nCUBE's Parallel I/O with Unix® Compatibility

Erik DeBenedictis and Peter Madams

nCUBE Corporation

Abstract

This paper presents a parallel I/O facility based on an extension of Unix. This facility, both scalable and transparently integrated, is part of the upcoming release 3 of nCUBE's system software.

With the addition of scalability for I/O as well as computing, distributed memory machines become balanced between the two functions, suiting them for a wider applications range than their traditional domain of computation-intensive tasks.

The basis of the I/O facility is a system-level data structure called a mapping function. A mapping function describes how data from the parts of a parallel program or parallel I/O device are combined to form a single I/O stream. Combining mapping functions from senders and receivers allows the system to use an optimal communications strategy.

Finally, these facilities are added as extensions to Unix. For programs with a single processor, an exact Unix environment is provided. For parallel programs, the Unix environment is extended in a natural way to accommodate parallel I/O.

1 Introduction

The rule of thumb in the supercomputer industry is "a megabyte per megaflop is balanced." This means that a computer has balanced I/O and computing capability if the I/O rate in megabytes per second is equal to the computation rate in megaflops per second. Our goal was to enhance the system software to "balance" the nCUBE, enabling a broader applications range.

1.1 Applications range

Early hypercube[1] systems were not balanced. Figure 1 shows a traditional hypercube application. The program shown, integrated circuit simulation, reads perhaps a dozen megabytes of circuit description, a few kilobytes of test vectors, and generates a few kilobytes of output. In between it does a *tremendous* amount of computing. While early hypercubes had plenty of computing power, their I/O was the disk on the host workstation, which obviously limited the rate to that of the workstation. Hypercubes and their successors are, in fact, still concentrated on problems with scalable and potentially huge computation demands, but constant and potentially trivial I/O requirements. Since such applications are far from the norm expected by the supercomputer industry, one would expect that there are not very many such applications. This is consistent with the current use level of distributed memory machines.

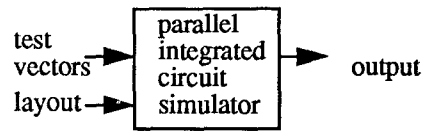

Figure 1: A Computation-intensive Application

Recently introduced parallel I/O hardware has the potential of increasing I/O capacity to the point where it balances computing capacity. Figure 2 illustrates a system with parallel I/O hardware. The I/O device, secondary storage in the example, is a replication of a basic unit consisting of a disk drive and a communications path to the computational array. With appropriate data layout, storage capacity, transfer bandwidth, and average seek rate can increase proportionally to the degree of replication [Patterson 88]. One could say, therefore, that the I/O capability

