

User-Level Parallel File I/O

Y. Wang and E. L. Leiss*
 Department of Computer Science
 University of Houston
 coscel@cs.uh.edu

Abstract

Parallel disk I/O subsystems are becoming more important in today's large-scale parallel machines. Parallel disk systems provide a significant boost in I/O performance reducing the gap between processor and disk speeds. We describe a Unix-like file I/O user interface, implemented in a parallel file I/O subsystem on an MIMD machine, the nCUBE 2. Based on message passing, we develop parallel disk read/write algorithms to achieve higher parallelism when more disk drives are used. We use the closed queuing network model to analyze the effect of some tunable system parameters for our parallel file system. We then performed simulation experiments in order to obtain more realistic performance data, by comparing the original vendor-supplied file system with ours. The results indicate that the speedup in I/O performance is almost equal to the number of disk drives we use. Thus, our user-level parallel file I/O approach will provide scalable I/O performance.

1. Introduction

High computation rates may mean little if an application has large file I/O requirements, involving many megabytes of input/output data. In many multiprocessors, file I/O is handled by a host (front-end) computer attached to the massively parallel machine. Thus, hundreds of processors share a few channels to communicate with the host for the use of disks. In addition, historic trends clearly indicate that the improvement in the speed of disks is not keeping up with the increase in the speed of processors [Leiss&Johnson, 1988] [Leiss, 1995]. Consequently the computing capability and I/O performance are increasingly unbalanced in these massively parallel computing systems. We describe a Unix-based parallel file I/O system and discuss our experiences with its implementation on an nCUBE 2 hypercube with 1024 nodes. While this system is rather old (for a high-performance computing system), recent congestion studies using Sandia's Intel Paragon (a 2D mesh topology) under two different high-performance operating systems (SUNMOS and Puma) suggest that the results reported here are substantially valid for newer generations of MIMD systems as well [Sohn&Leiss, 1996] [Zhao, 1997].

The nCUBE 2 has an MIMD hypercube architecture. By using VLSI technology to integrate all the components necessary for parallel computing onto a single chip, the nCUBE 2 system may

* Work supported by Sandia National Laboratories; access to Sandia's nCUBE 2 is acknowledged. This is an expanded and up-dated version of a paper that was presented at PANEL '96 [Wang&Leiss, 1996].

scale from as few as 8 to as many as 8,192 compute processors in a single system architecture [NCUBE, 1992].

- *Processor:* Each nCUBE 2 node is a VLSI CISC (Complex Instruction Set Computer) 64-bit-register processor which is rated at approximately 2.7 double-precision MFLOPS and 7 MIPS with a 20 MHz clock.
- *Memory:* The nCUBE 2 is a distributed-memory supercomputer with up to 16 Mbytes memory for each processor.
- *Communication:* A network communication unit includes 14 DMAs (Direct Memory Access), by which Input/Output channels can access main memory without the intervention of the CPU. These are used to transfer data between main memory and I/O channels. Of the 14 DMAs, one is reserved for I/O devices such as disks to provide full support for system designs with up to a dimension 13 hypercube. The DMA channels are full duplex and parity checked for reliability with a 2.2 Mbytes/second network bandwidth.
- *I/O:* The NCUBE NChannel boards [NCUBE, 1988] provide the connections between the hypercube and the external world. Each NChannel board has sixteen serial channels connecting to the peripheral devices. Each serial channel on the NChannel board is controlled by an I/O processor - an identical processor to those used in the hypercube. Each serial channel can support one I/O peripheral controller.

The nCUBE 2 system makes it easy for users to allocate system resources. Programs run on *subcubes* (a subset of the total compute processors in the hypercube that forms itself a hypercube), with sizes varying from a single compute processor to the entire system configuration. To address the processors in a subcube, a program uses the processors' logical numbers within the subcube, rather than actual hardware addresses. This means a program can function consistently no matter where in the hypercube it is loaded. Within a subcube, processors are numbered starting at 0. For example, a 2-dimensional subcube has four processors which are numbered 0, 1, 2, and 3. Users can select the size of the subcube at compile-time or at run-time without modifying their programs.

Each I/O board has 128 full duplex channels (see Figure 1) directly connected to the hypercube [NCUBE, 1988]. Each I/O board contains 16 I/O processors, each connected to eight compute nodes in the hypercube by using 8 bidirectional channels. In contrast to the eight channels used by an I/O node on the NChannel board, the iPSC/2 CIO system [Intel, 1988] has only one Direct-Connect link from one I/O node to the hypercube. The architecture used in the nCUBE 2 permits high I/O data transfer rates between the hypercube itself and the external world.

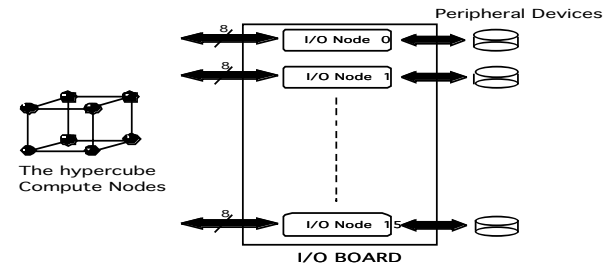


Figure 1: An NChannel I/O Board

To understand the multiple-channel configuration, consider the following example. We use 8 I/O nodes on an I/O board with logical number 3. These 8 I/O nodes are shared by 1024 compute nodes in the hypercube through 64 distinct I/O SI channels. For example, the I/O node 2 uses its channel 0 to connect to the compute node 11, channel 1 to the compute node 75, channel 2 to the compute node 139, and so on.

