N93-15510

Proposal for Massively Parallel Data Storage System

M. Mansuripur, Optical Sciences Center, University of Arizona, Tucson, AZ 85721

ABSTRACT

An architecture for integrating large numbers of data storage units (drives) to form a distributed mass storage system is proposed. The network of interconnected units consists of nodes and links. At each node there resides a controller board, a data storage unit and, possibly, a local/remote user-terminal. The links (twisted-pair wires, coax cables, or fiber-optic channels) provide the communications backbone of the network. There is no central controller for the system as a whole; all decisions regarding allocation of resources, routing of messages and data-blocks, creation and distribution of redundant data-blocks throughout the system (for protection against possible failures), frequency of backup operations, etc., are made locally at individual nodes. The system can handle as many user-terminals as there are nodes in the network. Various users compete for resources by sending their requests to the local controller-board and receiving allocations of time and storage space. In principle, each user can have access to the entire system, and all drives can be running in parallel to service the requests of one or more users. The system is expandable up to a maximum number of nodes, determined by the number of routing-buffers built into the controller boards. Additional drives, controller-boards, user-terminals, and links can be simply plugged into an existing system in order to expand its capacity.

1. Background : The proliferation of computer networks in the near future is likely to create a tremendous demand for large data storage systems capable of handling massive amounts of information. In the foreseeable future, file-servers having capacities in the terabyte (10^{12}) range will serve a host of users whose demand for fast access and rapid data transfer rates can be satisfied only with sustained data rates of several hundred megabytes per second. At the present time, the use of high-capacity/high-data-rate storage devices may be confined to large banks and insurance companies, airline reservation systems, military and aerospace organizations, hospitals and certain medical facilities, major research and development centers, and so forth. Nonetheless, these applications represent a growing market for high-end data storage products, with performance requirements that in many instances outpace advances in the electronic storage technology.

Presently, three storage technologies dominate the mass-storage market. These are the magnetic tape, magnetic hard disk, and optical disk technologies. Tape storage is used mainly for backup and archival applications. Hard disks are predominant in high-density, fast access, high data-rate environments. Optical disks are now beginning to emerge as serious contenders for the same application area as has traditionally been assigned to magnetic disks; they also show great promise for large volume, archival applications. Roughly speaking, there is room for 1 GByte of storage on a 5.25" platter, and the achievable transfer rate to and from a disk drive (assuming reasonable disk rotation speeds), is in the range of 10 Mbit/sec (single channel). These numbers may be over-estimates for magnetic disks and under-estimates for optical disks; they are intended only as rough estimates here.

It is clear that in order to achieve high capacity and/or high data rate, small units of magnetic/optical storage must somehow be integrated. One approach to integration calls for the incorporation of multiple disks/heads within a single drive. This is certainly possible and, in fact, it is a path that has been vigorously pursued for many years. One disadvantage of this approach is the high cost and complexity of the resulting system. Also, since high performance

drives of this type concentrate large quantities of data in one location, their failure can be catastrophic. Another possible approach to integration involves networks of interconnected storage units. Here several drives are placed on a single bus (or on multiple buses), and controlled by an intelligent device driver. IBM Corporation's 3990 storage subsystem with a maximum capacity of 181 GByte (and a price tag of approximately $2,500,000) exemplifies the latter approach. Several other vendors offer integrated drives for the high-end storage market as well. Noteworthy among available products are Digital Equipment Corporation's DECarray SA900 with up to 40 GByte of storage and 2000 I/O sec, and the system DS 323 from Recognition Concepts, Inc. with 91 GByte of capacity at 72 MByte/sec data rate. These systems are typically configured around a powerful central controller, use only a few disk drives, have limited expandability, are very expensive, and are highly specialized and directed towards specific classes of applications.

We propose a massively parallel data storage system based on a network of distributed drives and controllers. The system that we envision will have the following distinct features and capabilities:

- o Constructed from small, inexpensive drives by large-scale system integration.
- o Storage capacity ranging from a few GByte to several TByte.
- o Data rate from a few MByte/sec up to several ten MBytes/sec.
- o Number of independent users from one to several hundred.
- o Expandable by adding plug-in modules.
- o Reconfigurable by changing the network of interconnections.
- o Immune to drive failures by adding redundancy to data and distributing blocks of data among several drives.
- o Designated drives for backup; automatic backup operations.
- o Automatic disk-space management and defragmentation throughout the system.
- o Open architecture; standard interface with the user.
- o Inexpensive.

In the following sections we shall describe our approach to drive integration, discuss certain important features of the distributed controller, and present preliminary results from a computer simulation program that has been used to evaluate the system's performance.

2. Introduction : The present document describes a system of interconnected data storage units (such as magnetic or optical disk/tape drives, or a mixture of them). The system allows massive amounts of data to be stored in a distributed network of storage units, with apparent data rates and access times which are substantial improvements over those available from the individual drives. Multiple users can access the system independently of each other, and will use the system by competing for resources. The architecture of the system has the geometry of a hypercube or that of an extended hypercube, depending on the required data rates which would be determined by the number of interconnections among the units. The backbone of the system is its communication network, with individual connections being made either by ordinary wires, or coaxial cables, or fiber-optic links. The control of the network is distributed, with one controller board at each node of the (extended) hypercube. Each controller manages the communication with the local user-terminal (if one is attached) and the local drive. The controller also handles the arriving message/data blocks from adjacent nodes, and determines the proper route for sending them towards their destination.