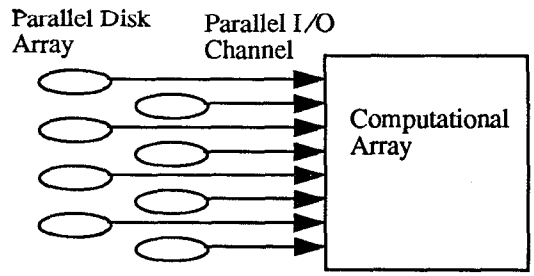

Figure 2: I/O-intensive Application

1. Hypercubes were a precursor to distributed memory computers.

0-8186-2290-3/91/0000/0270/$01.00 © 1991 IEEE

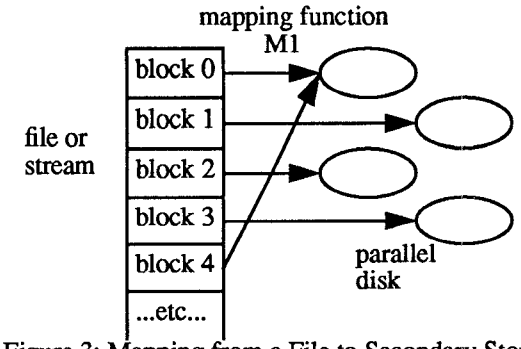

Figure 3: Mapping from a File to Secondary Storage

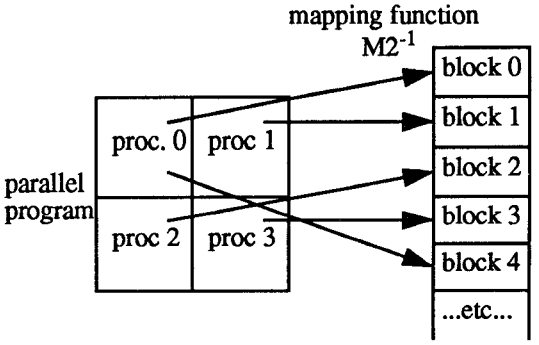

Figure 4: Mapping from a Parallel Program to a Stream

of such a system is "scalable." Now, a scalable machine where the computing and I/O resources scale at the same rate satisfies the criterion for a balanced system.[2]

We say only that there is the *potential* for balancing scalable computing and I/O. While there have been some demonstrations of I/O-intensive applications, the number and generality of these demonstrations is significantly behind those of computation-intensive applications. We believe this is due to the relative newness of parallel I/O and the consequent lack of software. To remedy this, the design of our parallel I/O software is presented, and this paper is our contribution toward terminology for discussing parallel I/O.

For pragmatic reasons, Unix compatibility is a major goal in both the details and concepts of nCUBE's parallel I/O system. If the details of using an nCUBE are the same as Unix, then existing Unix programs will run without adaptation, and programmer training requirements are reduced. By allowing existing Unix programs to run, if even on only a single processor, a tremendous amount of code automatically becomes available. This code includes not only customer-written code, but also Unix tools such as electronic mail and text editors. The presence of such tools make a system much more approachable. The operating system interface can similarly be made more approachable. For the many operating system interface issues that are the same between an nCUBE and a Unix machine, using the exact Unix interface avoids the need for training. The approachability issue applies to concepts also. There are many concepts in an operating system's treatment of I/O and computing that are independent of parallel versus non-parallel implementation. Our approach, therefore, is to express parallelism as a natural and transparent extension of Unix.

1.2 Approach

The basis of parallel I/O is *mapping functions*. Figure 3 shows mapping function M1, which describes the mapping of the bytes in a file (or stream) to positions in a series of disks. This example is the same as disk striping [Salem 86]. Figure 4 shows a mapping function that combines the output from the various processing elements of a parallel program into a single file (or stream). We use $M2^{-1}$ rather than M2 to enforce the convention that mapping functions always direct data from a file (or stream) to a parallel device. This example is the same as a parallel program writing a matrix (distributed in memory in the usual way) [Fox 88] to a disk file. The key to efficient parallel I/O is for the program to transfer its data to the disks without any bottleneck. In our implementation of parallel I/O, each program and I/O device specifies the mapping function to the operating system. The operating system then computes the function $M = M1 \circ M2^{-1}$.[3] In figure 5, M represents the mapping from processors in the program to disks, without reference to the file (or stream). With function M known by all processors, the most efficient communications paradigm can be chosen.

The basis of making parallel I/O compatible with Unix is Unix's merging of the concepts of file and network interfaces [AT&T 90]. To avoid bottlenecks in file output, data must flow directly from the generating processor to the appropriate disk. This means that at some low level a connection to a parallel file must be able to send data to multiple destinations, just like a network connection. In

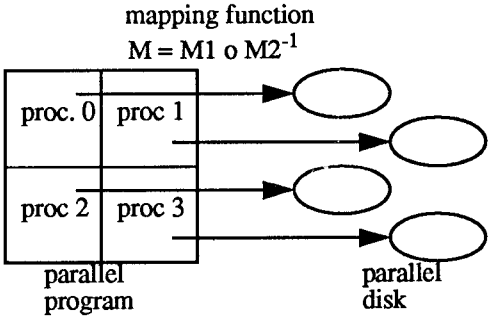

Figure 5: Combined Mapping Functions

2. Strictly speaking, not only do the scaling rates have to be the same, but the point "one megabyte/second I/O rate and one megaflop/second computing rate" must be on the curve.

3. o represents function composition from the left: for functions a and b, a o b is b(a()).

