributed Depth First Search Scan

```
#include <stdio.h>

#define BIOS 1

#define VCC 'Z'      /* Visit Call Code     */
#define VRC 'X'      /* Value Request Code */
#define NN  'Y'      /* New Now             */
#define NC  'W'      /* Not Connected       */

#define MAXNODES 10 /* Maximum number of nodes in the system */

struct j  { short x; void(*addr)(); };
typedef struct j *TABLE;

TABLE bios, *p;

BIOSREG biosr;
CALLREG callr;

/* Adjacency Matrix of the graph */
static short adj[10][10];

main()
/* Depth First Search - Driver Code */
{
int now = 'a';
int f,t;
int val1,tcc,val2;
void ccall();
/* f-from t-to */
for (f = 0;f <= MAXNODES; ++f)
  for (t = 0;t <= MAXNODES; ++t)
    adj[t] = 0; /* Initialise Adjacency Matrix */
p = (TABLE *) BIOS, bios = (TABLE) ( ( (int) *p) - 3);
/* Send Visit Call Code */
callr.call = &bios[PUNCH];
callr.C = VCC;
ccall(&callr);
/* Send the value of now */
callr.call = &bios[PUNCH];
callr.C = now;
ccall(&callr);
do
  {
  /* Receive all reports ofconnection */
  callr.call = &bios[READER];
  ccall(&callr);
  /* If New Now received then break */
  if (callr.A == NN) break;
  val1 = callr.A;
  ccall(&callr);
  /* If New Now received then break */
```

```
    if (callr.A == NN) break;
    tcc = callr.A;
    ccall(&callr);
    /* If New Now received then break */
    if (callr.A == NN) break;
    val2 = callr.A;
    if (val1 != NC && val2 != NC)
      /* Update Adjacency Matrix */
      adj[val1-'a'][val2-'a'] = 10 * adj[val1-'a'][val2-'a'] + (tcc-'0');
    }
while (callr.A != NN);
/* Receive the New Now value */
ccall(&callr);
now = callr.A;
printf("\nTotal Number of connected Nodes = %3u\n\n",now-'a');
/* Print the Adjacency Matrix */
printf("\n          FROM\n    ");
for (f = 0;f < MAXNODES; ++f)
  printf(" %4u",f);
printf("\n  TO ");
for (t = 0;t < MAXNODES; ++t)
  {
  printf("\n %3u",t);
  for (f = 0;f < MAXNODES; ++f)
    printf(" %4u",adj[t]);
  }
printf("\n\nNote: Vertex 0 represents the connection point to the graph");
printf("\n      The numbers indicate which links form the connection\n");
}
```

```
#define VCC 'Z'          /* Visit Call Code     */
#define VRC 'X'          /* Value Request Code */
#define NN  'Y'          /* New Now             */
#define NC  'W'          /* Not Connected       */

static short usernproc = 1;

extern int  rtbyte();
extern int  stbyte();
extern int  rwbyte();
extern void swbyte();

static int now ;
static int val = 'a';

proc1()
/* Depth First Search - Node Code */
{
int report ;
int ti  = 01;
for (;;)
  {
  do                          /* Loop */
    {
    if ((ti = ti*2) > 8) ti = 1;
    report = rtbyte(0,ti);
    }
  while (report > 255);    /* Until a value is received */
  /* If a value Request Code then return value */
  if (report == VRC) swbyte(0,ti,val);
  /* If Value Call Code then visit answer */
  if (report == VCC) vanswer(ti);
  }
}

int valint(tv)
/* Value Interrogation */
int tv;
{
int count = 0;
int report ;
swbyte(0,tv,VRC);
do
  {
  ++count;
  if ((report = rtbyte(0,tv)) < 256) break;
    /* If a value is received then leave loop */
  }
while (count < 1024);
if (count == 1024)
  /* If timeout then return Not Connected */
  return(NC);
else
  /* If not timeout then return value of other node */
  return(report);
}

void vcall(ti,tc)
```

```
/* Visit Call */
int ti;
int tc;
{
int report;
int tr = 01 ;
/* Send the value of now to the interrogated node */
swbyte(0,tc,now);
/* Report the connection to the node being interrogated */
swbyte(0,ti,val);
swbyte(0,ti,(tc + '0'));
swbyte(0,ti,++now);
do
  {
  do                                    /* Loop */
  {
  if ((tr = tr*2) > 8) tr = 01;
  report = rtbyte(0,tr);
  }
while(report > 255);                    /* Until a value is received */
/* Leave the loop if a new now is received */
if (report == NN) break;
if (report == VRC)
  /* If a Value Request Code is received then reply */
  /* This is essential to cope with circuits in the graph */
  swbyte(0,tr,val);
else
  /* Send all reports of connection back to the */
  /* interrogating node */
  swbyte(0,ti,report);
}
while(report != NN);
/* Receive the updated value of now */
now = rwbyte(0,tc);
}

void vanswer(ti)
/* Visit Answer */
int ti;
{
int to ;
int vi ;
/* Receive the value of now */
now = rwbyte(0,ti);
vi = now;
val = ++now;
/* Report the connection to the interrogating link */
swbyte(0,ti,val);
