nCUBE Parallel I/O Software

Erik DeBenedictis
Scalable Computing

Juan Miguel del Rosario
nCUBE Corporation

Abstract

This paper is about I/O system software that makes I/O on a massively parallel computer as scalable as computing. The paper is centered around mapping functions, which form an intellectual framework from which to understand parallel I/O. The I/O system not only provides the raw capability of doing parallel I/O, but is also convenient to use and compatible with Unix. These features are in release 3 of nCUBE system software.

1 Introduction

Massively parallel computers have some fame for being able to solve very large problems, but are regarded as having limited applicability. Perhaps the near-absence of I/O capability has restricted them to the small fraction of applications that are extremely computation-intensive. Providing an I/O capability, however, could let massively parallel computers address a much wider range of new applications.

Figure 1 illustrates the concept of scalability for a massively parallel computer. There is an upper limit to the performance of a computer of conventional design caused by the finite velocity of light and material properties of semiconductors. In a quest for ever higher computer speeds, the scalable, massively parallel, computer architecture was invented. This architecture combines many computers of a convenient size into a single computer with many times the performance.

Until recently, however, massively parallel computers had very limited I/O capabilities. Usually, the massively parallel machine itself was attached to a host computer, and all the I/O was done by that host computer. The host computer, typically a workstation, would often have one disk. While the computing capability could be made as large as one wanted by adding more processors, the I/O capacity could not be increased.

As a result, massively parallel computers have not followed the rule-of-thumb in the supercomputer industry that "a megabyte per megaflop is balanced." This means that a machine where the I/O rate in megabytes per second equals the floating point rate in megaflops per second has a relatively good balance between computing and I/O. In graphing the distribution of computer programs by I/O requirements, figure 2 shows a bell-shaped curve centered on a 1:1 ratio between megabytes and megaflops. Since a massively parallel machine with N processing elements has O(N) computing capability and O(1) I/O capability, such a machine would only be able to address applications to the left of the 1/N point of the graph. Without parallel I/O, therefore, massively parallel computers get more unbalanced as the number of processors increases.

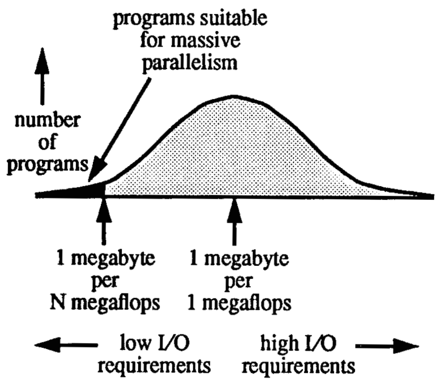


Figure 2: Distribution of Programs by I/O Requirements

As with computing, there is a limit to the performance of an individual I/O device. The speed of a disk drive, for example, is limited by the speed with which actuators move and with which wires can convey data to and from the media. By using many disk drives together, as illustrated in figure 3, one can hope to achieve the same effect as a single disk drive with many times the performance.

Recently, the vendors of massively parallel computer hardware have been introducing hardware for scalable I/O. This hardware interfaces a scalable number of I/O media, such as disk drives, to a massively parallel computer. With a 1:1 ratio between processing elements and I/O media, each processor can produce I/O data at its characteristic rate and be assured that the I/O system can accommodate this data rate. Furthermore, vendors are attempting to apply scalability consistently. Scalable I/O channels appear between scalable I/O devices and scalable computers, for example, to assure that the potential for scalability can be realized.

While hardware is now available for parallel I/O, one observes relatively few applications which involve significant amounts of I/O. We believe this is due to a lack of convenient software.

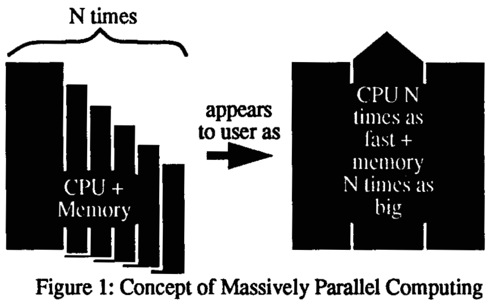


Figure 1: Concept of Massively Parallel Computing

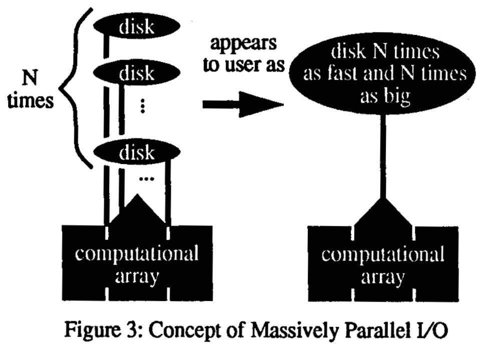


Figure 3: Concept of Massively Parallel I/O

nCUBE has identified this as a general problem and is releasing the system described in this paper as a solution.