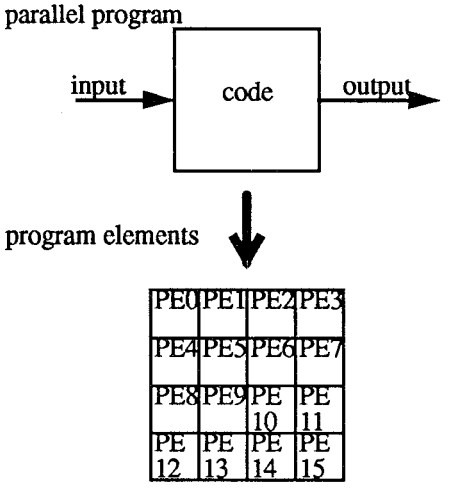

Figure 6: Parallel Program and Program Elements

the nCUBE system, therefore, connections to parallel files use the Unix facilities for a network connection. In Unix terminology, a special network is created for the open file where the only devices on the network are the various disks. Since files and networks share many of the same facilities, standard Unix allows one to write to a network connection, causing an error (since one is supposed to put messages to network connections). On an nCUBE, however, writing to the network representing a parallel disk invokes the mapping software. This software breaks the data into pieces and sends them as messages to the appropriate destinations.

2 System Features

The remainder of this paper describes the features being phased in to release 3 of nCUBE's system software. For completeness, the discussion begins with a review of parallel computing support. Following this are six sections describe I/O issues.

2.1 Combining processors to achieve scalable computing

There is a very simple paradigm underlying the way nCUBE software addresses scalable computing. This paradigm is that many processing elements (PEs) of fixed performance are combined to form parallel programs of any degree of performance.

The nCUBE operating system extends the concept of a program to embody both parallel programs and their constituent programing elements, as shown in figure 6. Parallel programs are like regular programs in that they can transform input to output under the control of a program (code). With the nCUBE operating system, however, a parallel program may consist of a number of PEs, each being a computer with its own CPU, memory, and (typically) executing a copy of the program code.

Both uniprocessor and parallel programming capabilities are available to the user. Where a parallel program consists of exactly one PE, the parallel program and PE become indistinguishable and collectively identical to a Unix process. For compatibility, the nCUBE operating system gives Unix process semantics to single PE parallel programs.

For multiple PE parallel programs, the nCUBE operating system invokes them in Single Program Multiple Data (SPMD) mode. In this mode, each statement is executed by all the PEs, but typically on different data. Subroutines that are called while in SPMD mode execute in parallel, hence forming the basis of a parallel software infrastructure (library). Unlike Single Instruction Multiple Data (SIMD) machines, a PE can easily and efficiently execute a unique statement and later rejoin the collective execution behavior of the other PEs. It is also possible to depart from SPMD mode altogether, loading different programs on different PEs, resulting in Multiple Instruction Multiple Data (MIMD) execution.

While some Unix process attributes are mapped without change to the parallel program or PE abstractions, other process attributes are significantly changed and may be partially mapped to both abstractions. For example, since the nCUBE has distributed memory, it is obvious that the memory management system calls should be mapped to the PE abstraction. On the other hand, when a user's program goes awry, only one interrupt signal should be sufficient to abort the entire parallel program. The interrupt signal is therefore mapped to the entire parallel program. The adaptation is I/O is less straightforward. Consider a program containing a division. Before dividing, one might check for division by zero and print an error message to the user's terminal. Since each PE has different data, these error messages are clearly associated with a PE, rather than the parallel program. On the other hand, consider a sorting program. One wants a parallel sorting program because it runs faster. Essential to achieving high speed is the distribution of input and output among the PEs. I/O from a sorting program is therefore an attribute of the program, not any particular PE.

2.2 Combining I/O channels for scalable I/O

The paradigm underlying parallel I/O is a simple extension of the parallel computing paradigm. In the I/O paradigm, many communications paths of fixed bandwidth are combined to form a channel of greater bandwidth. For a program to have the capacity to generate I/O at high enough rates to require multiple channels, the program must itself be scalable, which gives it enough PEs to serve as the endpoints of the parallel I/O channel.