Table 1: Multiple I/O Channels to the Hypercube

	I/O Node 0	I/O Node 1	I/O Node 2	I/O Node 3	I/O Node 4	I/O Node 5	I/O Node 6	I/O Node 7
Channel 0	3	7	11	15	19	23	27	31
Channel 1	67	71	75	79	83	87	91	95
Channel 2	131	135	139	143	147	151	155	159
Channel 3	195	199	203	207	211	215	219	223
Channel 4	259	263	267	271	275	279	283	287
Channel 5	323	327	331	335	339	343	347	351
Channel 6	387	391	395	399	403	407	411	415
Channel 7	451	455	459	463	467	471	475	479

An SI node is a compute node directly connected to its corresponding I/O node via the SI channel. All other nodes (non-SI nodes) route their I/O messages to their nearest target SI node, which then routes further to the corresponding I/O node. The path from the non-SI node to SI node follows the e-routing scheme [NCUBE, 1992]. The nCUBE 2 provides 8 different channels to increase the bandwidth. Nevertheless, the disadvantage is that several compute nodes share the same SI node.

2. Parallel File I/O Mechanism

The disk driver and file system interface “*nsdisk*” runs on each I/O node and manages files. It can be viewed as a separate file system process that controls attached disks. These *nsdisks* receive messages from compute nodes and perform I/O jobs. *nsdisk* is a UNIX-like file system which uses file descriptor tables to keep the information of open files for each process. On the compute node side, the process only keeps a file table structure containing the address of the I/O process (e.g., *nsdisk*) servicing this file and the file descriptor used by this I/O process. A separate file in this I/O process has its own file descriptor. A compute process can be viewed as a client process and an I/O process as a server process. We may create separate file descriptors in each I/O node to access several files simultaneously. A compute process can refer to these different files as a logically single, large file to achieve parallel I/O.

Figure 2 illustrates the file structure, with process A opening two files, one in I/O node 0, the other in I/O node 1. Process B opens a file in the I/O node 0. The UFID (User File ID) is used as a file descriptor in the compute node, and the SFID (System File ID) is for the I/O node. The

Vertex operating system, the vendor-supplied OS for the nCUBE 2, allocates two UFIDs (0 and 2) for process A, and receives SFID 0 and 1 from I/O node 0 and I/O node 1, respectively. Process B has UFID 1 and SFID 2 in the I/O node 0. The entry in the process file table contains only the information of a pointer to the system file table in the I/O node. The process file table resides in the compute node.

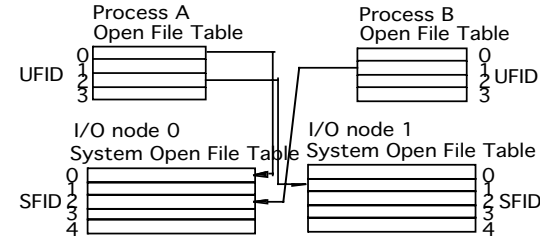


Figure 2: File Structure Used in the nCUBE 2

The problem of multiple disks on separate computing entities has been studied for distributed file systems [Reddy&Banerjee, 1989]. NFS (Network File System) from Sun is an example of such a file system [Lefler *et al.*, 1989]. Note that any one file is constrained to reside on a single disk. This clearly causes a bottleneck at the I/O node if data are required concurrently by several compute nodes. The vendor-supplied nCUBE 2 file system is based on NFS. Each I/O node uses the file server *nsdisk* process to manage its independent file system. The nCUBE 2 uses the disk volume name as a prefix of the path name to open a file, e.g., “//df00/...”. A disk’s volume name is unique to the whole nCUBE 2 file system. Therefore, the user process can decide to which I/O node the I/O message should go. Although a file is constrained to reside on a single disk, we still can use the data declustering technique [Reddy&Banerjee, 1989] to open several files on each disk to make the data distribute to these different files.

I/O in the nCUBE 2 file system is based on message-passing. It includes the asynchronous and synchronous services. The nCUBE 2 library provides *read* and *write* function calls (the same as the UNIX read and write function calls) to achieve the synchronous block I/O operations. The asynchronous function calls are *readstart/readend* and *writestart/writeend*:

```
readstart (file_descriptor, length_of_buffer), readend(file_descriptor, buffer),
writestart(file_descriptor, buffer, length_of_buffer), writeend(file_descriptor);
```

readstart requests a read operation from file descriptor *file_descriptor* of size *length_of_buffer*, then *readend* fills the data buffer with the data requested by *readstart*, with the same file descriptor argument used by a previous *readstart* call. If there are several *readstarts* with the same file descriptor, *readend* will choose the most recent request. *writestart* and *writeend* have the same calling convention. In our implementation of parallel I/O, we use the *nread* or *nreadp* function call [NCUBE, 1992] to select messages from I/O nodes. The message will contain the I/O node number, data and returned status. With this information, we can easily determine which disk I/O has finished.

3. Design and Analysis of the nCUBE 2 Parallel File I/O

We describe the design of our Unix-based parallel file I/O system and then provide an abstract model (a closed queuing network) in order to analyze its expected behavior. We present the predicted performance based on this model in this section; the experimental results obtained via simulation are presented and discussed in the subsequent section. These two sets of data should be viewed in parallel.