1.1 Software Engineering Issues

nCUBE's parallel I/O system software is directed at much more than just giving the user the *capability* of doing parallel I/O: we strive to make parallel I/O *convenient*. As a roadmap for introducing scalable features, we will review in the next section those attributes of conventional I/O systems that make them convenient to use.

The *goal* of parallel I/O is a bottleneck-free pathway between a group of processing elements and an equal number of I/O devices. If this were the limit of the complexity of real parallel I/O activities, then we expect that programmers would code I/O directly into their programs and there would be no need for this paper. Some of the realities of parallel I/O are illustrated in figures 4 and 5 and described below [DeBenedictis 91].

1. A 1-1 mapping of the data between the parallel program and the parallel I/O system does not occur often. If, for example, a user has purchased a different number of disk drives than processing elements, then the mapping cannot be 1-1. If a user has a budget for only 2 disk drives, for example, he will want to get the best performance he can from this configuration, even though the I/O system will not run as fast as the processor.

2. Furthermore, except for programs that use temporary files which are deleted before termination, data must be stored on disk systems in a standard format. This is so that another program (or a text editor) can read the data. It is an unusual coincidence when the j'th processing element in a parallel program produces just the right data for the j'th disk drive according to the standard format. A task for the parallel I/O system then is to translate between distributions convenient for programs and a standard form.

3. The details of how to interface with the parallel device are likely to vary also. On workstations, for example, programmers like to think that files are stored on a disk system. This is often just an abstraction, however. Many workstations are diskless and their files are actually stored on the disk of another computer. In these cases, I/O from a program goes to a network, not a disk. A parallel I/O system may be constructed similarly. High performance disk systems are now available where the actual interface to the computer is via a HiPPI network. The other end of the network connects to a parallel disk system, and software allows the network interface to masquerade as a file system. Furthermore, on a nCUBE system, the high performance network interface is itself a parallel device. The parallel I/O system must have the same interface for all devices.

4. Figure 5 illustrates parallel I/O in a typical software development cycle. Consider program development along a common paradigm: a programmer initially writes and debugs a parallel program using a hand-generated dataset for input and looks at the output with a text editor. When the program is debugged to his satisfaction, the program is run with a full-sized dataset, which resides on a parallel disk. In reality, the character of the I/O task changes radically as parallel I/O is being used. Programmers expect, however, that this transition can be done without rewriting the program. Later on, other programmers may choose to run the program from remote locations. Perhaps the remote locations are connected via a high speed network, such as HiPPI. The I/O task is different again because a network protocol is required rather than a disk protocol. It is essential that this change can be effected without rewriting the program, since the original programmers are not even involved.

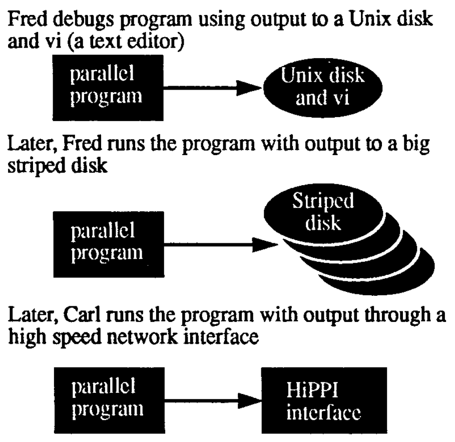

Fred debugs program using output to a Unix disk and vi (a text editor)
parallel program
Unix disk and vi
Later, Fred runs the program with output to a big striped disk
parallel program
Striped disk
Later, Carl runs the program with output through a high speed network interface
parallel program
HiPPI interface

Figure 5: Software Life Cycle

Since conventional operating systems address these issues, extending their structure to parallel I/O is an obvious approach. This structure is illustrated in figure 6. When programming under a conventional operating system, the user writes the application-specific portion of the program with I/O done in a generic way. In Unix, for example, all I/O is imagined to be a simple stream of bytes. The operating system has device drivers, each of which interfaces between generic I/O and a specific device. The program is dynamically connected to the device driver according to the name of the file being accessed. With this method, the source code for the user's program contains no reference to a specific device, making it portable to any device. Since nothing in this approach is incompatible with parallel I/O, it can serve as a model for a parallel I/O system.

To summarize, we are discussing a system that offers both capability and convenience. We draw an analogy to programming languages. There is nothing a programmer can do in Fortran or C that cannot also be done in assembly language. To see this, consider that on modern computers, the Fortran and C compilers trans-

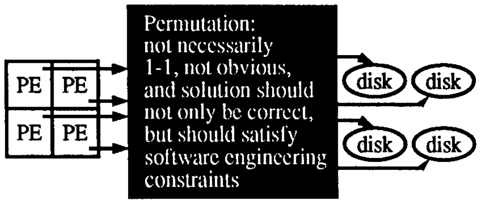

PE
PE
PE
PE
Permutation: not necessarily 1-1, not obvious, and solution should not only be correct, but should satisfy software engineering constraints
disk
disk
disk
disk

Figure 4: Parallel I/O's Data Permutation

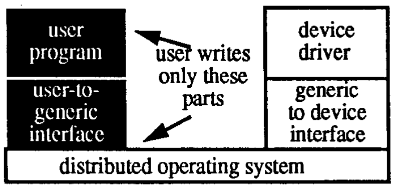

user program
user-to-generic interface
user writes only these parts
device driver
generic to device interface
distributed operating system

Figure 6: System-level Support for Parallel I/O

late their input to assembly language. Likewise, any I/O that can be done with the system-level support described in this paper can also be done with only raw access to I/O. Our claim, however, is that both computer languages and system-level support for parallel I/O are useful for their convenience, even though they do not add fundamental capability.