The nCUBE operating system extends the concept of an I/O connection to embody both parallel I/O connections and their constituent physical paths, as shown in figure 7. To the user, a parallel I/O connection is just like a regular connection in that it conveys data between pro-

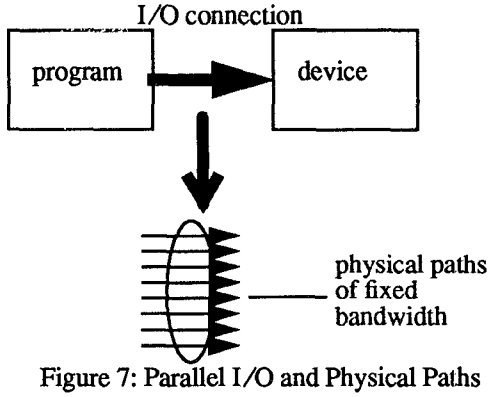

Figure 7: Parallel I/O and Physical Paths

grams and/or I/O devices. The nCUBE operating system implements parallel I/O connections by using many identical physical paths. The nCUBE operating system makes this use of multiple physical paths transparent, however, by taking charge of the allocation of the data to the communications paths. In connecting to a communication channel, a parallel program specifies how the data is to be distributed (or mapped) to the different PEs of the parallel program. Parallel secondary storage is handled similarly: file data is mapped to several disk drives. The operating system, which is present at both ends of the channel, uses its knowledge of the mappings at both ends to generate a parallel communications strategy.

As with computing, I/O is available in both high performance parallel and backwards compatible conventional modes. Unlike computing, however, the selection of the I/O mode is the responsibility of the operating system, not the user. The typical example of high performance I/O is where programs or I/O devices with the same degree of parallelism communicate with the same mapping. In this case, each data source will be paired with one data destination and the pair will have one dedicated communications path. Data will flow in parallel and without bottlenecks. The user's only role is to specify how each program works; the operating system decides what I/O method to use. Because of this, the parallel program could be paired with a uniprocessor program (say for debugging purposes). In this case, the operating system would choose a I/O method that gives correct behavior and as much performance as is feasible under the circumstances.

Significant Unix extensions are brought into play when an open file, or more generally a communications endpoint, is owned by a parallel program.

A low level example of such a communications endpoint would be the pipe between two parallel programs. In such a pipe, the PEs of the two programs would send messages to each other. To send a message, therefore, one must identify which PE in the destination parallel program the message is to be sent to. Receipt of a message similarly identifies the sending PE. This sort of connection is represented in the nCUBE operating system by a Unix-style connectionless transport endpoint.[4]

There is an addressing domain in the nCUBE operating system for selecting among the different PEs of a parallel program. In this addressing scheme the PEs are numbered sequentially from zero (although the addressing scheme also has a process identifier and message type for compatibility with earlier nCUBE releases).

Each PE in both the sending and receiving programs has a file descriptor for the pipe. To send or receive a message, a PE uses the standard Unix system calls for message I/O from a connectionless transport. These calls take the file descriptor for the pipe, a data buffer, and an address as arguments (the sender uses the address, whereas the receiver returns the address of the sender).

The transmission of messages between the PEs of a parallel program is an extension of the parallel pipe just presented. As part of program setup, each program receives a parallel pipe to itself. Messages sent to this file descriptor are delivered elsewhere in the same parallel program. For backwards compatibility and ease-of-programming, simple calls are provided to do message I/O within the PE's own parallel program.

Algorithms may also construct addressing domains for their own purposes. Matrix algorithms are an example of where these may be useful. Where an entire matrix is loaded into a parallel program, it may be useful to do certain operations on a per-row basis. To do this, one can construct addressing domains that embody only the PEs on a particular row. Parallel algorithms written for parallel programs in general can then be applied to the pseudo parallel program.

2.3 nDEVICES

In order to effectively support the efforts of researchers in developing new parallel devices, nCUBE provides a prototyping tool called nDEVICES. The nDEVICES feature allows users to write code for new parallel devices and, in effect, if not in reality, incorporate them into the operating system. Since these devices do not really become part of the operating system, capabilities relying on kernel-level implementation are not available. nDEVICES does fully support all the parallel aspects of implementation, however, which is the critical area of new parallel device research.