A parallel I/O file system must distribute data among multiple disk devices. The distribution should be as even as possible; otherwise *disk skew* [Kim, 1986] occurs which decreases the overall I/O performance because it degrades parallel access. If we want to write a block b to a disk, the placement rule should determine which disk block b resides locally. The data placement rule should avoid disk skew. *Disk interleaving* is chosen here. For a file striped across disk drives numbered from 0 to $D-1$ (D the total number of disk drives), block b (numbered from 0) is on the disk drive numbered $(b \bmod D)$.

The nCUBE 2 uses independent disk controllers which are attached to separate processors (I/O nodes) in the interconnection network. In interleaving data on n disk devices, the goal is to reduce the read/write access time by a factor of $1/n$. However, it is not easy to achieve an I/O speed-up of n unless we disregard the contention of interconnection communication. Therefore, the connectivity plays an important role in the multiprocessor machine.

We use *PIO File* for a logically single, large file whose blocks are declustered among small files on disk drives of I/O nodes. We index the first block on the first disk drive by 0. The parameters are:

- D : number of disk drives available indexed by D_k , $k \in \{0, \dots, D-1\}$,
- B : number of bytes per block indexed from 0 to $B-1$,
- C : number of blocks for each disk drive that can fit into the communication buffer, which resides in the user process space for the communication space of the I/O operations.

The *Vertex* operating system in each compute node is responsible for all primitive file I/O requests to the file system servers *nsdisk* on each I/O node. The implementation of these *Vertex* file I/O operations is based on the message-passing mechanism. *Vertex* sends data messages with distinct message types to request *nsdisk* to do I/O operations, and *Vertex* waits for the acknowledge or data message replying from the *nsdisk*. For example, if *Vertex* wants to execute a file open system call, *Vertex* finds an empty entry of the user process file descriptor table as a *user file descriptor* to keep the address of the corresponding file server and sends the file open related information with message type N_FOPEN (0x8021) to request the corresponding *nsdisk* to open a file for the user. Then, *Vertex* waits for the *system file descriptor* from *nsdisk* via an incoming message. *Vertex* stores this system file descriptor for successive I/O requests to *nsdisk*. The *read* or *write* I/O operations use the same method.

We need a *PIO File* table to keep the information of open files on each disk drive. The entries of this table are indexed by a user *PIO File* descriptor.

- *offset*: indicates the *PIO File* pointer position in terms of bytes; the file pointer always points to the byte that is ready to be read or written.
- For each open file on the disk drive D_k , $k \in \{0, \dots, D-1\}$,
 F_k : file descriptor of the file on D_k , which is required by *Vertex* to provide the information needed to send the I/O operation message to the file server.
 N_k : address of I/O node where D_k resides.

Each I/O node can have several disk drives attached. The number of these disk drives depends on the disk controller which is connected to an I/O node. Some disk controllers can control up to seven disk drives such as the SCSI disk interface, but others may not. However, an I/O node can only handle one incoming I/O message at a time and then issues the physical disk operation through the disk controller. For example, disk drives D_0 and D_1 are all attached to the I/O node N . *Vertex* issues an I/O operation to the I/O node N for D_1 and then issues another one for D_0 . No matter how long the disk D_1 takes to operate the I/O, the file system of I/O node N guarantees that the acknowledge (ACK) message of I/O will be sent for the first one (D_1), and then for D_0 .

The interconnection network is based on e-routing and wormhole routing [Ni&McKinley, 1993]. The path from a node to another node is fixed according to e-routing. Wormhole routing guarantees that the second message from N will be blocked until the first one is finished. In the *PIO File* table, we need to keep the addresses of I/O nodes where disk drives reside. The reason is that an I/O ACK message only contains the information of the I/O node address and the returned status of I/O operation. We need the I/O node address to search the data structure (message queued linked list) for the matched I/O node address's request queued in that data structure. When an I/O request is issued by *Vertex*, we append information related to this request to the message queued linked list. If an I/O ACK message arrives, we retrieve information of the corresponding queued request. Two primitives are used in our algorithms, namely "ASYNCHRONOUS DISK READ/WRITE" and "READ ACK FROM I/O NODES" (see Figure 3).

PIO OPEN, PIO CLOSE: Because each I/O node contains an independent file system, we use a file in each disk drive to store the declustered block data. *PIO OPEN* will return a user *PIO File descriptor* (*pdf*) to be used in successive I/O operations. *PIO CLOSE* is used to close all the involved files on disk drives.

```

• ASYNCHRONOUS DISK READ/WRITE
{issue an asynchronous disk read or write regarding  $F_k$  with  $D_k$  and  $N_k$ ; the current file position is offset}
if disk operation is read
then { read nbytes of data from  $D_k$  into user space buffer } call readstart( $F_k$ , nbytes);
else { write nbytes of data from user space buffer to  $D_k$  }
    call writestart( $F_k$ , buffer, nbytes);
Insert information of  $D_k$ ,  $F_k$ ,  $N_k$ , buffer, offset, nbytes
into data structure at the end of message queued linked list;

• READ ACK FROM I/O NODES
Wait for an incoming I/O ACK message from any I/O node with message type read returned or write returned;
if message type is read returned
then { read returned message contains I/O node address, # of bytes read, data from disk, disk operation
status}

```

```

Search for the first with the same I/O node address from the beginning of message queued linked list;
Retrieve information of  $D_k, F_k, N_k, buffer, offset, nbytes$ ;
Move read-in number of bytes data from communication buffer to the user space buffer;
else {write returned message contains 1. I/O node address 2. disk operation status or error code}
Search for the first with the same I/O node address from the beginning of message queued linked list;
Retrieve information of  $D_k, F_k, N_k, buffer, offset, nbytes$ ;
Delete this data structure from message queued linked list;
    
```