2 An Intellectual Framework for Parallel I/O

As a single task, we regard the data movement implied by parallel I/O to be too difficult. The problem is to generate the data permutation from the parallel program to a parallel I/O device. This permutation is not necessarily 1:1, may involve a non-obvious data permutation, and is expected not only to be correct, but to satisfy software engineering constraints also. This problem, while tractable, is certainly too difficult for ready acceptance by programmers.

The approach is to break the single difficult problem into three simple problems, as illustrated in figure 7. The starting point is the visualization of a dataset that programmers invariably have as part of algorithm design. The first problem is to describe the way this dataset is distributed among the processing elements of the parallel program. The distributions are called mappings, and this mapping will be called M1. The second problem is to describe the way the dataset is distributed among the disk drives of the parallel disk. This is a mapping also, and is called M2. The third problem is solved by the system software. The problem is to form the composite mapping $M1^{-1}$ o M2, (o representing function composition from the left) which is the overall permutation from the program to the device.

The mapping from the dataset to the program is known by the programmer as part of the algorithm design process. For example, textbooks on parallel programming give extensive coverage to the distribution of dense matrices onto parallel computers. The common methods are to put roughly square blocks, entire rows, or entire columns onto processing elements. The division of the matrix into blocks, rows, or columns contains is the mapping function M1 in an abstract sense. The only new activity for doing parallel I/O is that the programmer will formalize this mapping as a data structure which the program will present to the system software as a system call.

The mapping from the dataset to the I/O device is known by a programmer also, although typically a systems programmer. The well known disk striping technology works by dividing a dataset, viewed as a stream of bytes, into blocks, with the blocks being mapped in a round-robin manner to the different disk drives. This mapping from the dataset to the disk drives is M2.

The sequence of activities for parallel I/O can be summarized as follows:

1. the applications and systems programmers code functions M1 and M2 into their source code,
2. the applications program and device driver inform the system software of mapping functions M1 and M2 through system calls,
3. the system software computes the overall permutation $M1^{-1}$ o M2, and
4. I/O data then flows from the data producer to the data consumer by the most direct path without bottlenecks.

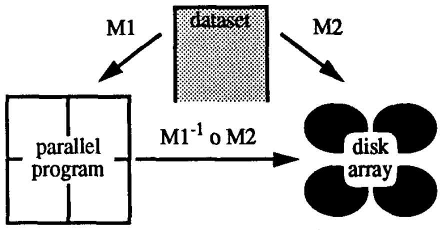


Figure 7: Mapping Functions

3 Variety in Parallel I/O

While the previous discussion has focused entirely on a parallel program doing output to a parallel disk, these concepts apply to a much wider range of activities.

3.1 Parallel Graphics

As a reality demonstration, we will summarize a real-time raster graphics application which is distributed as a sample program on nCUBE software distribution tapes. An early application of the nCUBE was the generation of motion images. This application is somewhat different from those typically found on graphics workstations. Graphics workstations strive to generate still images as quickly as possible. As the images become more complex, the display rate slows down. The nCUBE application is fixed to real-time, and as the images become more complex, more processors are added. Since the nCUBE processor array is scalable, it can accommodate the computing requirements for real-time graphics, but a suitable I/O device is needed. For this application, nCUBE constructed a parallel I/O device, called nGRAPHICS, for displaying raster data generated on the nCUBE.

The nGRAPHICS I/O board, illustrated in figure 8, has connections to the I/O channels on 128 processing elements in the processor array. These 128 channels are routed to 16 nCUBE processors on the nGRAPHICS board. The main memory on these processors is accessible both to the I/O processors and a DMA device which treats the collective memories of 16 processors as a single 512 bit wide memory. Images are displayed by DMA'ing an entire image as a single operation. The display hardware has a fixed mapping for displaying data. Specifically, the first two columns of the image are illuminated by bytes in the memory of the first I/O processor. The second two columns are illuminated by bytes from the second I/O processor, and so on for the first 32 columns. The next 32 columns are similarly illuminated by all 16 I/O processors, and so on until all columns are accounted for. This describes the inverse of M2, which is the mapping from the image to the channels of the parallel I/O device.