All controller boards within a network are identical in their basic design. A general-purpose board can be designed to handle a number of different protocols for communication with various types of storage devices and user terminals. Alternatively, separate boards may be built for handling specific protocols and device-interfaces. In any event, the differences among the boards for different nodes are confined to their interfaces with the local storage unit and/or the user-terminal; the routing mechanism remains the same for all boards. The boards are configured for a maximum number of adjacent-node connections. For instance, if 10 is the

maximum number of connections built into the boards, then hypercube networks designed around these boards can have as few as 2 and as many as $2^{10} = 1024$ nodes. The system is thus expandable, and can grow as the storage needs of the user increases. Details of the network's architecture and the controller board design are described in the following section.

Security against failures of individual units (head crashes, chip burnouts, etc.) may be provided by the addition of redundant blocks to the user data, and the distribution of these blocks among several drives. For instance, if five blocks of user-data are added (modulo-2) together to create a sixth block, and if these six blocks are stored on six differernt units, then failure of a single unit will not affect the integrity of the data, since the missing block can always be recovered by modulo-2 addition of the remaining five blocks. This is not a new idea; in fact the storage system architecture known as Redundant Array of Independent Drives (RAID) utilizes the same strategy for protection against drive failures. What is new here is that no specific drive is designated for storing the redundant block, and the allocation of resources involves a dynamic decision making process by the system itself. When a failure occurs, the faulty device can be replaced without halting the entire system. Subsequently the lost data is automatically reconstructed and stored on the new drive.

Another feature of the massively parallel data storage system is its ability to perform backups in the background, without requiring intervention from the users or from the system manager. Some of the nodes are designated as backup nodes, and the system is programmed to perform backups periodically, by storing the contents of all other drives at these designated nodes. Intelligence can be built into the system to perform backups with low priority, or to consider postponing such operations at times when the overall load/traffic within the system is heavy.

3. Definition of terms and description of the building blocks of the system : This section defines some of the technical terms used in the present document, and describes the building blocks of the massively parallel data storage system in detail.

Block of Data : A block of data is the minimum-size package of information that a storage system can handle. For example, a block may consist of 512 bytes of user data plus overhead (i.e., error control bits, synchronization bits, etc.). The users of the storage system submit their data in blocks, the system stores a given block contiguously on a sector somewhere within the system, and the block is returned in its entirety upon request. Error correction coding is used to ensure the integrity of the block in the presence of noise. Modulation coding may be used to tailor the block for storage on a specific device. Identification and synchronization bits may be added to the block for later identification and retrieval. All these additional bits constitute the overhead on the data. A block of user-data and its associated overhead are usually treated as a single unit of data, and recorded on a single sector. Within the storage system, each block is identified by a unique address. This address consists of two parts: The ID of the drive on which the block is stored, and the ID of the sector within that drive allocated to the block. For example, a system that stores up to 1 terabyte of data may have 1000 separate storage units (drives), each one of which handles 1 gigabyte of data. Assuming that each block is equal to one kilobyte, each drive can store up to 10^6 blocks; each sector, therefore, is identified by a 20 bit address ($2^{20} = 1,048,576$). Since each drive needs at least 10 bits for its unique identification ($2^{10} = 1024$), we see that a given block anywhere within this system is uniquely identified by a 30-bit address. A user submitting a block to the system for storage must receive this identifying address in return. All that the user needs to know about a block it submits is this address, until such time as the block is needed for processing. At that point the user will send the address to the system and request a retrieval. (Note: A data storage system dedicated to a single user can manage the task of address allocation at the user level, since the user always knows which sectors are available on various drives, where his stored data is located within the system, and so forth. On the other hand, a multi-user system that wants to keep the users independent of each other, must control the address allocation at a lower level. For example, the user in such a system must first send a request for allocation of an empty

sector. The system then sends a message, informing the user of an available address. Subsequently, the user sends its block of data for storage in that particular address, and keeps the address for future reference.)

Storage Unit (Drive) : This is a device capable of communicating via a well-defined protocol with its environment. The device must be able to accept blocks of data from a host and store that data internally. It must also be capable of delivering the stored blocks to the host upon request. The internal mechanism of storage is irrelevant as far as the outside world is concerned. The data might be stored on one or more magnetic disks, optical disks, magnetic/optical tapes, semiconductor memory chips, etc. The drive sends messages to its host, informing the host of its status. The busy message means that the drive is not available for new requests. This occurs, for instance, when the drive searches for a previously requested block of data in the read mode, or when it tries to record or erase a sector. The ready-to-transmit message means that the drive has found a previously requested block and is now ready to submit it to the host. There must be a table of contents for the drive that the host can access and modify. It is through this table that the host knows which sectors on the drive are available, what blocks are recorded on which sectors, and so forth.

Hypercube : This is a special geometry for connecting a number of devices. The simplest hypercube is a one-dimensional cube ($n = 1$), which has two nodes connected by a single edge (see Fig. 1(*a*)). The next hypercube has dimension $n = 2$, consists of $2^n = 4$ nodes, and each node is connected to two neighboring nodes via two edges, as shown in Fig. 1(*b*). The three dimensional cube is the ordinary cube with $n = 3$, has $2^n = 8$ nodes, and each node is connected to three other nodes via three edges. To construct an $n + 1$ dimensional cube, therefore, one must connect the corresponding nodes of two n-dimensional cubes via 2^n new edges. For example, Fig. 1(*d*) shows how a 4-dimensional cube may be constructed by connecting the eight nodes of two ordinary (i.e., 3*d*) cubes. In this way it is easy, for instance, to see that a 10-*d* cube consists of $2^{10} = 1024$ nodes, each node is connected directly to 10 neighboring nodes via 10 edges, and, in order to go from any node to any other node of the hypercube, one needs to go over at most 10 edges. In the architecture set forth in this document we shall connect data storage devices in the hypercube geometry (or an extension of the hypercube geometry to be described in the next paragraph). Thus individual drives are at the nodes of the hypercube, and the backbone of the communication network that connects these drives is formed by wires (i.e., twisted pairs, coax cables, fiber-optic links, etc.) that can be imagined as the edges of the hypercube. Of course we do not have access to a physical higher dimensional space than $n = 3$, so that the angles at the corners of our hypercubes are not 90^0 angles, but this is not a matter of concern, since all we are interested in is the number of connections between the nodes and the order in which these connections are made.

