I/O for TFLOPS Supercomputers

Erik P. DeBenedictis[†] Stephen C. Johnson[‡]

Abstract

Scalable parallel computers with TFLOPS (Trillion FLoating Point Operations Per Second) performance levels are now under construction. While we believe TFLOPS processor technology is sound, we believe the software and I/O systems surrounding them need improvement. This paper describes our view of a proper system that we built for the nCUBE parallel computer and which is now commercially available.

The distinguishing feature of our system is that scalable parallelism is implicit rather than explicit. We did not base our system on new commands, system calls, or languages. Instead, we extended some aspects of Unix® to add parallelism while keeping these aspects unchanged for nonparallel programs.

The result is a system that lets one use a future TFLOPS parallel computer without knowing parallel programming. As parallel versions of standard compilers arrive, and large data sets get distributed over multiple I/O devices, then standard Unix commands will run arbitrary mixtures of parallel and nonparallel programs and I/O devices. One gets scalable computing and I/O rates whenever a command includes only parallel components.

1 Introduction

1.1 Scalable architecture

A scalable architecture is a shorthand notation for describing a family of computers. FIG. 1 uses the notation to describe the nCUBE's architecture (which is similar to others [4] [5] [8]). The scalable architecture has a Processing Element (PE) repeated *n* times and an I/O device repeated *m* times. Each PE has a CPU and some local memory that only that CPU uses for the kernel of a computation. The use of local memory only ensures that increasing the repetition factor *n* does not degrade performance due to network limitations.

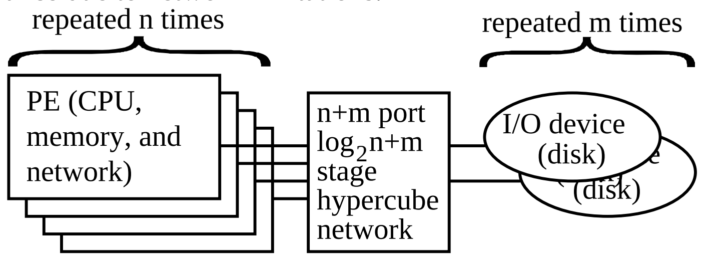

repeated n times
repeated m times
PE (CPU, memory, and network)
n+m port log2n+m stage hypercube network
I/O device (disk)
(disk)

FIG. 1: A scalable architecture

[†] President, Scalable Computing, Redwood City, CA.
[‡] President, Melismatic Software, Palo Alto, CA.

A $\log_2(n+m)$ stage hypercube communication network connects the PE's and I/O devices. The scalable architecture corresponds to the family of computers with different values of *n* and *m*. Leaving the repetition factors unbound allows us to see how performance and other factors vary.

TFLOPS-level computers are very large; those due for delivery about 1995 roughly follow FIG. 1 with *n*=10,000 and 100 MFLOPS per CPU. Furthermore, for each *n*=10,000 computer, there will be many computers built with smaller values of *n*. This means hardware and software will have to work over a four order-of-magnitude size range ($1 \leq n \leq 10{,}000$).

1.2 Scalable I/O

The traditional standard for balanced I/O on any computer calls for the I/O rate (in MBytes/second) to equal the computing rate (in MFLOPS). Others expect this to hold for TFLOPS supercomputers as well [10]. This means a scalable processor requires a scalable I/O system for balance. The natural way to achieve this adds enough ports (*m*) to the network for disk drives, as shown in the right side of FIG. 1. The ratio of *n* and *m* and the relative speed of the processors versus the I/O media determine the I/O balance. Keeping the ratio of *n* and *m* about the same keeps I/O balanced among members of the architecture's family.

Just because many processors and disks are wired together does not mean that an application automatically runs faster. However, algorithms have been developed for many problems that give proportional speedups as *n* and *m* increase [3, 10], especially for large problems. The challenge is to do this within the Unix framework

1.3 Extendibility in Unix

Unix® [9] has some internal interfaces that help make extensions natural. These internal interfaces act as barriers. They limit the effect an extension has on the rest of the system and on applications programs. This section discusses some internal interfaces and sets the stage for the following section that uses them as the basis of parallel extensions.