The software for doing real-time raster graphics involves describing M1 and then repeatedly outputting images, as illustrated

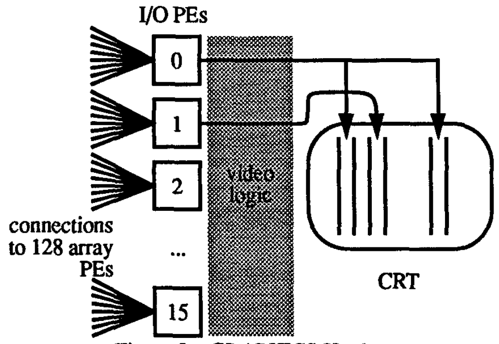
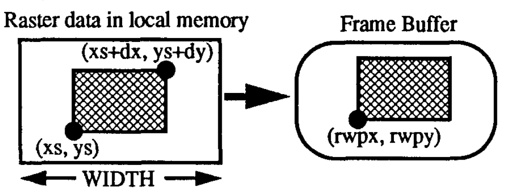


Figure 8: nGRAPHICS Hardware

```
ginit();
```



```
ldrwpx(rwpx, rwpy);
showdef(WIDTH, xs, ys, dx, -dy, zm);

for (; ; )
  showexec(bitmap);
```

Figure 9: A Program's Interface to Parallel Graphics

in figure 9. To specify M1, the programmer identifies a rectangular region for which a processing element will generate the image. In general, this rectangular region will be of a particular size and computed into a subset of a local array. The region will appear on the screen at a particular position. After initializing the graphics with ginit(), the program calls ldwrpx() and showdef() specifying the parameters of the M1 mapping. Following these calls, data can be displayed at a rapid rate by showexec(bitmap), where the bitmap argument is a pointer to the base of the array in local memory.

Figure 10 illustrates the overall flow of graphics data. From the perspective of the system software, the user has specified M1 by calls to ldrwpx and showdef, and the graphics device driver M2 encoded into it. After specifying M1, the system software computes $M1^{-1} \circ M2$, which is the overall data permutation. Each call to showexec then displays the data efficiently. Typically, showexec permutes the data in the computational array, leaving the result on 16 array processors. The data is then transferred in one operation to the graphics board.

3.2 Program Mappings

The mappings of data to programs are significant since they can be done in many ways, three of which are illustrated in figure 11. A 2-dimensional array is typically mapped in one of three different ways: by roughly square blocks, with entire columns on a processing element, and with entire rows on a processing element [Fox 88]. In the nCUBE parallel I/O system all three of these mappings are possible by the proper choice of a mapping function. After the programmer specifies the appropriate mapping function, the data permutation will be performed by the I/O system without further programmer intervention.

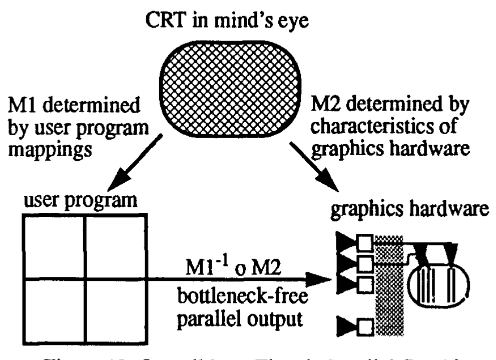

CRT in mind's eye
M1 determined by user program mappings
M2 determined by characteristics of graphics hardware
user program
graphics hardware
M1^{-1} o M2
bottleneck-free parallel output

Figure 10: Overall Data Flow in Parallel Graphics

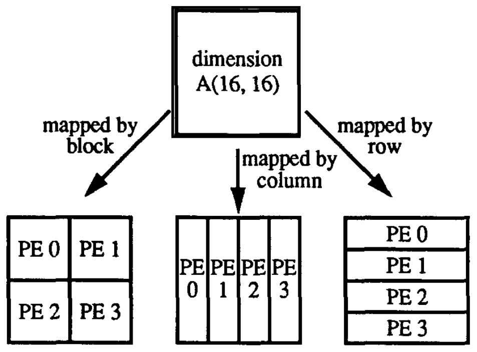


Figure 11: Program Mappings

3.3 Disk Mappings

We describe two variants of the RAID [Patterson 88, Salem 86] technology to illustrate the range of features available in parallel secondary storage. RAID, which stands for Redundant Arrays of Inexpensive Disks, is a technology for making many disks of the sort found on personal computers have performance commensurate with mainframe computers. While the RAID technology appears not to have been intended for massively parallel computers, the extension is obvious.

Disk striping, corresponding to RAID level 0, uses a number of logical disk drives to increase performance, as illustrated in figure 12. In addition, each logical disk drive may have a chain of several disk drives, but the chaining serves to increase capacity without a major increase in performance. Data is stored on the disk array by dividing the data stream into blocks which are mapped in a round-robin manner to the logical drives. Performance increases for large transfers because all the logical drives are transferring data simultaneously. Transaction processing applications, which do small randomly placed I/O operations, observe performance increases because the drives seek simultaneously. The disadvantage of this form of disk striping is that failure of any disk drive will cause loss of data.