---

4. Recent versions of Unix have combined the concepts of open files and network connections. Unix now uses file descriptors to represent both. A user program with data can *write* this data to a file descriptor representing a file, whereas both data and an address are required to put a message (*putmsg*) to a network connection. If one writes to a network or puts a message to a file, an error results. With an address, however, one can *connect* to a particular destination in a network, after which one can *write* data. In Unix terminology, a reference to an entire network is a connectionless transport endpoint, whereas a connection to a particular address on the network is a transport connection.

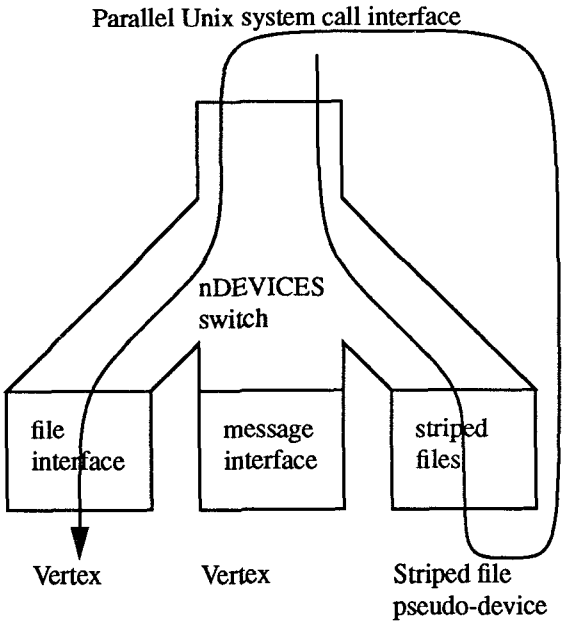

Figure 8: nDEVICES Feature

Figure 8 illustrates the nDEVICES feature. The essence of the nDEVICES feature is the switching of device operations between system and user-supplied devices. The user supplies code to implement all the system-call-level operations on the nDEVICE. In addition, the user supplies code for recognizing the name of, and opening, the nDEVICE. After this code is compiled with the user's program, and an initialization routine called, the nDEVICE is accessible just as though it were part of the operating system. The nDEVICES switch in the illustration has entry points for all standard Unix system calls, plus a small number of enhancements provided by nCUBE. The switch routes calls to these entry points to nDEVICE driver modules. At least two modules are loaded with every program: the file interface and message interface modules. The file interface supports non-parallel file I/O, where *read* and *write* are typical calls. The message interface supports inter program communications, with *getmsg* and *putmsg* being typical calls. Both these modules execute trap instructions to activate the Vertex[5] kernel. A user may optionally load other modules, either user-written or from the library. A striped file module is shown. The striped file module shown implements the segment files on standard Unix files. The call graph in the illustration shows the striped file module recursively calling the nDEVICES switch to manipulate the segment files.

2.4 Inter-program communications

The nDEVICES system always loads a message interface module, which provides messaging facilities. In Unix terminology, the message interface module defines an addressing domain and handles I/O to this domain. The domain is for messages directed between the PEs of a parallel program. The structure of an address in this domain is shown below:.

compatibility routine	Unix equivalent
message input	
nread(data, pattern)	getmsg(nself, data, pattern)
message output	
nwrite(data, address)	putmsg(nself, data, address)
test for message	
ntest(pattern)	ioctl(nself, pattern, ...)
get information	
whoami(...)	ioctl(nself, ...)

Table 1: Compatibility Routines

```
struct ncaddr {
        short pid;          /* 0..n-1 */
        short procid;       /* for multitasking */
        short type;         /* message type */
}
```

The PEs in a n-PE parallel program are numbered 0..n-1, and the pid field in the address identifies this PE. The procid field, used only when multitasking on individual PEs is enabled, identifies the specific process. nCUBE messages have a type associated with them, with non-negative types being available to the user. Consistent with Unix semantics, message I/O can be done with *getmsg* and *putmsg*. In an extension of Unix semantics, however, the message input call specifies an address template. The address template may specify a pid and/or type field explicitly, or may have wildcards for either. Only messages that match the template are considered. The actual address replaces the template when the call returns.