FIG. 2 shows how the Unix executable file is an internal interface that limits the effects of different programming languages. Compilers specific to each language translate programs into a single executable file type. This insulates the rest of the system from knowledge of a program's source language. By compiling parallel source into the common executable, we block most direct exposure of the user to parallelism. Of course, today's executable files lack features necessary for TFLOPS-level parallel processing. We first discuss how to get TFLOPS-level performance before returning to the issues of source code and executable files.

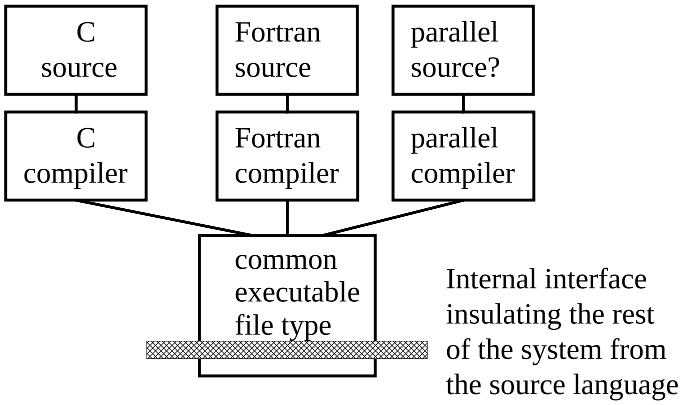

C source
Fortran source
parallel source?
C compiler
Fortran compiler
parallel compiler
common executable file type
Internal interface insulating the rest of the system from the source language

FIG. 2: Multiple languages

FIG. 3 shows the Unix internal interface for connecting application programs to each other and to I/O devices. Unix has a single type of "I/O pipe" that allows a user to connect any program to any I/O device (or any other program). Window-based user interfaces illustrate the value of this internal interface. Hundreds of Unix utility programs were first written decades ago on minicomputers. Their programmers knew only about printing terminals and CRT character displays because there were no windowing systems yet. Since modern windowing systems use the same I/O pipes as the old-style terminals, these programs run today in windows (but still as a text-based applications). Pipes let hundreds of programs get a free (limited) upgrade to a window-based user interface. This accelerated the acceptance of window-based user interfaces by automatically providing hundreds of utility programs.

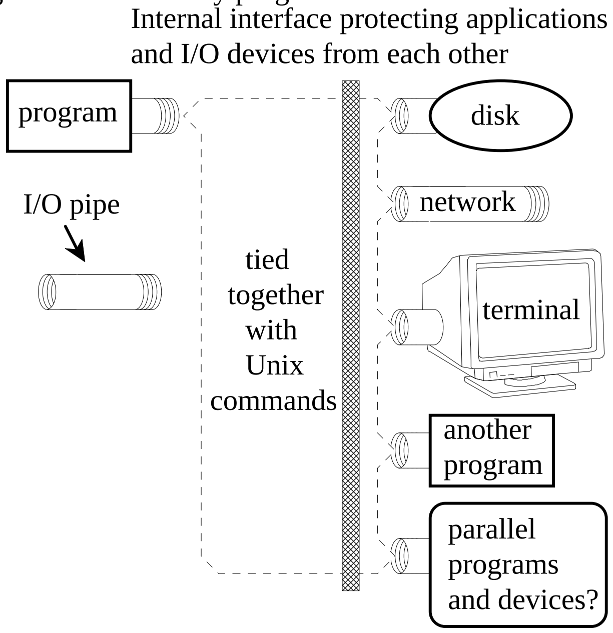

Internal interface protecting applications and I/O devices from each other
program
I/O pipe
tied together with Unix commands
disk
network
terminal
another program
parallel programs and devices?

FIG. 3: Data pipes in Unix

If parallel programs and parallel I/O use an extension of the standard Unix I/O pipe, we can preserve this interface. This will allow existing Unix programs to run on new parallel computers. Like the window-based user interfaces, this will provide a limited upgrade. As more utilities and applications are written to use the parallel extensions, performance will improve but the user interface will be unchanged. The next section discusses this parallel extension to I/O pipes.

2 Parallel Extensions for I/O Pipes

2.1 Current Unix I/O Pipes

The current Unix internal interface lets programmers believe all I/O is in the form of a byte stream. This means programs do not need to know the physical representation of files and output devices (e. g. tape and disk blocking, printer carriage control). Conversely, I/O devices do not need to know how a program interprets data (e. g. text or binary data). Instead, programmers imagine and manipulate data in the most natural way. Text on a screen (illustrated in FIG. 4) or images (illustrated in FIG. 5) are common views. While writing programs that print lines of text, programmers think of adding lines to the end of a screen. This view holds even if the program is sending output to a disk.