Figure 3: Asynchronous Disk Read/Write, Read ACK From I/O Nodes

However, when processes are terminated, Vertex will close all the files for the user processes. The algorithms of *PIO OPEN* and *PIO CLOSE* are given in Figure 4.

```

Algorithm PIO OPEN
Input: 1. file name used in each  $D_k, k \in \{0, \dots, D-1\} : filename$ 
       2. type of open (e.g., READ, WRITE) : type
       3. file permissions (for creation type of open) : flag
Output: user PIO File descriptor: pfid
{ Find an empty PIO File table entry, initialize its offset and return the index pfid;
  For each disk drive  $D_k, k \in \{0, \dots, D-1\}$ 
  do open_file_name =  $D_k$  volume name + filename;
      { open a file on  $D_k$  and get a file descriptor  $F_k$  from Vertex }
      call open(open_file_name, type, flag) and return a  $F_k$ ;
      { save  $F_k$  and  $N_k$  for later use.  $N_k$  : address of I/O node where  $D_k$  resides }
      Insert  $F_k$  and  $N_k$  into the data structure of PIO File table entry index by pfid;
  return pfid;
}

Algorithm PIO CLOSE
Input: user PIO File descriptor : pfid
Output: file status
{ Retrieve PIO File table entry indexed by pfid; { get the information of  $F_k$  }
  For each disk drive  $D_k, k \in \{0, \dots, D-1\}$ 
  do { close a file on  $D_k$  and Vertex remove  $F_k$  from its file descriptor table }
      call close( $F_k$ )
      Remove the PIO File table entry index by pfid;
  return file status;
}
    
```

Figure 4: Algorithm Parallel File I/O Open, Parallel File I/O Close

PIO READ, PIO WRITE: We use the disk striping technique for data and files. *PIO READ* and *PIO WRITE* do not take the contention of interconnection network into consideration. Initially, *PIO READ* and *PIO WRITE* issue a disk operation to each disk drive. Then, whenever an I/O ACK message from any disk drive comes, the request of the next corresponding data block belonging to this disk drive will be issued.

In Figure 5, the predefined parameter *C* is the number of blocks in the communication buffer of each disk drive. If we set *C* to be 2 instead of 1, *PIO READ* and *PIO WRITE* will issue two disk

operations to each disk drive initially. This will keep the disk drives busier. However, the product of block size, number of disk drives and *C* must not exceed the size of the communication buffer which is allocated in and uses up user memory space.

PIO SEEK: To allow random access of files, we also allow resetting the file pointer for the *PIO File*. The calling convention of *PIO SEEK* (code omitted here) is the same as the UNIX *Iseek* system call.

```

Algorithm PIO READ/WRITE
Defined Parameters: B = bytes per block, D = # of disk drives, C = # of blocks in communication buffer for each drive.
Assumption: bytes of communication buffer  $\geq B * D * C$ .
Input: 1. user PIO File descriptor : pfid,
       2. address of buffer in user process : buffer,
       3. number of bytes to read/write : nbytes.
Output: count of bytes copied into user space or written to PIO file.
{
  BUFFER  $\leftarrow$  buffer, NBYTES  $\leftarrow$  nbytes; { keep information used later }
  Retrieve PIO File table entry indexed by pfid; { get the information of offset, F_k, N_k }
  { initiate Disk Read/Write for each block of communication buffer per disk drive }
  c  $\leftarrow$  1;
  while nbytes > 0 and c  $\leq$  C
  do x  $\leftarrow$   $\lfloor offset / B \rfloor \bmod D$  { first disk drive to issue read/write operation }
      k  $\leftarrow$  x; { repeat issuing read/write operation in k, k+1, \dots, D-1, 0, 1, \dots, k-1 sequence }
      repeat
      do bytes_left_in_block  $\leftarrow$   $B - (offset \bmod B)$ ;
          if nbytes < bytes_left_in_block
          then bytes_op  $\leftarrow$  nbytes;
          else bytes_op  $\leftarrow$  bytes_left_in_block; { issue physical disk read/write and insert  $D_k, F_k, N_k$ ,
buffer,
                                                                    offset, bytes_op into data structure of message queued
linked list }
          ASYNCHRONOUS DISK READ/WRITE; { update new buffer, nbytes and offset }
          buffer  $\leftarrow$  buffer + bytes_op; nbytes  $\leftarrow$  nbytes - bytes_op; offset  $\leftarrow$  offset + bytes_op;
          k  $\leftarrow$  (k + 1)  $\bmod D$ ; { next disk drive to issue read/write operation }
          until (nbytes  $\leq$  0) or (k = x)
          c  $\leftarrow$  c + 1; { next block of buffer per disk drive }
      { continue to issue read/write operations until NBYTES of bytes copied into user space or written to PIO file }
  while message queued linked list is not empty
  do { read any first ACK message from I/O nodes and retrieve  $D_k, F_k, N_k, buffer, offset, bytes_op$  related to this
      ACK from data structure of message queued linked list, then delete this data structure from list }
      READ ACK FROM I/O NODES;
      buffer = buffer + bytes_op +  $B * (D * C - 1)$ ; { next tentative block to read }
      if buffer < BUFFER + NBYTES
      then offset  $\leftarrow$  offset + bytes_op +  $B * (D * C - 1)$ ;
          nbytes  $\leftarrow$  BUFFER + NBYTES - buffer;
          bytes_left_in_block  $\leftarrow$   $B - (offset \bmod B)$ ;
          if nbytes < bytes_left_in_block
          then bytes_op  $\leftarrow$  nbytes;
    
```

```

else bytes_op ← bytes_left_in_block; { issue physical disk read/write and insert  $D_k, F_k, N_k$ .
buffer,
linked list }
offset, bytes_op into data structure of message queued
ASYNCHRONOUS DISK READ/WRITE;
else continue; { exceed the length of user buffer to read/write }
update PIO file table offset for next read/write;
return (total number of bytes read/write);
}
    
```