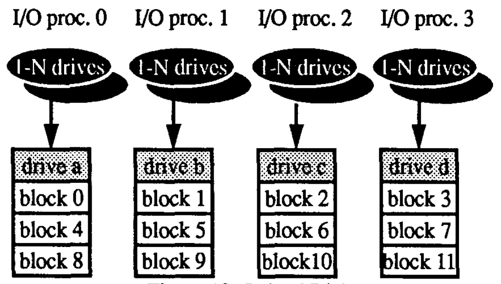


Figure 12: Striped Disk

Disk striping with parity blocks, corresponding to RAID level 5, allows continued operation after a disk failure, as illustrated in figure 13. With this technology, the number of logical drives is one greater than the striping factor. A parity block is generated in each row of blocks, but with the parity block shifting positions on each row. Even with one disk drive failure, the data can still be recovered by computation on the parity blocks. Furthermore, since the parity blocks are accessed more often than data blocks, the even distribution of the parity blocks over the drives levels performance. This technology, while complex, may offer a long-term solution to high capacity, large volume, data storage.

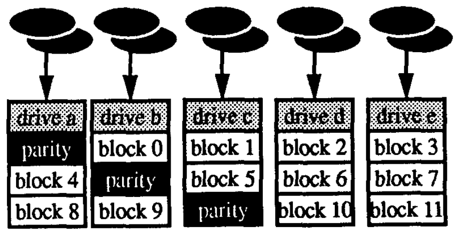


Figure 13: Striped Disk with Parity Blocks

3.4 Fast Network (HiPPI) Hardware

As the computation and I/O capabilities of massively parallel machines scale up with advancing technology, network bandwidths will scale up also. nCUBE has an announced HiPPI network interface which employs the scalable architecture illustrated in figure 14. Like the nGRAPHICS interface, the HiPPI interface interfaces to 128 processing elements in the computational array. The 16 I/O processors on the HiPPI board use video ram chips as main memory. Video ram chips are dual ported devices: one port is standard random access memory and the other port can shift the contents of the memory at very high speed. DMA hardware interfaces between the video ram and the HiPPI physical transport directly. Good mappings for the HiPPI interface distribute the data to the I/O processors in moderate sized blocks. In order to effectively use bandwidth to the 16 I/O processors, messages must be broken into at least 16 pieces. If block sizes are too small, however, the command sequence to the chained DMA hardware will take too long to setup. The I/O system being described applies to this HiPPI interface also.

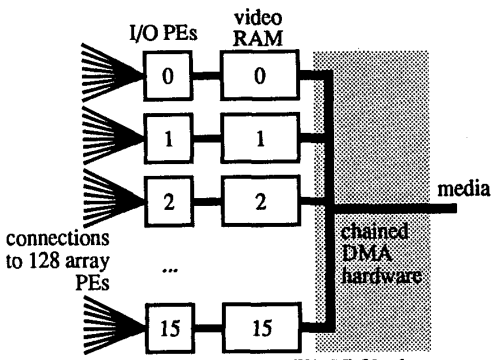
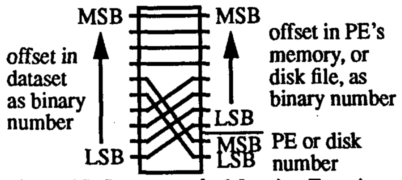


Figure 14: Fast Network (HiPPI) Hardware

4 Mapping Functions

The key to implementing this I/O system is the mapping functions [Chen 88]. On one hand, they must be flexible enough to represent all the mappings described in previous sections. On the other hand, they must be easily enough manipulated that system software can compute $M1^{-1} \circ M2$ and then transfer data efficiently using the composite mapping.



MSB
MSB
offset in dataset as binary number
offset in PE's memory, or disk file, as binary number
LSB
LSB
MSB
LSB
PE or disk number

Figure 15: Structure of a Mapping Function

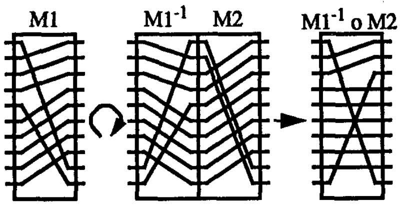


Figure 16: Inversion and Functional Composition

Mapping functions describe the mapping from the abstract dataset to byte positions in the distributed program or I/O device. The input is the index of a byte in the abstract dataset. The outputs are the unit number, being either a processing element number or a I/O device number, and the index of the byte in the unit.

Figure 15 illustrates the particular class of functions proposed as mapping functions. The functions operate by permuting the bits of their arguments. Specifically, the input to the function is represented as a series of bit values, applied from the left. The output of the function appears on the right in two fields. The unit number appears as the bits of lowest significance on the right side. The number of bits varies depending on the number of dimensions of the subcube or the number of I/O units. The remaining bits form the index into the dataset on the specified unit.

The necessary manipulations of these functions are straightforward, as illustrated in figure 16. The inverse of a permutation is easily obtained by reflecting the permutation about a vertical axis, as illustrated by M1 and $M1^{-1}$. Composition of two functions is done by tracing each arc from one end to the other, as illustrated by $M1^{-1}$ and M2 being composed to form $M1^{-1} \circ M2$.