Extended Hypercube : In certain applications it might be desirable to increase the number of connections at each node, i.e., to have more adjacent nodes (as compared to the hypercube) for each node. In such cases we shall use simple extensions of the hypercube. In principle, an arbitrarily large number of connections per node can be achieved, provided that one is willing to use the required number of wires. In Fig. 2 we have depicted, by way of example, the geometry of an extended hypercube. Upon close examination of the strategy employed in Fig. 2 and comparison with Fig. 1, it should be trivial to create further extensions of the hypercube containing even more connections per node.

In Fig. 2(*a*) we have the smallest unit of an architecture with 4 nodes and 3 connections to each node. In Fig. 2(*b*) the basic unit is replicated 4 times, and each node is connected to 3 similar nodes. We now have a total of 16 nodes with each node connected directly to 6 other nodes. (Remember that the hypercube of dimension 4 which also has 16 nodes, has only 4 connections per node.) To go to the next step we can take the structure in Fig. 2(*b*), replicate it four times, and connect each one of the resulting nodes to their three counterparts. We will then have a total of 64 nodes, with each node directly connected to 9 other nodes (that is three connections more than afforded by a 6-*d* hypercube which has the same number of nodes). In

like manner, the procedure can be extended to structures with 4^n nodes and $3n$ connections at each node.

Link : Each edge of the hypercube is a "link" in the actual system. A link is *a* bi-directional communication channel between two adjacent nodes of the network. Links may be simple twisted-pair wires, coaxial cables, or fiber-optic channels. An *n*-dimensional hypercube will have a total of $n2^{n-1}$ links. A message/data block created at one node and addressed to another node, must travel through one or more links before it arrives at its destination. Typically, a message/data block arrives at a node, waits for a link to an appropriate adjacent node to become available, and then travels to that node; the process continues until the block arrives at its destination. Assuming that traffic jams do not force the controller to re-route, each message/data block traverses at most *n* links in an *n*-dimensional hypercube. For instance, in a 10-*d* hypercube architecture which has 1024 nodes, travelling blocks need at most 10 hops to reach their destination. Most blocks, however, travel a shorter distance than the above maximum, since they do not travel between extreme opposite corners of the cube. Since an extended hypercube will have more links than an ordinary hypercube with the same number of nodes, communication within the extended cube is faster and more efficient.

Buffer : There are several buffers on each controller board. A buffer is an electronic storage device, such as a shift register, used as temporary storage for commands and data blocks between origination and destination points. A buffer can store one block of user data plus additional information such as a command, one or more command qualifiers, flags, and command-related addresses. Figure 3 shows a typical structure of one such buffer. Within a board, buffers can exchange their contents with one another, transfer the contents to the local storage device, send/receive command and data from the local user-terminal, and transfer contents to another controller board (located at an adjacent node) via the links. The buffers do not make these decisions themselves; the control logic reads the command section of each buffer and initiates the necessary tranfer(s). The buffers may have multiple layers of depth, as shown in Fig. 4. When traffic congestions occur on the board, or when the buffer needs to wait for unloading its contents (and this may happen for a variety of reasons), the control logic pushes the contents one level down and clears the way for the next transaction. All previous commands in this case wait "below the surface" until congestion problems are resolved.

Controller Board : At each node of the (extended) hypercube there resides a controller board. A block diagram illustrating a possible configuration of this board is shown in Fig. 5. A storage unit (drive) communicates with the controller through the drive buffer (*d*-buffer for short). Similarly, the *I/O*-buffer handles traffic in and out of the user-terminal. The *d*-buffer is used to store command/data on the way to the storage device, or on the way out of the storage device and to a specific node. It may happen that a given node is not selected as a user-terminal node, in which case the flags within the *I/O*-buffer are properly set to indicate this fact to the control logic; the control logic then ignores the *I/O* buffer.

Routing buffers (*r*-buffers) receive command/data items from adjacent nodes via the links, or directly from other buffers within the same node. The number of *r*-buffers on each board is equal to the number of adjacent nodes for the particular architecture under consideration. For example, an *n*-cube architecture requires *n* routing buffers per board. When the contents of an *r*-buffer are addressed to the current node, the controller extracts them and places them either in the *d*-buffer or in the *I/O*-buffer. When the contents of an *r*-buffer are addressed to another node, the controller searches for the best route to direct the item, then transfers it to the proper adjacent node. The routing decisions are made locally at each node, based on the destination address, availability of adjacent-node buffers, and whatever local information may be available to each controller. The cross-bar switch allows any buffer to be connected to any link, connects *r*-buffers to *d*-buffer and *I/O*-buffer, and allows *d*-buffer and *I/O*-buffer to exchange their contents. The state of the cross-bar switch is determined by the control logic which is the CPU of the controller board. The control logic reads the command segment of each buffer, identifies the best route for the contents of that buffer, and programs the cross-

bar switch for the necessary connections.

The diagram in Fig. 5 also shows a connection between the control logic and a unit called the status table. This status table keeps useful information about the state of other nodes within the system. For instance, a drive upon becoming engaged/released by a user-terminal can send a message to all nodes, informing them of its new status. Each node then updates the status of that particular drive and keep the information in the status table, thus allowing the control logic to draw on this information for making allocation decisions.