Figure 5: Algorithm Parallel File I/O Read/Write

In *PIO WRITE/READ*, if D is the disk size and C is the number of blocks in the communication buffer per disk drive, then there are at most $D * C$ active messages in the whole system if there are no other jobs in the system. Each message needs to start from the compute node and then goes through the interconnection network to the file server. After the service of I/O, the file server sends another message (ACK) back to the compute node. However, we can view this so that this message carries the return value back to the compute node. In the middle processing of *PIO WRITE/READ*, the number of active messages is always $D * C$; whenever a message comes back, it issues another message to the same destination. Of course, this is not true when *PIO WRITE/READ* is almost finished because some disk drives may have done all of their corresponding data blocks and could be idle. However, if we only consider very large files, the system almost always has $D * C$ active messages running through it.

This kind of workload suggests the *closed queuing network model* [Jain, 1991] which has no external arrivals or departures in the system. The jobs (messages) in the system keep circulating from one queue to the next. The total number of jobs (messages) in the system is constant. Each job goes through several service centers to receive the system resources and then re-circulates. Service centers may be of two types, *queuing center* and *delay center*. Jobs at a queuing center compete for the use of the server. Thus the time spent by a job at a queuing center has two components: waiting time and service time. Queuing centers are used to represent any system resource where jobs compete for service, e.g., the interconnection network and file system servers. At the delay center, jobs are allocated their own server, so there is no competition for service. Thus the residence time of a job at a delay center is exactly the job's service time. We consider messages as jobs. Each message must receive service from the system resource and then goes back to the same compute node. We use *PIO WRITE* or *PIO READ* to predict the performance and explain how influential the contention of interconnection network is. This illustrates that useful results such as how to define the three parameters D , B and C can be obtained.

Figure 6 shows our model. There are four service centers, all queuing centers. Each I/O request message competes with other I/O request messages for system resources. The first stage is the CPU which is responsible for the computation of the data block placement rule and copying data from the user space buffer to the communication buffer for data transmission. The CPU queuing service center takes one I/O request message at a time.

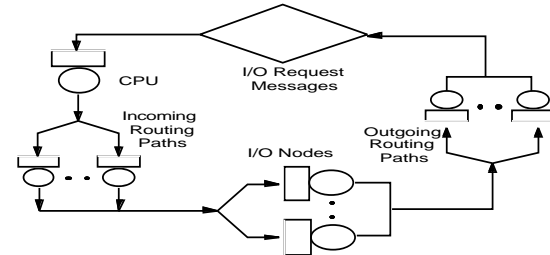


Figure 6: Parallel File I/O Closed Queuing Network Model

The next stage is the interconnection network with the e-routing scheme defining paths to disk drives for I/O request messages. If any two I/O request messages use the same DMA channel to route their respective messages, one waits until the other finishes its routing. These two paths are overlaid due to the shared channel; these paths are *channel-joint*. Paths with no common channel shared are said to be *channel-disjoint*. Channel-disjoint paths are viewed as separate queuing service centers. Each I/O request message chooses its queuing service center to go through to its corresponding I/O node. For example in Table 1, if compute node 3 wants to write three data blocks to I/O nodes 1, 2 and 3 through compute nodes 7, 11 and 15, respectively, the three paths are as follows:

Path 1 : 3→7→ I/O Node 1 Path 2 : 3→11→ I/O Node 2 Path 3 : 3→7→15→ I/O Node 3.

Paths 1 and 2 are channel-disjoint, but Paths 1 and 3 are not. Path 2 and Paths 1 and 3 are two channel-disjoint routings. Thus we need two queuing service centers A and B to represent the two channel-disjoint routings with A serving Paths 1 and 3 and B only Path 2. The I/O request message to I/O nodes 1 or 3 not only will spend time in data transmission but also for queuing in queuing service center A if any.

Definition:

$$\bar{P}(x, y) = \begin{cases} \text{right - most different bit position (index from 0) of } x \text{ and } y & \text{if } x \neq y, \\ -1 & \text{if } x = y. \end{cases}$$

Theorem: For any two paths $Path_1(x, y)$ and $Path_2(x, z)$, if $\bar{P}(x, y) \neq \bar{P}(x, z)$ then paths $Path_1(x, y)$ and $Path_2(x, z)$ are *channel-disjoint*. Therefore, if there are n paths $Path_k(S, D_k)$, $k \in \{1, \dots, n\}$, from a common source S to n destinations $D_k \in \{1, \dots, n\}$, then there are m *channel-disjoint routings*, where m is the number of different values of $\bar{P}(S, D_k)$ for k from 1 to n .