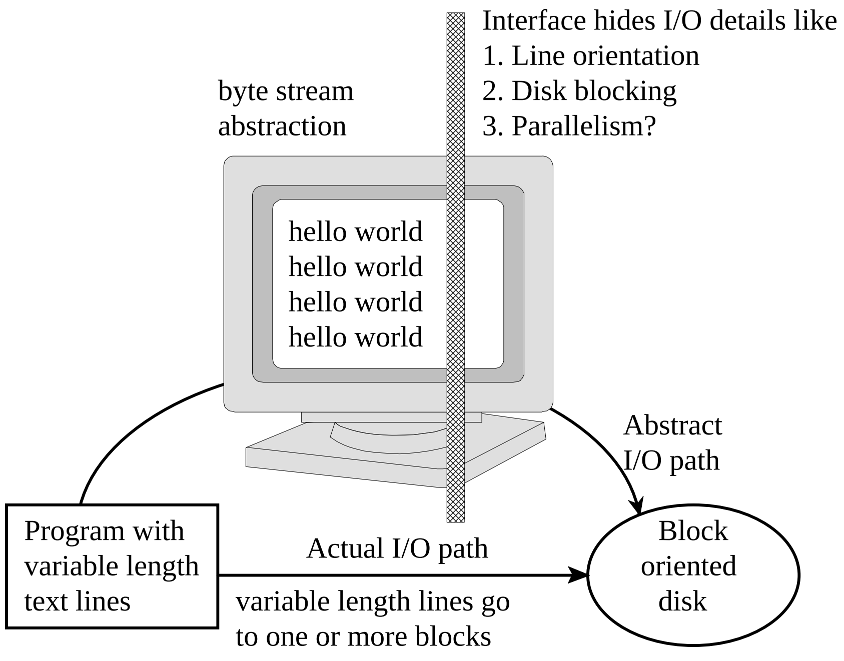


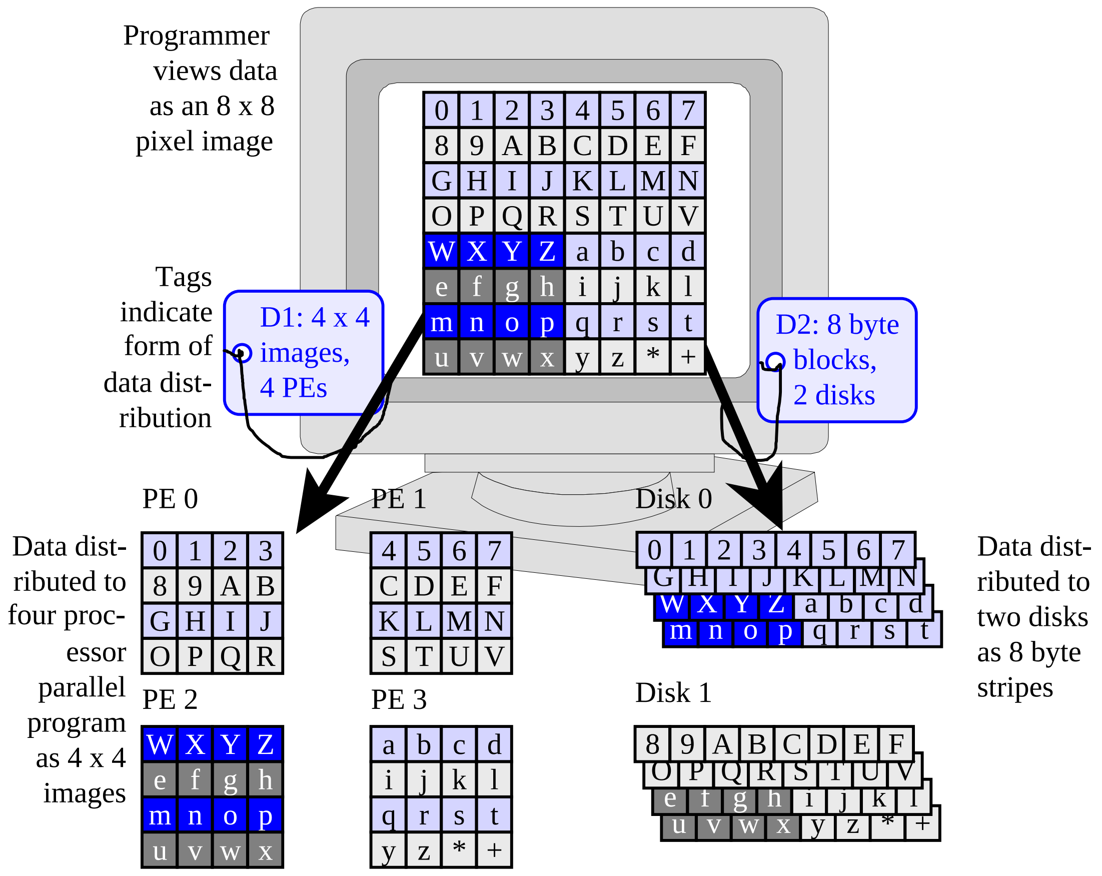
FIG. 4: Byte stream abstraction

Byte streams are abstractions because the system does not really do I/O that way. Instead, data is moved from a program to a device in a single composite step as shown in FIG. 4. Writing text to a disk, for example, may change the end of one disk block, go to the free list for another block, add that block to the file description in the directory, and change the beginning of the new block.

2.2 Unix extensions for data distribution

This section shows specific extensions to the Unix I/O pipe interface to support data distributions.

FIG. 5 is an example of a parallel program in its execution environment. The program is a parallel image processing application that writes an image to a parallel disk. The parallel program, shown on the bottom left, uses the popular Single Program Multiple Data (SPMD) [3] execution paradigm. In such a program, each of the four Processing Elements (PEs) illustrated runs the same program. Furthermore, the PEs execute similar sequences of instructions. Specifically, all PEs execute system calls in the same sequence but on different data.



Programmer views data as an 8 x 8 pixel image
0 1 2 3 4 5 6 7
8 9 A B C D E F
G H I J K L M N
O P Q R S T U V
W X Y Z a b c d
e f g h i j k l
m n o p q r s t
u v w x y z * +
Tags indicate form of data distribution
D1: 4 x 4 images, 4 PEs
D2: 8 byte blocks, 2 disks
PE 0
PE 1
PE 2
PE 3
Disk 0
Disk 1
Data distributed to four processor parallel program as 4 x 4 images
Data distributed to two disks as 8 byte stripes

FIG. 5: Parallel system example

The programmer views the data as the 8 × 8 image, as shown at the top of FIG. 5. We have labeled the pixels with characters to show correspondences. Image processing applications commonly divide data into roughly square 2-d regions and distribute these regions to the processors viewed as a 2-d array. The arrow from the image to the processor array represents this distribution. We put a tag on this arrow with the words "4 × 4 images, 4 PEs" to document this distribution.