4. Advantages : The massively parallel data storage system described in this document has several advantages over the existing mass storage devices. Some of the advantages of this system are briefly described in the following paragraphs.

i) The system is built from small units (drives) that are produced in large quantity for the personal computer market. These units are cheap, reliable, and are available from a number of different manufacturers. It is therefore possible to reduce the overall cost of the system and maintain the flexibility to respond to new technological developments.

ii) The system can be expanded by adding new storage units/links to the old ones. Thus when one's need for storage capacity/transfer rate increases, one can upgrade an existing system and pay only the cost of additional units. In the case of hypercube architecture, for example, if the controller boards were designed to handle 10 adjacent nodes, then the upper limit to the number of units would be 1024, if the boards were designed for 11 adjacent nodes the upper limit would be 2048, and so on. One might also choose to reconfigure a system without adding new drives, but by changing the architecture to an extended hypercube which has more links per node. For example, a 4-*d* hypercube with 16 drives and 4 links per node (total number of links = 32) can be reconfigured to the architecture shown in Fig. 2(*b*) which has the same number of drives but, with 6 links per node, requires 16 additional links. The new configuration will now have an increased data rate, since its nodes have more connections to the rest of the system.

iii) The massively parallel data storage system can handle as many independent user-terminals as there are nodes in the system. Each user has access to the entire system, and interacts with the system as though there were no other users. Of course the presence of many users on the network will affect the response-time and the traffic load, but it does not modify the logical modes of interaction between the user and the system.

iv) By adding redundancy to the data and distributing the blocks of data among various drives, the performance of the massively parallel data storage system becomes immune to failures. Thus when a storage unit fails, the system proceeds to reconstruct the lost data from the remaining blocks. In the meantime, the failed device may be replaced without having to shut down the entire system. Once the new drive is up and running, the system automatically replaces the lost data.

v) The system performs automatic backups during periods in which the network's load is light. One or more nodes may be assigned to backup drives (such as tape drives), and the system instructed to transfer the contents of other units to the backup units during appropriate time intervals. No action on the part of the user(s) is therefore required for safe-keeping of the data.

5. Computer Simulation : *A* highly simplified version of the hypercube-configured, massively parallel data storage system was simulated (in FORTRAN) on a VAX-Station. The purpose of the simulation was to demonstrate the feasibility of the concept, and also to investigate the performance of various routing and communication algorithms. Since the

simulated system incorporates some of the essential features of the proposed system, this section is devoted to a brief description of certain features of the simulation. The following terminology has been used in developing the simulation program.

ID: Number of dimensions of the hypercube (ID_{max} = 16)
NODES: Number of nodes in the network (NODES = 2^{ID})
CNODE: Current node address (16-bit format)
DNODE: Destination node address (16-bit format)
GNODE: Origination node address (16-bit format)
SFLG: Status-flag, a logical array with as many elements as there are nodes. Each node keeps a copy of *SFLG* which is continually updated throughout the system to reflect the most recent state of the drives. *SFLG*(n) = true/false depending on whether or not the drive at node n is being used by some other node.

Input/Output Buffer : Within each I/O-buffer two flags and six segments are utilized for command, DNODE/GNODE address, begin/end addresses for data-blocks, and the data-block itself.
IOFLG1: When this flag is false, buffer needs attention from the user-terminal.
IOFLG2: When this flag is false, buffer needs attention from the controller
IOFLG3: (not used)
IOBFR1: DNODE or GNODE address, depending on the type of command
IOBFR2: (not used)
IOBFR3: (not used)
IOBFR4: Command
IOBFR5: Command-parameter
IOBFR6: Begin address of the block
IOBFR7: End address of the block
IOBFR8: One data-block sent either from user-terminal for storage, or from a drive for pickup by the terminal

Drive Buffer : Within each d-buffer three flags and five segments are utilized for command, DNODE/GNODE address, begin/end addresses for data-blocks, and the data-block itself.
DFLG1: When this flag is false, buffer needs attention from the drive.
DFLG2: When this flag is false, buffer needs attention from the controller.
DFLG3: This flag becomes false upon reservation; it becomes true again when released by the node that made the reservation. When reserved, only commands arriving in r-buffers with the correct DNODE address are accepted. Whenever the status of DFLG3 changes, the corresponding SFLAG is reset throughout the system.
DBFR1: DNODE or GNODE address, depending on the type of command
DBFR2: (not used)
DBFR3: (not used)
DBFR4: Command
DBFR5: (not used)
DBFR6: Begin address of data-block
DBFR7: End address of data-block
DBFR8: Data-block

Routing Buffers : Each node has a number of r-buffers equal to ID, one for each link attached to the node. Each buffer can accept one command/data-item from its input link, or from a local r-buffer, d-buffer, I/O-buffer, or x-buffer. The contents of a buffer may be transferred to another node via the corresponding link and r-buffer, or they may be shipped to the local d-buffer or to the local I/O-buffer. The r-buffers utilize one flag and eight segments, as follows.
RFLG1: When this flag is true, r-buffer is empty in which case it can receive a command/data-item. When false, the buffer must be attended to by the control logic.