Proof: We consider a source $x = (x_{n-1} x_{n-2} \dots x_j \dots x_i \dots x_1 x_0)_2$, and two destinations $y = (y_{n-1} y_{n-2} \dots y_i \dots y_1 y_0)$ and $z = (z_{n-1} z_{n-2} \dots z_j \dots z_1 z_0)$. We assume that i is the right-most bit

position where x and y differ, and j for x and z . If $i \neq j$, then we know that the source node uses channel i or j to route to node $(x_{n-1} x_{n-2} \dots x_j \dots \bar{x}_i \dots x_1 x_0)_2$ or $(x_{n-1} x_{n-2} \dots \bar{x}_j \dots x_1 x_0)_2$ first, and then continues to y or z , respectively (the complement of x_i is \bar{x}_i). In e -routing, the message travels in ascending order of the dimension. Therefore, nodes through which the path from x to y passes are all with least significant bits $\bar{x}_i \dots x_1 x_0$, but those nodes which the path from x to z travels have least significant bits $x_i \dots x_1 x_0$. They are all different nodes; thus, it is not possible to use the same channel in their respective routing. Therefore, if paths use different channels to take off from a common source, then they are channel-disjoint. If there are m different channels, then we have m channel-disjoint routings.

The third stage is the I/O nodes system. Each I/O request message needs to be handled by their corresponding I/O node first and then be transferred to a physical I/O operation to its target disk drive by its I/O node. Each I/O node can only handle one incoming I/O message at a time; it must wait until an I/O operation finishes so that it can issue another I/O operation to their attached disk drives. If there are some I/O messages arriving at the same I/O node at the same time, they will be queued.

The last stage is similar to the second stage. If the I/O request is *write*, it is only used for the transmission of the returned status of an I/O operation. If the I/O request is *read*, it is used for the transmission of the read-in data block and the returned status. When an I/O request message goes through the fourth stage, it waits for the CPU and carries another I/O request message to repeat the process again.

We classify I/O request messages into different clusters (classes) according to their destination. I/O request messages belonging to class i must use the resource of I/O node i even if another I/O node is idle and has free resources. Based on this multiple class model, we use the *Multiple-Class Closed Queuing Network Model* [Jain, 1991] to derive our predicated performance measurement.

We obtained the following system parameters using monitor programs to estimate the service times (service demand) of service centers. This does not include any time spent waiting for service.

- CPU Service Center (Stage 1)
 - { including memory copy of data and computation of placement rule }
 - Data Block $B = 4$ Kbytes \rightarrow 800 microseconds
 - Data Block $B = 8$ Kbytes \rightarrow 1065 microseconds
 - Data Block $B = 16$ Kbytes \rightarrow 1820 microseconds
- File Server Disk Read/Write (Stage 3)
 - { including overhead of file management and physical disk I/O }
 - Data Block $B = 4$ Kbytes \rightarrow 17300 microseconds
 - Data Block $B = 8$ Kbytes \rightarrow 19500 microseconds
 - Data Block $B = 16$ Kbytes \rightarrow 20500 microseconds
- Interconnection Network Service Center (Stages 2 and 4)
 - { assumes a message travels an average of five dimensions in our 10D hypercube }
 - Data Block $B = 4$ Kbytes \rightarrow 1854 microseconds
 - Data Block $B = 8$ Kbytes \rightarrow 3697 microseconds
 - Data Block $B = 16$ Kbytes \rightarrow 7384 microseconds
 - A returned status = 8 bytes \rightarrow 15 microseconds

We used different data block sizes (4Kbytes, 8Kbytes and 16Kbytes) with the same parameters - eight I/O nodes and only one channel-disjoint routing (one service center in stage 2) to predicate the performance of *PIO WRITE*. We used only one active I/O message for each I/O node. We neglect the influence of the fourth stage because very little time is spent there (15 microseconds). Table 2 lists some performance measurements:

Table 2: Predicted Performance with Varying Data Block

	System Response Time for each I/O Request (msec)	CPU Residence Time for each I/O Request (msec)	Interconnection Network Residence Time for each I/O Request Message (msec)	Interconnection Network (messages)	Utilization of Each I/O Node (%)
B=4 Kbytes	22.27	1.06	3.91	1.40	77.67
B=8 Kbytes	33.00	1.39	12.11	2.94	59.09
B=16 Kbytes	59.69	2.36	36.83	4.94	34.34

A class i I/O request message carries a data block B to its corresponding I/O node i . The spent time in the I/O system is 22.27, 33.00 or 59.69 msec for 4 Kbytes, 8 Kbytes or 16 Kbytes, respectively. In this example, there are eight active messages in the system. The utilization (the fraction of time the resource is busy servicing requests) of each I/O node is lower for 16 Kbytes data block because of the interconnection network congestion. The average number of I/O request messages in the Interconnection Network for 16 Kbytes data block is 4.94, which means any I/O request message needs to wait on average for an additional four messages (4.94-1) in order to finish its jobs. The I/O request message for 4 Kbytes data block almost needs to be queued in the Interconnection Network. However, the disk operational rate for a 4 Kbytes data block (4/0.0173=231.2 Kbytes/second) is slower than for a 16 Kbytes data block (16/0.0205=780.5 Kbytes/second). Suppose we have a 2 Mbytes data buffer to write to the I/O system. If a 4 Kbytes data block is used, 512 data blocks need to be transferred to I/O nodes. In this example, each class i I/O request message needs to run 64 (512/8) times to finish 2 Mbytes data *PIO WRITE*. There are 8 active I/O request messages doing I/O operations concurrently in the I/O system. Therefore, the total time spent in *PIO WRITE* is 1425.28 (64*22.27) msec for using 4 Kbytes data blocks and its I/O operational rate is 1.40323 Mbytes/second. The I/O operational rate for 8 Kbytes or 16 Kbytes data block is 1.89394 Mbytes/second or 2.09415 Mbytes/second, respectively. The bigger data block provides a better I/O operational rate. However, the bottleneck seems to be in the interconnection network if we only use one *channel-disjoint routing* like in this example. The percentage of time spent in the interconnection network is 17.56%, 36.70%, and 61.70 % for 4 Kbytes, 8 Kbytes, and 16 Kbytes data blocks, respectively. Although a 16 Kbytes data block has good performance, the interconnection network congestion is a problem.