As an example of the use of mapping functions, consider mapping functions for the program mappings discussed earlier. The dataset in the mind of the programmer is a 16x16 array of floating point numbers which are to be mapped by block, column, and row to the memories of four processing elements.

Figure 17 attaches a meaning to the bits in the binary representation of the index of each byte in the dataset. The dataset consists of 1024 bytes, consisting of 256 groups of 4 bytes, each group of 4 bytes representing a floating point number. It should be clear, therefore, that the least significant two bits of the binary representation of the index identify the byte within a floating point number.

The first 16 floats (64 bytes) in the dataset represent the first column of the matrix (given Fortran array ordering). Since the elements of a column have the same column index and varying row index, the next four bits in the binary representation represent the row number where such a byte would be sent. The remaining four

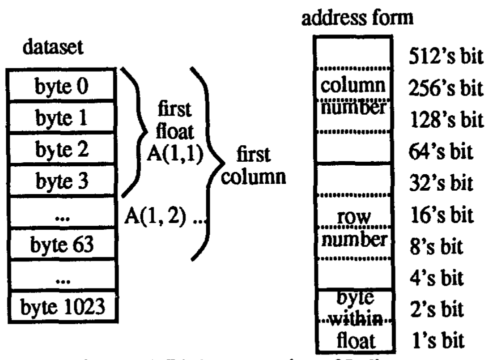


Figure 17: Bit Interpretation of Indices

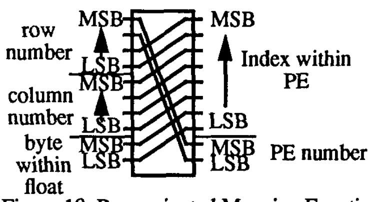


Figure 18: Row-oriented Mapping Function

bits in the binary representation of the index therefore represent the column number.

Figure 18 illustrates the mapping function where entire rows appear on a processing element. In this case, the factor determining which processing element a byte is mapped to is the row number. More specifically, since the columns are mapped in groups, with only one group mapped to a processing element, it is the most significant two bits of the row number that determine the processing element.

Figure 19 illustrates the mapping function where the matrix is mapped in blocks. Which processing element a byte is mapped to depends on both the row and column of that byte. More specifically, since the blocks are eight entries on a side and their are two blocks in each dimension, the highest order bit of the row and column indices determine the processing element number. The permutation retains the Fortran-imposed order of the remaining bits.

Consider the mapping function where entire columns appear on a processing element. Using the same reasoning as before, the highest two bits of the column number determine which processing element a byte is mapped to, with the order of the other bits being unchanged.

This class of mapping functions can describe many popular mappings, but there are limitations. The functions can describe a hierarchy of round-robin blocked mappings. This includes the matrix distributions as well as "disk striping" mappings. Furthermore, while the mapping functions do not support RAID parity units and Gray code orderings, these can be accommodated as an additional functional level.

The limitation of this class of mappings is that all sizes must be powers-of-2. While block sizes on disks and the sizes of "subcubes" are typically powers-of-2, the number of disk drives in a system often is not. We regard this limitation as significant and expect to enhance the class of mapping functions eventually.

5 Low-Level I/O

A new low-level I/O system has been added to support parallel I/O. The new system, which is compatible with the old, not only supports the functions described so far but adds value in the areas of Unix compatibility [AT&T 90].

The old low-level I/O system had two independent types of I/O from the processing elements. I/O to the filesystem was handled by Unix I/O calls. These calls included open, close, read, write, and acted on file descriptors. Message-based I/O for inter-

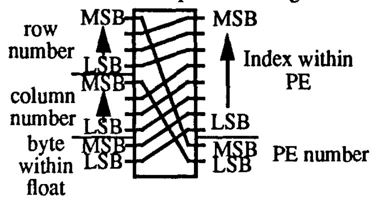
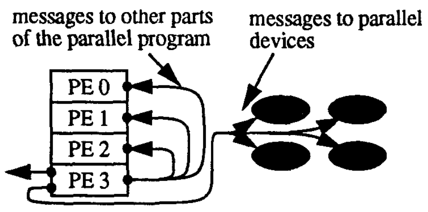


Figure 19: Block-oriented Mapping Function



messages to other parts of the parallel program
messages to parallel devices
PE 0
PE 1
PE 2
PE 3

Figure 20: Merged Message and File I/O

processor communications was handled by nCUBE proprietary system calls such as nread, nwrite, and ntest. The nCUBE proprietary calls, which do not use a file descriptor, do I/O either to other processing elements in the subcube, or to I/O processors.

The old I/O system has been extended as illustrated in figure 20. First, it is now possible to send messages in different addressing domains. Specifically, messages can still be sent to other processing elements in the subcube, in which case addresses 0..N-1 represent the processing elements in the subcube. In addition, messages can be sent to the units of a parallel device, in which case addresses 0..N-1 could represent the different disk drives of a parallel disk. Secondly, messages can be sent via file descriptors. This means that file I/O, which is directed at file descriptors, can be translated into message I/O directed at the different units of the parallel device.