RFLG2: (not used)
RFLG3: (not used)
RBFR1: DNODE address for the command (16-bit binary format)
RBFR2: GNODE address for the command (16-bit binary format)
RBFR3: Counter. If 0, command does not propagate, otherwise, it is shipped to the adjacent *r*-buffer, with the corresponding bit of its DNODE address toggled and its counter decremented by 1.
RBFR4: Command
RBFR5: Command-parameter
RBFR6: Begin address of data-block
RBFR7: End-address of data-block
RBFR8: Data-block

Auxiliary Buffers : Each buffer in the controller-board has its own x-buffer corresponding to depth level #1 (see Fig. 4). When traffic congestions make it impossible for a buffer to ship its contents to the proper address, the contents are automatically pushed to the *x*-buffer immediately below for temporary storage. The *x*-buffer pops back up whenever its parent buffer becomes available again.
XFLG1: When this flag is false, *x*-buffer is occupied in which case it needs attention from the control logic.
XFLG2: (not used)
XFLG3: (not used)

Cache : The cache memory residing in each controller-board receives data-blocks from local user-terminal and stores them at the address indicated by BPNTR. Blocks move out sequentially from the address indicated by TPNTR. Cache is empty when BPNTR = TPNTR.
NMAX: Maximum number of blocks cache can store.
TPNTR: Pointer to the top of the occupied block within the cache.
BPNTR: Pointer to the bottom of the occupied block within the cache.
CFLG: When this flag is false, cache is full.

Internal system commands : The following commands were used in the computer simulation. This is a minimum command set required for basic routing and data transfer procedures.
F. Set at each and every node the status-flag of the drive at CNODE. SFLG(n) indicates whether or not the drive at node *n* accepts read/write/erase requests (when accepting, SFLG(n) = True).
H. Hold drive at DNODE on behalf of GNODE.
P. Reporting the status of drive in response to *a* H-request. Informs GNODE whether or not *d*-buffer at DNODE is available. If available, it also delivers an assigned address for the data-block.
L. Release drive at DNODE from an earlier hold made by GNODE.
S. Deliver data-block for storage at DNODE.
E. Erase data-blocks (address = begin : end) from drive at DNODE.
R. Retrieve data-blocks (address = begin : end) from drive at DNODE.
O. Transfer data-block to *I/O*-buffer at DNODE.

Results of simulations : Simulations were performed for hypercube networks of disk drives with varying dimensions. In each case, the number of users was varied in order to study the effect of overall load on the level of interaction between the system and individual users. Users were given equal priority, and were allowed to issue random requests for READ and WRITE operations. The WRITE commands were preceded by requests for drive allocation, without specifying the drive. The controller then tried to assign the local drive to the task, except when that drive was busy, in which case it selected another (free) drive. Subsequently the data-block was shipped to the reserved drive for storage. In case of a READ command, the user-terminal first made a request for reservation from the drive that contained the desired

block of data. If the request was granted, the drive would be asked to read a 5-sector block and forward it to the terminal that had issued the request. When a request for reservation was denied, the user-terminal simply abandoned the request. The simulation program was incapable of admitting concurrency of operation among the various nodes, leaving us with no option but to scan the nodes sequentially. The results obtained by simulation, therefore, are worst-case scenarios and do not reflect the true level of parallelism inherent in the system. Despite this and several other shortcomings of the simulation program, the results are impressive and indicate that significant gains can be expected from a massively parallel data storage system.

The unit of time T in these simulations is taken as the time needed to transfer the contents of one buffer over a single link (i.e., transfer between adjacent nodes). Each disk operation (sector-read or sector-write) is assumed to take 20T. Table I lists the total number of completed read/write operations in several simulations that were performed over a time interval of 500T. Notice that a single drive, working incessantly to perform sector read/write jobs, would complete only 25 such operations during the same period of time. There was no cache memory on the simulated boards, and the buffers had only one level of depth. When traffic jams occurred, the program was aborted and the simulation repeated with a reduced frequency of read/write requests on behalf of each user.

The simulation results in Table I indicate a significant level of drive activity within the system. For instance, the hypercube storage device with 16 drives and 4 users performs 371 read/write operations during the period under consideration, i.e., the equivalent of 15 full-time drives. Or consider the system with 64 nodes and 12 users, which delivers the performance of 30 full-time drives. Even ignoring the fact that the simulation under-estimates the power of the system, and also the fact that the users in these simulations make frequent requests for fairly short operations, the performance figures are still very impressive.

Table I. Simulated performance results for hypercube mass-storage systems. When more than one user is present, the users are given equal priority, and allowed to make random requests for READ and WRITE operations from the system. No attempt has been made to optimize the performance or to break traffic jams. The results shown are simply examples of sustained operation over a long time interval. (For comparison, note that a single drive running incessantly during the same period would perform 25 such operations.)

Nodes	Users	Requests issued/granted	Operations read/write
16	1	144/099	255/48
16	4	217/115	320/51
64	1	149/116	270/62
64	7	255/213	505/112
64	12	290/236	630/110
128	1	118/110	250/60
128	10	374/272	730/126