For compatibility with existing nCUBE code, a set of routines are provided that do message I/O only within the calling parallel program. These routines implicitly reference a transport endpoint called *nself*, which is initialized to a reference to the enclosing parallel program. Also, for compatibility with existing code, the details of the way data is passed to these routines is different from the related Unix calls. To give the flavor of these routines, four of the more popular ones are outlined in table 1. Compatibility with Reactive Kernel primitives [Seitz 88] is provided similarly

To know how to distribute data, a PE will need to know the number of PEs associated with the other transport endpoint. The *ioctl* system call returns the number of PEs at the other transport endpoint. There are additional states to deal with the fact that a transport endpoint can be created by a program with some number of PEs (like the shell with one PE) and then inherited by a parallel program with a different number of PEs.

5. Vertex is the name of the nCUBE kernel.

Sometimes one needs I/O not only to the PEs at the other end of the transport, but also to the other PEs at the same end of the transport. nCUBE's parallel graphics provides an example of this. The computational array sorts graphic output before transferring it to the graphics hardware. This sorting is done among the PEs which are connected to the graphics system. To accomplish this, the addressing is augmented: initially, a pid field of 0..n-1 addresses the other transport endpoint, with all other addresses illegal. An *ioctl* call can be used to map the PEs at the same transmission endpoint to otherwise illegal addresses.

The message semantics, or protocol in Unix terminology, will be the same as in earlier nCUBE systems, but may be enhanced in the future. nCUBE messages obey buffered datagram semantics: they are transported reliably, but are placed in buffers at the sending and receiving ends. An advantage of this method is that, while message order is preserved by default, messages in the input queue may also be read according to address and type templates. Where the protocol is understood well enough, one make the message buffer large enough to avoid overflow, otherwise the protocol must be enhanced to deal with lost messages.

Future releases are likely to embody changes in several areas. The underlying message semantics can be changed or enhanced to avoid buffer overflow. In addition, higher level protocol processing through a streams-like mechanism is possible.

2.5 Global I/O mode

Traditional nCUBE global mode is supported by another nDEVICES driver that is loaded by default. While global mode was introduced to deal with programs where all the PEs read the same file, it has wider applicability. Global mode is applicable when an SPMD program does I/O. Since these I/O operations are performed by all the PEs, distributed algorithms can be used to optimize performance. To see the significance of this optimization, consider reading a file in global mode: the disk is activated only once and the contents are broadcasted to all the PEs. Without global mode, the disk would be activated n times resulting in a speedup of n.

2.6 Mapped I/O mode

The semantics of mapped I/O mode [Chen 88] is sufficiently intricate that it will be explained through a series of examples.

Figure 9 illustrates a parallel program doing mapped output to a conventional Unix file. The output from PE0 and the file are illustrated as streams, meaning that they may either be blocks of data of some length, or that they may a flow of data that can continue for the entire time the program runs. A typical mapping function might be to consider the file a sequence of 1024 byte blocks, with the

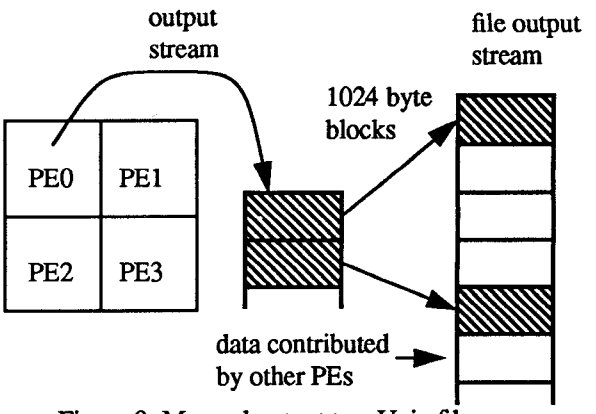

Figure 9: Mapped output to a Unix file

blocks originating from the PEs in a round robin order. The figure illustrates the first two blocks of PE0's output appearing as the first and fourth blocks in the file. This output scenario has been frequently used on previous nCUBE systems, where one PE at a time does output and sends a synchronization message to the next PE. With mapped I/O, each PE specifies the mapping function, after which it outputs its data with no regard to synchronization or block size.