These enhancements have caused a change in roles of proprietary and Unix communications primitives, as illustrated in figure 21. In the old system, the communications kernel (called Vertex) supported the proprietary primitives directly, but had compatibility libraries for Unix primitives and other primitives (such as reactive kernel [Seitz 88]). In the new system, Unix primitives become the new standard interface, and compatibility libraries exist for the proprietary and other primitives.

The new system is an adaptation of Unix networking. In Unix, a program may have a network attached to a file descriptor. An addressing domain will be associated with the network which associates a unique address with the file descriptors of all the other programs which are likewise attached to the network. To send a message, one uses the call putmsg(fd, data, address). The argument fd is the file descriptor of the "connectionless transport endpoint" which attaches the network to the program. The data and address arguments convey the data to be sent and the address it is to be sent to.

The nCUBE system sets up miniature Unix networks to support interprocessor communications and parallel I/O. These networks will be attached to file descriptors of the various processing elements and will support putmsg type calls.

The *nself* feature, illustrated in figure 22, supports interprocessor communications and backwards compatibility. When starting a parallel program, the operating system creates an addressing domain embodying all the processing elements in the parallel program. This addressing domain is attached to a file descriptor called

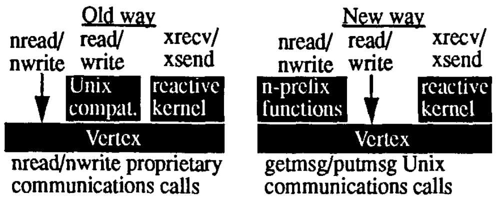


Figure 21: Function Layering

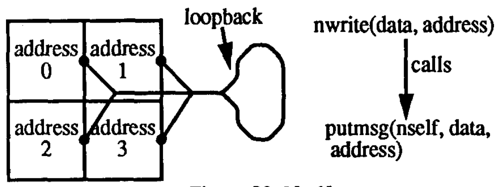


Figure 22: Nself

nself in all the processing elements. Messages are sent within the subcube using nself. The proprietary primitives are then implemented using nself: nwrite(data, address) is translated immediately into putmsg(nself, data, address), using the global variable nself.

Access to parallel I/O devices is easily added at this point, as illustrated in figure 23. For each I/O connection, an Unix-style addressing domain is constructed. The addressing domain includes all units at both the producing and consuming ends of the parallel I/O channel and provides the low-level capability for sending messages among the units.

These low-level I/O capabilities give transparent support for I/O redirection and pipes, which are high-level Unix abstractions. Since parallel I/O channels are associated with file descriptors, and a program inherits the standard input, output, and error file descriptors from the shell when it is started, programs can be run with I/O redirected to parallel files. Parallel pipes are similarly supported.

Tie-in to Mapping Functions

The tie-in between the low-level I/O capabilities and the high-level mapping functions can now be described. Say the user writes one byte to a parallel file. Now, the parallel device appears to the user program like the network in figure 23, rather than a file. Since without an address, one does not know where on a network to send data, regular Unix does not define the behavior of writes to a network. We are therefore free to have writes to networks invoke the mapping function algorithms. Evaluating the $M1^{-1}$ o M2 then yields the destination unit number and a position within this unit for the single byte being written. Following evaluation of the mapping functions, the system code does a putmsg call sending the byte to the appropriate unit.

As in Unix, stream devices and random-access devices are treated differently. For a random-access device, the message specifies the index within the I/O unit, and the byte is written there. For a stream device, the position in the I/O unit can be reconstructed from a count of the bytes sent from a processor to an I/O unit.

Say the user writes many bytes to a large file. The write is broken into blocks where all the bytes in a block are destined for the same I/O unit. While it is beyond the scope of this paper to elaborate, symbolic processing of mapping functions can yield not only where the byte goes, but also the number of consecutive bytes following that will end up contiguously stored on the same media unit. This processing is applied repeatedly to a transfer, shortening

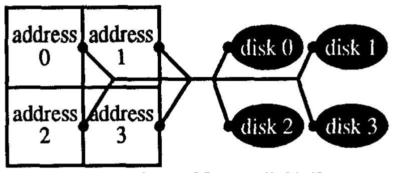


Figure 23: Parallel I/O

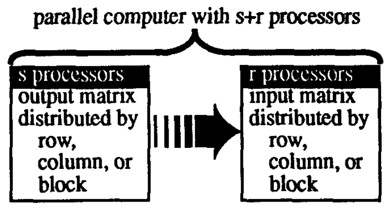

parallel computer with s+r processors
s processors
output matrix distributed by row, column, or block
r processors
input matrix distributed by row, column, or block

Figure 24: Test configuration for parallel pipes

the transfer by the size of the transmitted block each time, until there are no more bytes left.