Figure Captions

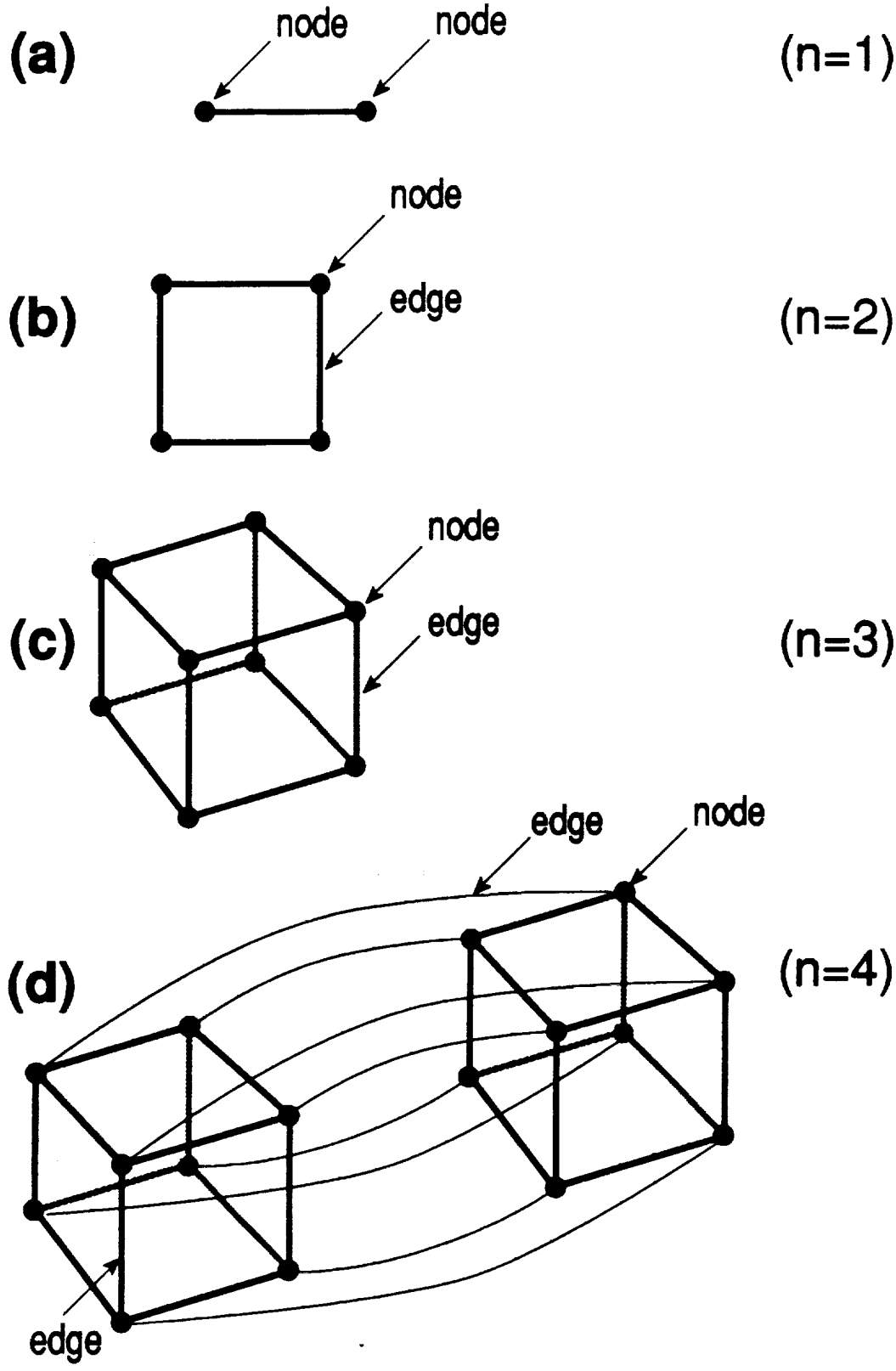
Fig. 1 shows n-dimensional cubes with n = 1, 2, 3 and 4. A cube of dimension $n > 3$ is often referred to as a hypercube. An n-dimensional cube will have 2^n nodes and $n2^{n-1}$ edges.

Fig. 2 shows the principle of construction of an extended hypercube network. The lowest-dimensional structure of this type shown in (a) has 4 nodes, where each node is directly connected to 3 other nodes. The next higher-dimensional structure shown in (b) consists of 16 nodes, where each node is directly connected to 6 other nodes.

Fig. 3 is the block diagram of a typical buffer register. Several such registers are used as temporary storage units within each controller board. A buffer register is essentially a shift register which is loaded and unloaded electronically, either serially, or in parallel, or in a mixed mode. The control logic reads the "command & control" section of the register to decide what task(s) to perform and/or which route to send the block along. The "data" section of the register contains one block of user-data (typically 512 or 1024 bytes) which is on its way to be stored on a drive or to be retrieved by a user-terminal.

Fig. 4 shows a buffer register with multiple depth layers. At times of congestion, contents of the entire register are pushed one level down in order to free the main register (top) for continued operation. When the congestion condition disappears, contents of the register are pushed up (layer by layer) until all levels are cleared up.

Fig. 5 shows a block diagram of the controller board whose responsibility is the control of operations at each node of the system. To each node is attached a storage unit and, possibly, a user-terminal, both communicating with their respective buffers. The n routing buffers are associated with n adjacent nodes of the system. The control logic has access to the "command & control" section of each buffer, and makes all the decisions regarding transfer of data both within and without the board. The bidirectional cross-bar switch, which is capable of connecting any buffer to any other buffer or to any link, receives its commands from the control logic. The status table keeps track of the state of various nodes in the system; its contents are updated frequently by status messages that circulate through the network.