This example illustrates several features of mapped mode. Mapping is a mode that applies to file descriptors; files are not in mapped mode initially, but enter mapped mode through an *ioctl* call that specifies the mapping function. The encoding of the mapping function into a data structure is described elsewhere, but one views the mapping functions as mapping of each byte in a file (or stream) to a specific byte position in the I/O stream of a specific PE. For each byte written in mapped mode, the byte's offset in the PE's I/O stream is translated to a position in the actual file and the byte is written there. Where I/O is noncontiguous in the greater file, the system effectively seeks through the file. In mapped mode, however, each PE has its own read/write pointer for its sub-file to which *lseek* operations can be applied.

Figure10 illustrates a single PE doing output to a parallel disk (striped disk). In reading this example, recall the design goal that single PE programs be Unix compatible and notice that all the parallel activity occurs outside the user's program. The way in which conventional striped disks allocate data to the various segments is similar to the round robin blocked mapping function described in the previous example. In the mapping function for a striped disk, the block size is typically chosen to correspond to the block size on the magnetic media, and the number of PEs corresponds to the number of disks. The mapping function for the disk is a parameter of the disk subsystem, however, and is specified by the system administrator. Application programs just write to the file as though it were a Unix file.

A number of things happen transparently in this example. To avoid bottlenecks, data physically leaving the PE