swbyte(0,ti,(ti + '0'));
swbyte(0,ti,vi);
/* For all links except the interrogating one */
for (to = 01;to < 16;to = to*2)
  {
  if (to != ti)
    {
    /* Interrogate the link */
    vi = valint(to);
    if (vi == 'a')
```

```
        {
        /* Visit an unvisited node */
        swbyte(0,to,VCC);
        vcall(ti,to);
        }
      else
        {
        /* Report the connection to a visited node */
        /* or No - Connection */
        swbyte(0,ti,val);
        swbyte(0,ti,(to + '0'));
        swbyte(0,ti,vi);
        }
    }
}
/* Send back an updated value of now */
swbyte(0,ti,NN);
swbyte(0,ti,now);
}
```

```
#define VCC 'Z'          /* Visit Call Code     */
#define VRC 'X'          /* Value Request Code */
#define NN  'Y'          /* New Now             */
#define NC  'W'          /* Not Connected       */

static short usernproc = 3;

#include "proc1bb.h"
#include "proc2bb.h"

extern int  rtbyte();
extern int  stbyte();
extern int  rwbyte();
extern void swbyte();

static int now ;
static int val = 'a';

proc1()
/* Depth First Search - Node Code */
/* MultiProcess Version */
{
int report ;
int ti  = 01;
for (;;)
  {
  do                          /* Loop */
    {
    if ((ti = ti*2) > 8) ti = 1;
    report = rtbyte(1,ti);
    }
  while (report > 255);    /* Until a value is received */
  /* If Value Call Code then visit answer */
  if (report == VCC) vanswer(ti);
  }
}

int valint(tv)
/* Value Interrogation */
int tv;
{
int count = 0;
int report ;
swbyte(2,tv,VRC);
do
  {
  ++count;
  if ((report = rtbyte(1,tv)) < 256) break;
    /* If a value is received then leave loop */
  }
while (count < 1024);
if (count == 1024)
  /* If timeout then return Not Connected */
  return(NC);
else
  /* If not timeout then return value of other node */
  return(report);
}
```

```
void vcall(ti,tc)
/* Visit Call */
int ti;
int tc;
{
int report;
int tr = 01 ;
/* Send the value of now to the interrogated node */
swbyte(2,tc,now);
/* Report the connection to the node being interrogated */
swbyte(2,ti,val);
swbyte(2,ti,(tc + '0'));
swbyte(2,ti,++now);
do
{
do                                  /* Loop */
  {
  if ((tr = tr*2) > 8) tr = 01;
  report = rtbyte(1,tr);
  }
while(report > 255);                /* Until a value is received */
/* Leave the loop if a new now is received */
if (report == NN) break;
if (report == VRC)
  /* If a Value Request Code is received then reply */
  /* This is essential to cope with circuits in the graph */
  swbyte(2,tr,val);
else
    /* Send all reports of connection back to the */
    /* interrogating node */
    swbyte(2,ti,report);
  }
while(report != NN);
/* Receive the updated value of now */
now = rwbyte(1,tc);
}

void vanswer(ti)
/* Visit Answer */
int ti;
{
int to ;
int vi ;
/* Receive the value of now */
now = rwbyte(1,ti);
vi = now;
val = ++now;
/* Report the connection to the interrogating link */
swbyte(2,ti,val);
swbyte(2,ti,(ti + '0'));
swbyte(2,ti,vi);
/* For all links except the interrogating one */
for (to = 01;to < 16;to = to*2)
  {
  if (to != ti)
    {
    /* Interrogate the link */
```

```
vi = valint(to);
if (vi == 'a')
  {
  /* Visit an unvisited node */
  swbyte(2,to,VCC);
  vcall(ti,to);
  }
else
  {
  /* Report the connection to a visited node */
  /* or No - Connection */
  swbyte(2,ti,val);
  swbyte(2,ti,(to + '0'));
  swbyte(2,ti,vi);
  }
 }
/* Send back an updated value of now */
swbyte(2,ti,NN);
swbyte(2,ti,now);
}

proc2()
/* Mixes VRC responses with normal data traffic */
{
int t2 = 01;
int val2;
for (;;)
  {
  do                                  /* Loop */
    {
    if ((t2 = t2*2) > 8) t2 = 01;
    if ((val2 = rtbyte(2,(t2*16))) > 255)
      /* If no VRC response to send look for bytes from proc1 */
      val2 = rtbyte(2,t2);
    }
  while(val2 > 255);                  /* Until a value is received */
  /* Send the byte out */
  swbyte(0,t2,val2);
  }
}

proc3()
/* Value Request Code detection */
{
int t3 = 01;
int val3;
for (;;)
  {
  do                                  /* Loop */
    if ((t3 = t3*2) > 8) t3 = 01;
  while((val3 = rtbyte(0,t3)) > 255);
                                      /* Until a value is received */
  if (val3 == VRC)
    /* If a Value Request Code is received then reply */
    swbyte(2,(t3*16),val);
  else
    /* If not then send the byte through to proc1 */
```

```
      swbyte(1,t3,val3);
    }
  }
```