Illustraiton by Vamer
Mansuripur / Opt sci fig 1

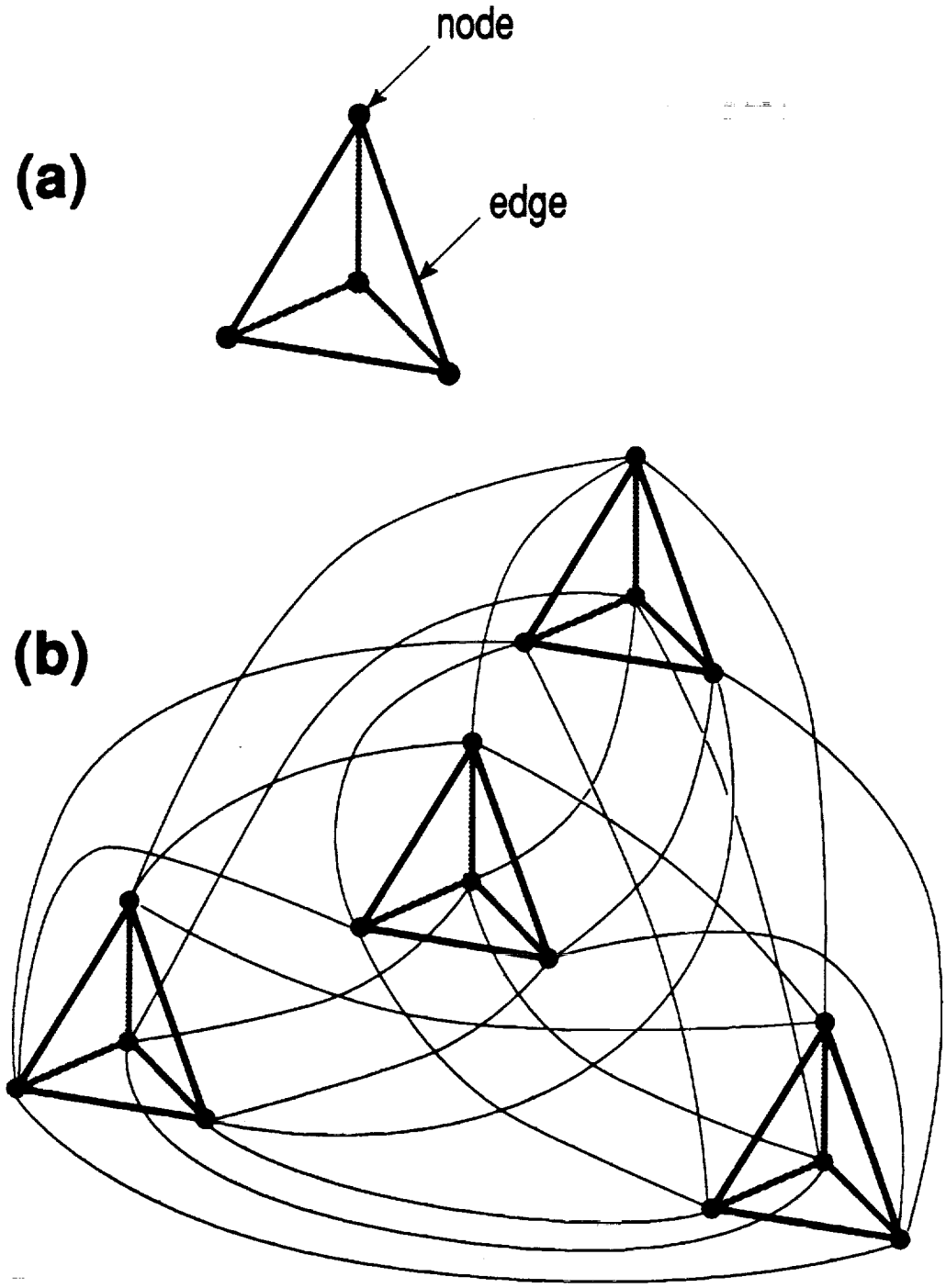
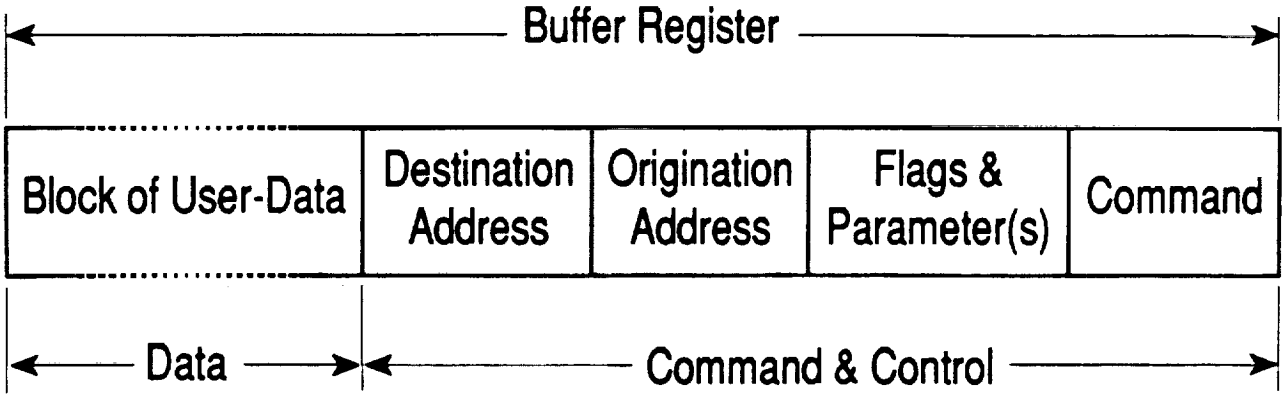
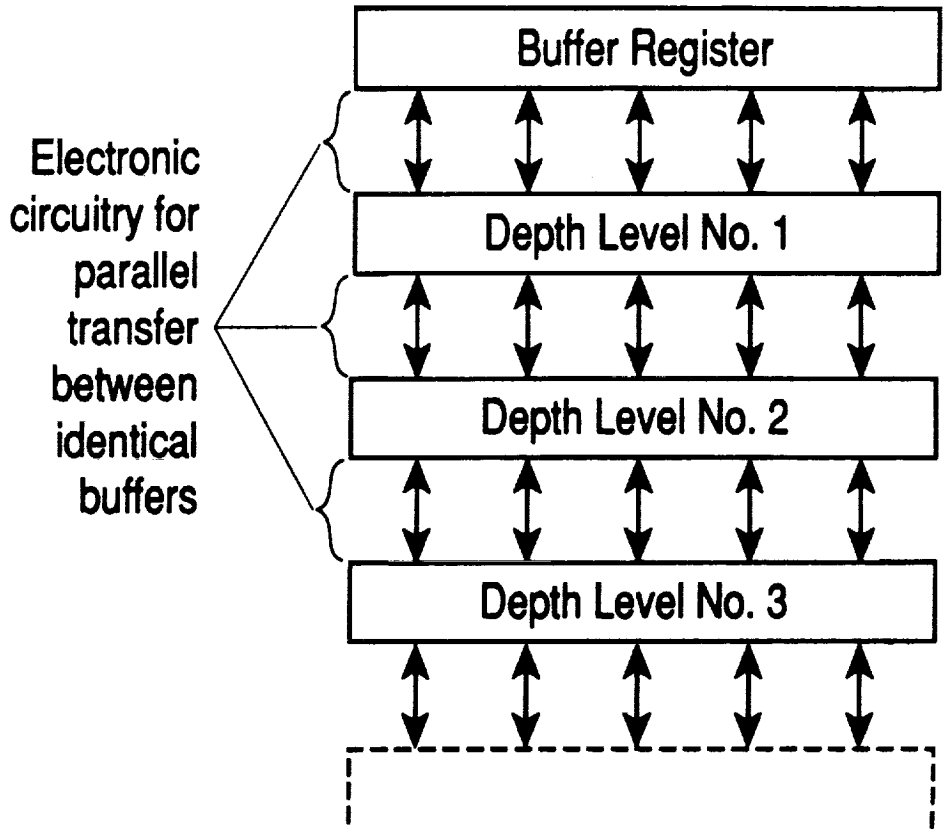