We show normal striping [7] for the two-disk storage system on the right. The storage system views data as a 1-d sequence of 8 byte blocks, with the blocks distributed round-robin to the disk drives. The tag on the right side documents this distribution.

As with the current Unix systems, I/O is not really done this way. To achieve scalable transfer rates, the operating system sends data directly from the processors to the disks, transmitting, blocking and writing the data as needed — the next section gives the details. Throughout FIG. 5, we shaded the pixels in PE 2 and the first stripe. This shows how output from PE 2 creates a complex pattern on the disk. Specifically, output from PE 2 goes to disk 0 then 1 and then repeats disks 0 and 1. Furthermore, PE 2's data in each stripe neither starts nor ends either stripe and is not even contiguous within the stripe. Complex patterns like these often result from composing two data distributions.

The tags D1 and D2 *are* the details of parallel I/O. We can preserve the byte stream abstraction by avoiding contamination of the program and I/O devices with each other's tags.

2.3 Scalable pipes

FIG. 6 shows a scalable algorithm suitable for an I/O pipe. Compare this to the processors, network, and I/O units of the scalable architecture shown in FIG. 1. We show data switches on the left and right sides of the figure. These switches get put in each processor and I/O device. As the data passes through the left hand switches, individual bytes or blocks are routed to the proper channel on the right. Sometimes, we need the switches on the right. They order the arriving data into a single stream. The network in the middle conveys the data blocks to the proper unit.