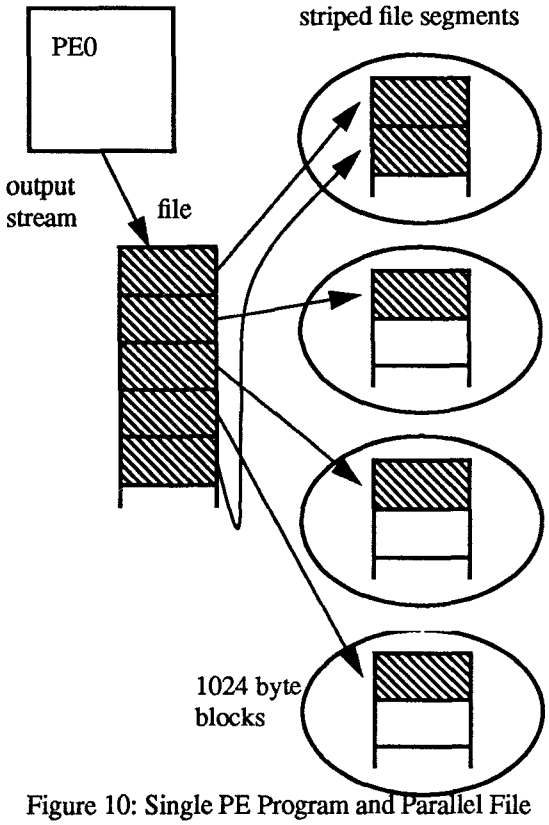

Figure 10: Single PE Program and Parallel File

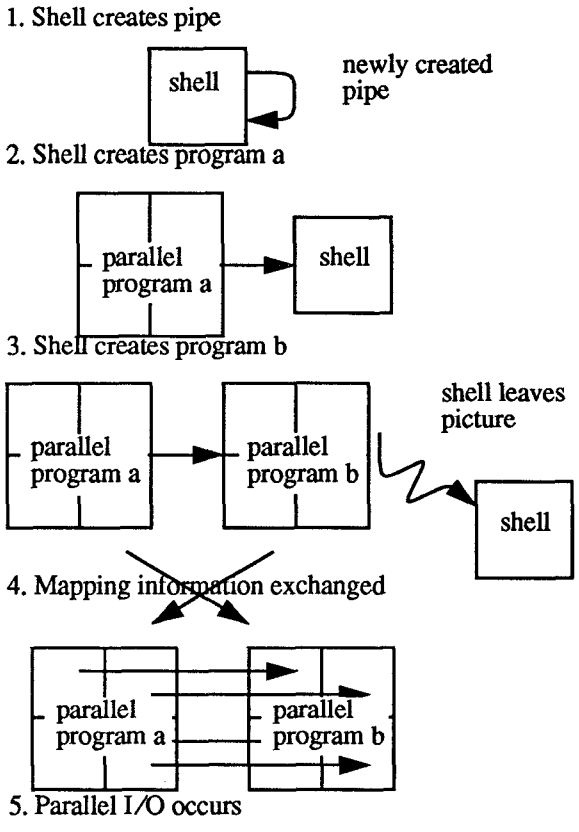

Figure 11: Setup of a Parallel I/O Channel

must go directly to the disk it is stored on. As a result, the file descriptor referencing the striped file is actually a connectionless transport endpoint. At a low level, the system can and will send data to the individual disks with putmsg system calls. At a high level, however, the user does conventional writes. Now, a plain write to a connectionless transport endpoint will give an error message on a Unix system, but will attempt to enter mapped mode on the nCUBE system. In this example, mapped mode can be entered because mapping functions are available for both ends of the connection: the mapping function for the striped disk is provided by the disk controller, and the mapping function for all one-PE parallel programs is necessarily the identity mapping. With information about both mapping functions, system level code can split the output from the one PE into pieces that are sent directly to the appropriate disk drive.

Figure 11 illustrates the setup of the Unix pipeline command alb executed on an nCUBE where a and b become an i- and j-PE parallel programs. For performance reasons, the pipe between a and b should be parallel, meaning that data will flow directly between producing PEs in a to consuming PEs in b. The sequence of events is as follows:

1. The shell first does a pipe system call create both ends of what will become the parallel pipe between a and b. In the nCUBE system pipes are initially in an indeterminate state since it is not known how may PEs will be at each end and what I/O mode each end will be in.
2. The shell then runs program a, connecting one end of the pipe to a's standard output. While program a might immediately try to specify its mapping function (at which time the number of PEs in a is known) and then do output, output will be blocked until after b specifies its mapping, however.
3. Program b is started similarly.
4. Mapping information is exchanged so that all PEs know the mapping functions of both ends of the pipe.
5. At this point, the information necessary to identify the most direct path for communications between the two programs is present where needed and parallel data transfer occurs. High level I/O code in the PEs break up high level I/O transfers into a series of messages directed to PEs on the other side of the parallel pipe.

3 Conclusions

nCUBE's parallel system software integrates support for parallel I/O and enhanced Unix compatibility with it's previous support for parallel programs. Parallel I/O is implemented primarily by system software, thereby allowing any two programs to be interfaced at run time -- including

uniprocessor programs in a Unix-compatibility mode. Inter-program communications are now compliant with Unix networking capabilities, while also retaining proprietary message communications primitives. In addition, the nDEVICES feature allows user-level in-the-field enhancement of parallel-I/O and/or Unix interfacing capabilities for research purposes.

References

[AT&T 90] "Unix System V, Release 4 Programmer's Guide: Networking Interfaces," Prentice-Hall, 1990.

[Chen 88] M. Chen, E. DeBenedictis, "Separate Compilation and Dynamic Linking of Parallel Programs," M. Chen, E. DeBenedictis, Yale University, May 1988.

[Fox 88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker, "Solving Problems on Concurrent Processors," Prentice-Hall, 1988.

[Patterson 88] D. Patterson, G. Gibson, R. Katz, "A Case for Redundant Arrays of Inexpensive Disks," *Proceedings of the 1988 ACM SIGMOD Conference on Management of Data*, Chicago, IL, June, 1988.

[Salem 86] K. Salem, H. Garcia-Molina, "Disk Striping," *IEEE 1986 International Conference on Data Engineering*, 1986.

[Seitz 88] C. Seitz, J. Seizovic, W. Su, "The C Programmer's Abbreviated Guide to Multicomputer Programming," *Caltech Computer Science Technical Report Caltech-CS-TR-88-1*, January 1988 (revision of April 1989).