We use the same parameters as in the above example but we only focus on a 16 Kbytes data block. In the above example, I/O request messages use only one channel-disjoint routing. A message needs to wait for another one completing its transmission. In the following example, we assume we have multiple channel-disjoint routings available. We assign the class i I/O request message going through the channel-disjoint routing $[i / 2]$ in the case of 4 channel-disjoint routings. In the case of 8 channel-disjoint routing, the class i I/O request message goes through the channel-disjoint routing i . As shown in Table 3, when we use four channel-disjoint routings, the I/O operational transfer rate is almost twice that

of one channel-disjoint routing. However, if we add four more channel-disjoint routings (for a total of eight routings), the I/O operational transfer rate is bounded. There is little difference between four and eight channel-disjoint routings; the residence time for four or eight channel-disjoint routing is 7.38 or 9.17 msec, respectively, compared with one channel-disjoint routing.

Table 3: Predicted Performance with Multiple Channel-Disjoint Routings

	System Response Time for each I/O Request (msec)	CPU Residence Time for each I/O Request (msec)	Interconnection Network Residence Time for each I/O Request Message (msec)	Utilization of Each I/O node (%)	I/O Operational Transfer Rate (Mbytes/Sec)
1 Routing	59.69	2.36	36.83	34.34	2.09415
4 Routings	32.53	2.86	9.17	63.03	3.84260
8 Routings	30.81	2.92	7.38	66.54	4.05712

We reserved only one block of communication buffer for a I/O node, the *single buffer* approach. If we reserve two blocks of communication buffers for a I/O node, there are two active I/O request messages in the system for a I/O node, the *double buffers* approach. We could reserve more blocks per I/O node but this uses up local memory. We use the same parameters as before for the *double buffers* approach. I. e., there are 16 active I/O request messages in this I/O system. Two I/O request messages are assigned to the same I/O node. If we use more communication buffers to increase the number of active messages, the utilization of each I/O node is boosted by four or eight channel-disjoint routings (see Table 4). But the utilization of each I/O node or I/O operational transfer rate is bounded when we add extra channel-disjoint routings. With 8 channel-disjoint routings, although we spend less time (9.11 msec compared with 13.09 msec for 4 channel-disjoint routing) in the Interconnection Network, we spend more time in the CPU stage. That is why there is no significant difference between 4 and 8 channel-disjoint routings.

Table 4: Predicted Performance with Double Buffers Approach

	System Response Time for each I/O Request (msec)	CPU Residence Time for each I/O Request (msec)	Interconnection Network Residence Time for each I/O Request Message (msec)	Utilization of Each I/O node (%)	I/O Operational Transfer Rate (Mbytes/Sec)
1 Routing	118.15	2.39	91.29	34.70	2.11595
4 Routings	49.74	3.93	13.09	82.43	5.02614
8 Routings	47.06	4.17	9.11	87.12	5.31237

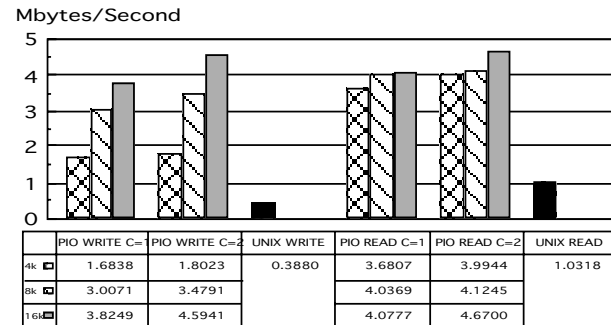
We conclude that the best parameters are 16 Kbytes data blocks and the double buffer approach. If we can at least find four channel-disjoint routings, the parallel file I/O system will attain almost optimal behavior, namely more than 5Mbytes/second I/O operational rate over 8 I/O nodes.

4. Experimental Results

We implemented the user-level parallel file I/O system described in the previous section, utilizing the message-passing mechanism and multiple file servers in I/O nodes to allow the user to take advantage of the multiple disk drives provided by a large-scale parallel machine, here the nCUBE 2. We used one real compute node in the hypercube and eight I/O nodes with multiple channels connected to the hypercube. We use one disk drive for each I/O node, since the number of disk drives attached to a I/O nodes does not affect the performance of I/O nodes. The primary performance metric is the I/O transfer rate (bytes/seconds), the number of bytes of data written to the disk drives divided by the real execution time, incorporating overhead, such as memory copy, interconnection network transmission and calculation of placement rule. Each parallel I/O request involved 2 Mbytes data from the user buffer. Our parameters were as follows:

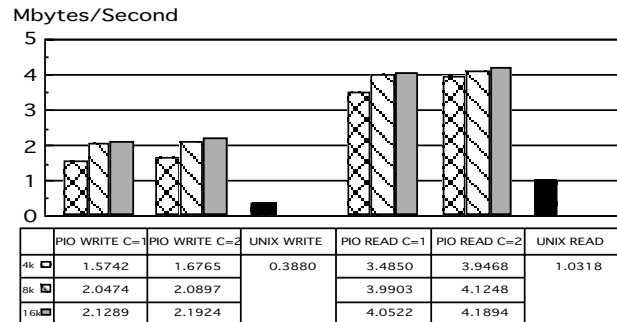
- Disk Drive Size (D) = 8,
- Block Size (B) = 4, 8 or 16 Kbytes,
- Communication Buffer Size (C) = 1 or 2 block(s) per Disk Drive,
- User Data Buffer Size = 2 Mbytes.