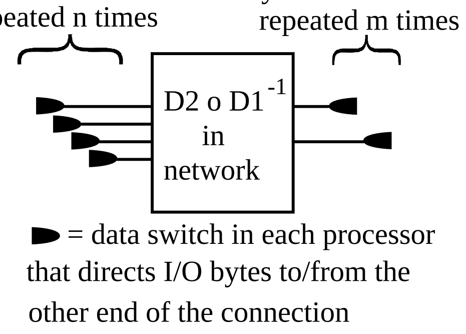

repeated n times
repeated m times
D2 o D1^{-1}
in
network
= data switch in each processor that directs I/O bytes to/from the other end of the connection

FIG. 6: Parallel algorithm to convert distributions

The algorithm in FIG. 6 is both theoretically and practically scalable. As detailed in Ref. 2, the data switching algorithms are theoretically constant-time and fast in practice. Since they are independent and executed in parallel, they do not impair scalability. Coupled with a scalable network, the overall algorithm would be scalable. Now, some would argue that there are no general purpose networks that are truly scalable. These arguments are most convincing for shared-memory networks where low network latency is important. They are less applicable to message-passing networks. Since the algorithm uses messages and does not depend on low network latency, the network does not impair scalability either.

The algorithm in FIG. 6 is the new feature needed in an I/O pipe to make the bandwidth scalable. The overall effect of the algorithm is to redistribute data from the left hand distribution D1 to the right hand distribution D2. This is D1$^{-1}\circ$D2 in functional notation. While algorithms for the data switching are straightforward, the operating system must know D1 and D2 to configure the algorithm. The next section describes how the system gets D1 and D2 from executing programs through system calls. Also, we give nonparallel programs and devices a default tag that says "no distribution, one processor." This makes the algorithm handle connections between parallel and nonparallel programs and devices automatically.

2.4 Interface to emerging parallel languages

FIG. 7 illustrates an interface with parallel compilers. We illustrate the array "dimension a(8, 8)" from the example at the top center of the figure. Parallel compilers select data distributions like D1 and use them for distributing arrays and calculations on them. Current parallel compilers then discard this information, making it unavailable to other parts of the system. We suggest enhancing parallel compilers to put this information into the executable file. We also suggest that compilers generate system calls to deliver this information to the operating system before doing I/O. This gives the operating system the information necessary to use the pipe extension described above.

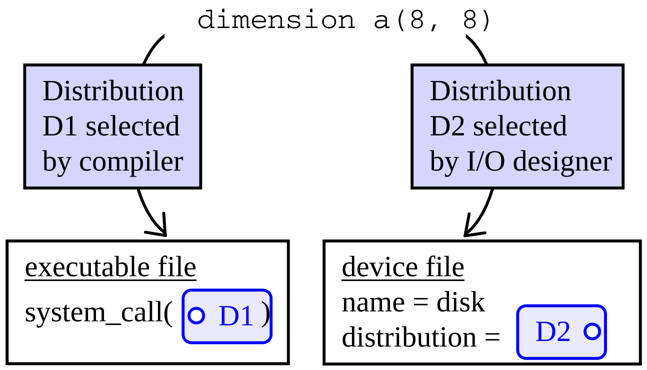


FIG. 7: Flow of data distribution information

There is a similar situation with data distributions in I/O devices. The system administrator selects D2 based on the number of disks on the system and other requirements. These distributions typically distribute a byte stream using striping [7] and RAID [6]. In current I/O device technology, data distribution is entirely the responsibility of the storage subsystem and is unavailable elsewhere. This is equivalent to the parallel compiler discarding data distribution information. Making D2 available in the device driver enables scalable I/O to the device.

A system call delivering distribution information to the operating system is the primary extension needed to executable files. Since the default is "no distribution, one processor," the new system call does not affect existing programs. Parallel programs make the new system call and thus enable parallel I/O.

3 Scalable I/O hardware

While TFLOPS computers are under construction, the fastest networks at the same stage of development are under 10 GFLOPS. This is under 1% of the bandwidth required for a traditional I/O balance. By using the software described earlier and the "stream combiner" architecture in FIG. 8, one could build a scalable network that can be driven by any Unix program. One could then scale the network to a terabyte/second to balance a computer scaled to TFLOPS.