6 Performance Results

The first test demonstrates heterogeneous programming, or parallel pipes. In this test two parallel programs are running simultaneously in a single nCUBE parallel computer and communicate via a parallel communications path. Each program has a 1024 × 1024 character array distributed in the various combinations of row, column, and block mappings. After specifying the mappings, one program transmits its part of the array the array with a single write and the other receives it with a single read. Several features are demonstrated in addition to scalable performance. When the programs were compiled they knew only about their own data distribution, and not the distribution of the other, hence modularity is demonstrated. In some tests, the distributions differ in the two programs, thereby demonstrating the ability of the I/O system to translate between data distributions.

Figure 24 illustrates the configuration, with tables 1 and 2 illustrating performance figures. All rate and bandwidth measures are in units of MBytes/sec.

Table 1: Performance of parallel pipes

s,r	sender dist.	receiver dist.	rate per node	aggregate bandwidth
1	row	row	2.20 MBps	2.20 MBps
2	row	row	2.18 MBps	4.36 MBps
4	row	row	2.14 MBps	8.56 MBps
8	row	row	2.07 MBps	16.6 MBps
16	row	row	1.94 MBps	31.4 MBps
32	row	row	1.72 MBps	55.0 MBps
64	row	row	1.68 MBps	108.0 MBps

Table 1 contains performance results for various configurations where the number of processors, s, is equal to r, and both the sending and receiving programs use a row distribution. The column containing results for the aggregate bandwidths of each configuration shows that the bandwidths increase almost linearly with the number of allocated processors, demonstrating the scalability of the parallel pipe.

The first row of the table shows a rate per node of 2.20 MBytes/sec. This matches the maximum attainable unidirectional node-to-node transfer rate on an nCUBE computer. Since the selected mapping represents the identity mapping between the nodes of the sending and receiving programs, it is expected that the rate per node would remain constant as the values of s and r are altered. However, the results show that there is a gradual divergence between the observed and expected per node rates. It has been observed that this reduction in the rate per node can be accounted for by assigning a constant overall transmission overhead of approximately 4 milliseconds. Further, this overhead and the resulting reduction in the per node transfer rate is only significant when the size of each node's local data is below 1 Megabyte. For test cases where each node's local data was maintained at greater than or equal to 1 Megabyte, the rate per node remained constant for all values of s and r, and the aggregate bandwidths exhibited perfect linear speedups. The exact contributions to this overhead from the various possible sources are currently being investigated and, due to publication deadlines, is relegated to some future paper.

Table 2 provides results for a parallel pipe using various mapping combinations. The results show the expected reduction in rates for non-identity mappings caused by the increase in overhead related to additional data manipulations. The exact contributions

Table 2: Parallel pipe mapping variations

s,r	sender dist.	receiver dist.	aggregate bandwidth
64	row	row	108.0 MBps
64	column	column	108.0 MBps
64	row	column	2.11 MBps
64	row	block	23.2 MBps
64	block	block	108.0 MBps

to this overhead from the various possible sources are currently being investigated and, due to publication deadlines, is relegated to some future paper.

The second test demonstrates I/O between parallel programs and secondary storage. In this test, a parallel program transfers a 1024 × 1024 character matrix to a parallel disk. After specifying a row mapping, the program writes the matrix to the disks. The data is stored as 1024 character stripes distributed in round robin fashion among the disks.

Figure 25 illustrates the configuration, with table 3 illustrating performance figures.

The results show that the aggregate disk bandwidth increases in proportion to the number of disks. However, it also shows that the limiting factor in the overall bandwidth is determined by the minimum of the bandwidth capacities of the processors and that of the disks. This is evidenced by the fact that doubling the number of nodes for a given set of disks will cause only slight variations in the overall bandwidth.

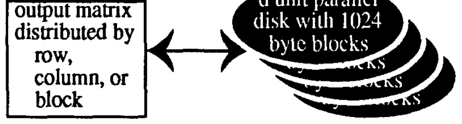

r processors
output matrix distributed by row, column, or block
d unit parallel disk with 1024 byte blocks
Figure 25: Test configuration for parallel disk

Table 3: Performance of parallel disk

s	output dist.	disk units	disk bandwidth
1	row	1	1.01 MBps
64	row	1	1.30 MBps
32	row	2	2.52 MBps
64	row	2	2.49 MBps
32	row	4	4.83 MBps
64	row	4	4.96 MBps

7 Conclusions

We have discussed a long term plan for parallel I/O on massively parallel machines. Some of these features will be included in release 3.0 of the nCUBE system software, although other features will not be included until later.

This system will transfer data in parallel between scalable programs and/or I/O devices with any number of units. We claim to have made I/O as scalable as computing.

Furthermore, I/O and computing are integrated.

The additional features were added in the Unix model, where possible. In other places, the Unix model has been extended in a natural way.

Unix abstractions have also been extended to parallel I/O. I/O redirection (program < file) and pipes (program1 | program2) are supported naturally.

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.

[DeBenedictis 91] E. DeBenedictis, P. Madams, "nCUBE's Parallel I/O with Unix Compatibility," *Proceedings of the Sixth Distributed Memory Computing Conference*, Portland, OR, April, 1991

[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).