Illustration by Varner
Mansuripur / Opt sci fig 2

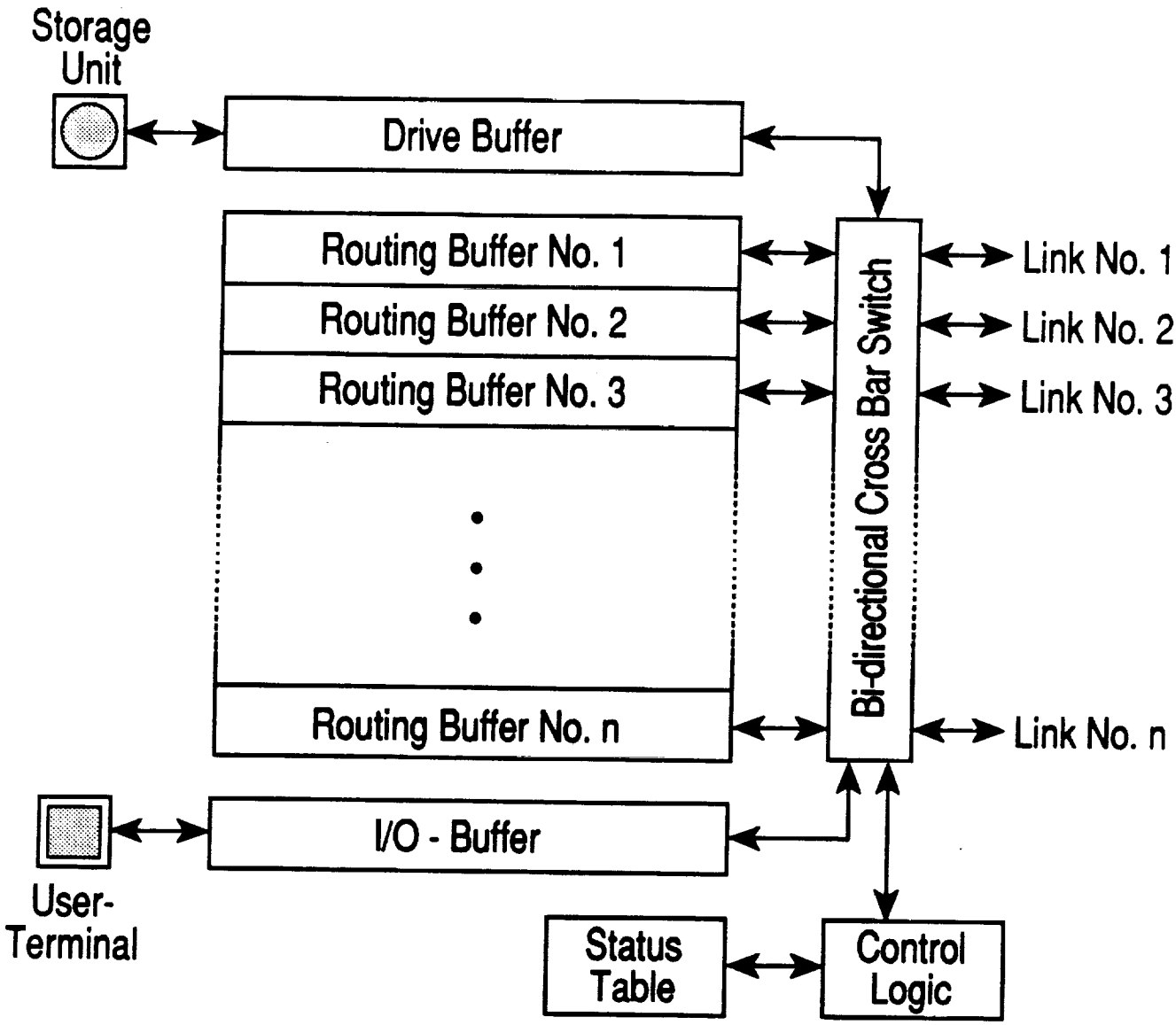
C-2



Illustraiton by Vamer Mansuripur / Opt sci fig 3



Illustraiton by Varner Mansuripur / Opt sci fig 4



Illustraiton by Varner Mansuripur / Opt sci fig 5

APPENDIX F