Figures 7 and 8 show the experimental results of *PIO READ* or *PIO WRITE*. The best I/O performance for our 2 Mbyte file access over 8 disk drives can be achieved by using 16 Kbyte data blocks and the double buffer approach for both *PIO READ* or *PIO WRITE*. The best speedup of the I/O performance is



C=1: Single Buffer; C=2 : Double Buffers

Figure 7: Experiments: Compute Node with 4 Channel-Disjoint Routings



C=1 : Single Buffer; C=2 : Double Buffers

Figure 8: Experiments: Compute Node with 1 Channel-Disjoint Routing

4.53 or 11.84 for *PIO READ* or *PIO WRITE*, respectively, as compared with the conventional (vendor-supplied) single disk (UNIX) *read* or *write*. These results should be compared with the predicted performance given in Tables 3 and 4.

The nCUBE 2 provides a better multiple-channel connectivity instead of the single channel to an I/O node provided by iPSC/2 CIO. If we centralize I/O activities in some compute nodes with more channel-disjoint routings to I/O nodes, these compute nodes send file data to those with few channel-disjoint routings. Since our approach applies equally to other large-scale machines we can use it in general message passing MIMD systems to get scalable I/O performance.

5. Conclusion

Our user-level parallel file I/O approach satisfied our goal of achieving significant I/O performance to close the gap between the processor and the disk. We provided parallel Unix-like file interfaces by which the user can explore more functionality of parallel file I/O such as the mapping mode described in [DeBenedictis&Rosario, 1992]. Our user-level parallel file I/O is scalable to the hardware architecture. The more disk drives are available, the greater the I/O performance will be (up to a point). Moreover, our approach, developed for the nCUBE 2, can be applied to other large-scale MIMD machines. Although the underlying configuration is important, we can use the Theorem to decide which compute node has the maximum number of channel-disjoint routings to the specific disk drives and then centralize our file I/O.

Our parallel file system is not only good for sequential files but can also be used for random access, which provides the user a flexible way to control data thoroughly. It is transparent to the

user. The user can also use the approach of I/O minimization [Leiss, 1995] to reduce the number of disk I/O activities. The capability of random access provides a good way to achieve it.

6. References

- [DeBenedictis&Rosario, 1992] E. DeBenedictis and J. Rosario: "*nCUBE Parallel I/O Software*", 11th Annual IEEE International Conference on Computers and Communications, 1992, pp. 117-124.
- [Intel, 1988] Intel Corporation: "*iPSC/2 I/O Facilities*", 3rd Conference on Hypercube Concurrent Computers and Applications, 1988.
- [Jain, 1991] R. Jain: *The Art of Computer Systems Performance Analysis*, John Wiley & Sons Publishing Company, Inc., 1991.
- [Kim, 1986] M. Kim: "*Synchronized Disk Interleaving*", IEEE Transactions on Computers, Vol. C-35, No. 11, Nov. 1986, pp. 978-988.
- [Leffler et al., 1989] S. Leffler, M. K. McKusick, M. Karels, and J. Quarterman: *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.
- [Leiss, 1995] E. L. Leiss: *Parallel and Vector Computing: A Practical Introduction*, McGraw-Hill Publishing Company, Inc., 1995.
- [Leiss&Johnson, 1988] E. L. Leiss and O. G. Johnson: "*Advances in High-Performance Processing of Seismic Data*", Supercomputers in Seismic Exploration, E. Eisner (ed.), Pergamon Press, Oxford.
- [NCUBE, 1988] NCUBE Corporation: "*The NCUBE Family of High-Performance Parallel Computer Systems*", 3rd Conference on Hypercube Concurrent Computers and Applications, 1988, pp. 848-851.
- [NCUBE, 1992] NCUBE Corporation: *nCUBE 2 On-line Documentation*, May 1992.
- [Ni&McKinley, 1993] L. M. Ni and P. K. McKinley: "*A Survey of Wormhole Routing Techniques in Direct Networks*", IEEE Computer, vol. 26, Feb. 1993, pp. 62-67.
- [Reddy&Banerjee, 1989] A. L. N. Reddy and P. Banerjee: "*I/O Issues for Hypercubes*", IEEE Proceedings of International Conference on Supercomputing, June 1989, pp. 72-80.
- [Sohn&Leiss, 1996] B. J. Sohn and E. L. Leiss: "*Performance of Bandwidth Intensive Applications Under SUNMOS on the Intel Paragon*", Report to Sandia National Laboratories, Department of Computer Science, University of Houston, March 1996.
- [Wang&Leiss, 1996] Y. Wang and E. L. Leiss: "*User-Level Parallel File I/O*", Proceedings, CLEI PANEL'96 - Conferencia Latinoamerica de Informatica, June 3-7, 1996, SantaFe de Bogota, Colombia, 383-394.
- [Zhao, 1997] H. Zhao: "*A Congestion Study: The Intel Paragon Under the SUNMOS and Puma Operating Systems*", M. S. Thesis, Department of Computer Science, University of Houston, August 1997.