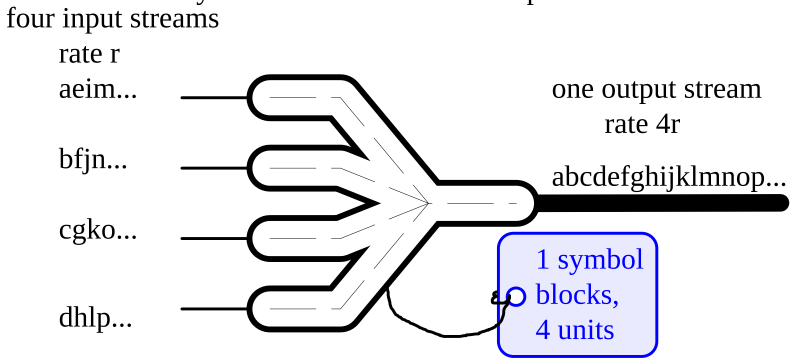


FIG. 8: Stream combiner

The stream combiner shown in FIG. 8 lets the software just described drive any fast medium. This device works like the section of roadway between a toll booth and a bridge in a big city. Since toll booths are slower than traffic lanes, a two-lane bridge may have a dozen toll booths. Cars emerge slowly from the toll booths on a dozen lanes of roadway. As these lanes quickly merge into two lanes, the traffic speeds up and the cars become closer together. This lets the bridge run at full capacity though individual toll booths are slow. The stream combiner illustrated works similarly to four toll booths and a one lane bridge. Where cars merge randomly on a roadway, the stream combiner merges the data according to the specific pattern shown.

The merging pattern shown corresponds to a 1-d data distribution as described earlier. This lets one create a data distribution tag for the stream combiner. If one then treats stream combiner inputs as the units of a parallel entity, the software described earlier will work.

One must tailor a stream combiner to each specific medium. An engineer must pick a size for the symbols in FIG. 8 and understand the flow control requirements of the medium, for example.

4 Work Performed

We built system software for the nCUBE parallel computer around the ideas in this paper. This software became version 3.0 of nCUBE's system software, and is commercially available.

We ran many performance trials, including program-to-program, program-to-disk, and program-to-device (video display) I/O. We also recompiled Unix programs, like *tar*, and put them on the system release tape as system utilities. We detail these results in ref. [2] and the nCUBE technical documentation.

5 Conclusions

We conclude by describing the logical result of consistently applying this design approach to scalable computers. The high level block diagram of a future massively parallel computer may be the same as a contemporary computer. There will be scalable components inside the boxes in its block diagram, however. The processor section will consist of a processing element replicated *n* times. The replication factor *n* will increase the performance by the same factor. Furthermore, one can have any value of *n* provided one can afford the resulting computer. The secondary storage, I/O subsystem, and systems software will have a similar design and properties.

Software and programming for such a computer may bear striking resemblance to that for today's computers. The user who does not program but deals solely with applications may see no difference. This user will see a computer with a familiar user interface and supporting the same abstractions as a contemporary computing environment. Today's programs also will run on any single processor.

The programmer will see a similar environment, but programmed with enhanced languages. Programmers will write in existing languages enhanced for data parallelism. The compilers for these languages will produce executable files that are interchangeable with today's Unix files.

When parallel programs run in combination with parallel I/O devices, TFLOPS computing levels occur.

References

1. E. DeBenedictis and P. Madams, "nCUBE's Parallel I/O with Unix Compatibility," in *Proceedings of the Sixth Distributed Memory Computing Conference*, May 1991.
2. E. DeBenedictis and M. del Rosario, "Modular Scalable I/O," Journal of Parallel and Distributed Computing, January 1993 (to appear).
3. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker, "Solving Problems on Concurrent processors," Prentice-Hall 1988.
4. Intel Corporation, technical documentation on the Paragon parallel computer.
5. MasPar Corporation, technical documentation on the MP-1 parallel computer.
6. D. Patterson, G. Gibson, R. Katz, "A Case for Redundant Arrays of Inexpensive Disks," in *Proceedings of the 1988 ACM SIGMOD Conference on Management of Data*, Chicago, IL, June, 1988, pp. 109-116.
7. K. Salem, H. Garcia-Molina, "Disk Striping," *IEEE 1986 International Conference on Data Engineering*, 1986, pp. 336-342.
8. Thinking Machines Corporation, "The Connection Machine CM5 Technical Summary," 1991.
9. Unix System Laboratories, Inc., "Unix System V Documentation," Prentice-Hall, 1992.
10. J. Van Zandt, "Parallel Processing in Information Systems," Wiley, 1992.