Extending Unix for Scalable Computing

Erik P. DeBenedictis and Stephen C. Johnson
Scalable Computing

Will tomorrow's parallel computer function like today's workstation, only faster? Applying the Unix operating system to a parallel environment may be the answer.

Because it retrieves all instructions and data from a single memory, the highly prevalent von Neumann computer architecture has a fundamental speed limit. The speed of light limits the rate data can move from a single memory, thereby limiting the speed of the computer. Attempts to push the physical limits in supercomputers reduce their efficiency; in fact, this is why a personal computer provides more performance for the dollar than a supercomputer. Nonetheless, continued progress in science and engineering requires faster supercomputers. Consequently, researchers have long investigated non-von Neumann computer architectures. The scalable multicomputer architecture, which uses many microprocessors together to solve a single problem, promises to be a solution.[1] In fact, scalable parallel computers that run at tera (10^{12}) floating-point operations per second are now under construction.

While Tflops processor technology is well known, the scalable operating and I/O system technology necessary for those speeds is not. This article describes how Unix can be extended to scalable computing to permit Tflops speeds. We designed this technology into the system software of the Ncube-2, the predecessor to Ncube's announced Tflops parallel computer. We describe the system specifically and provide some performance numbers.

Parallel programming. Advancing technology makes parallel processing less explicit. Newer systems avoid new commands, system calls, or languages. Instead, parallel extensions are placed into existing software without disrupting nonparallel programs. The resulting systems will let you use a Tflops parallel computer without knowing parallel programming as emerging parallel versions of standard languages move these operations to the compiler. In addition, parallel storage systems automatically distribute large data sets over multiple I/O devices. Standard Unix commands now run arbitrary mixtures of parallel and nonparallel programs and I/O devices. However, users will get scalable computing and I/O rates from commands that include only parallel components.

Scalable architectures. Several vendors[2-4] sell machines that have the scalable architecture shown in Figure 1. The architecture has n processing elements (PEs) and m I/O media. A log $(n + m)$ stage communications network[5] allows the PEs and the I/O devices to communicate (one vendor[2] uses an $(n + m) * $ stage net-

0018-9162/93/1100-0043$03.00 © 1993 IEEE

work). Each PE contains a microprocessor and some local memory. This architecture avoids the von Neumann bottleneck because the total memory-access rate grows with the number of processors. Other bottlenecks are avoided because the communication network capacity grows with the number of PEs and I/O devices.

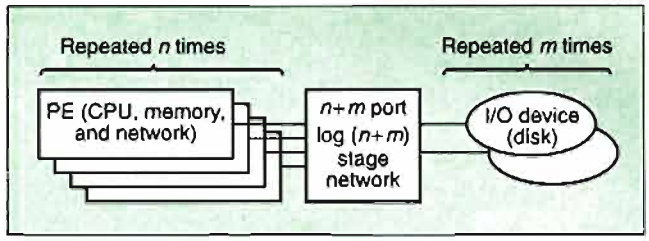

Figure 1. Scalable architecture.

The Tflops computers due for delivery in 1995 approximate Figure 1, with n = 10,000 and 100 million floating-point operations per second (Mflops) per CPU. (See the sidebar "Message passing and shared memory.")

The traditional standard for balanced I/O requires the I/O rate (in megabytes per second) to equal the computing rate (in Mflops). Some researchers expect this to hold true for Tflops supercomputers.[6] This implies that a scalable processor requires a scalable I/O system for balance. The ratio of n to m — and the relative speeds of the PEs and I/O media — determine the I/O balance. Keeping the ratio of n to m about the same keeps I/O balanced as the computers scale in size.

Programming. The data-parallel method dominates programming for scalable computers. Originally, programmers wrote data-parallel programs by hand.[7] At that time, programmers also wrote a single program for all n PEs. They distributed large arrays among the PEs, leaving $1/n$th of the data and computations on each one. More recent compilers can distribute arrays themselves.[8] Extensions to conventional programming languages let the programmer declare forms of parallelism. The compilers then divide the data and computations among the various processors. A similar approach applies to I/O. The programmer specifies certain attributes of the secondary storage system, and the compiler optimizes the data distribution across the PEs and I/O media simultaneously.

So what keeps parallel supercomputers out of the mainstream?

System software. It takes more than power to produce commercially successful computers. The power must arrive in familiar, easy-to-use, portable packages. These packages include an Application Program Interface (API), the most widely known being DOS, Windows, Macintosh, and Unix. These packages offer functional capabilities and promise future continuity, running on many generations of hardware. In recent years, however, packages that deliver more computing power have become as important as the hardware itself. Unfortunately, existing packages for nonparallel computers need reworking to deliver scalable computing rates. This article describes reworking Unix for this purpose.

The notion of the API is central to this reworking. Programmers code applications to run with an operating system, not directly on the computer's hardware. The API defines the interface between applications and the operating system. When compiled and linked, applications can run on any computer and with any I/O devices that support the API. This gives hardware vendors a ready market for new machines as long as they support the API on successive generations of hardware. Similarly, software vendors code to the API, not to the hardware, to ensure that their software will run on future computers.

The APIs developed for nonparallel computers are inadequate for parallel processing. Specifically, existing multitasking API implementations lose efficiency when values of n and m exceed several dozen. More seriously, they restrict I/O to a single channel at a

Message passing and shared memory

The *message-passing* type of scalable computer uses the network to send messages between processors. These computers require the distribution of data and computations.[1] This means that large arrays become distributed among the memories of the processors. Programmers then try to assign each computation to the processor holding the data it acts on. Inevitably, however, some computations will use data stored in several memories. In this case, messages convey data between processors.

The *shared-memory* type of scalable computer can route memory accesses from one processor through the network to memory in another processor. The programmer still distributes data arrays and computations on shared-memory computers, but the distributions do not have to match so precisely. This is because a computation on any processor can access data in any other processor's memory. Announced products of this type have performance below the Tflops level, however, with network performance being the limiting factor.

While research continues into faster networks for shared memory, there is another approach as well. Newer compilers control the distribution of data arrays and computations for the inner loops of computations.[2] This reduces the number of memory accesses that occur over the network, reducing network performance requirements. This improves performance but blurs the distinction between the shared-memory and message-passing approaches.

Even though matching data distribution with I/O is a main concern in this article, it does not apply directly to shared-memory computers. However, matching distributions lessens network performance requirements for I/O, as it does for computation. This might make existing networks adequate. We suggest, therefore, that this article applies indirectly to shared-memory computers.

References

1. G. Fox et al., *Solving Problems on Concurrent Processors*, Prentice Hall, New York, 1988.

2. D. Loveman, "High-Performance Fortran," *Parallel & Distributed Technology*, Vol. 1, No. 1, Feb. 1993, pp. 25-42.

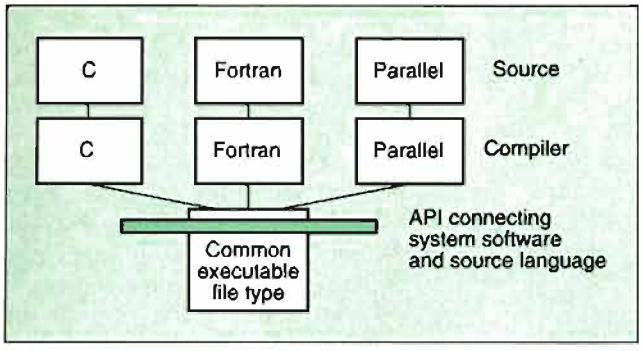

Figure 2. Multiple languages.

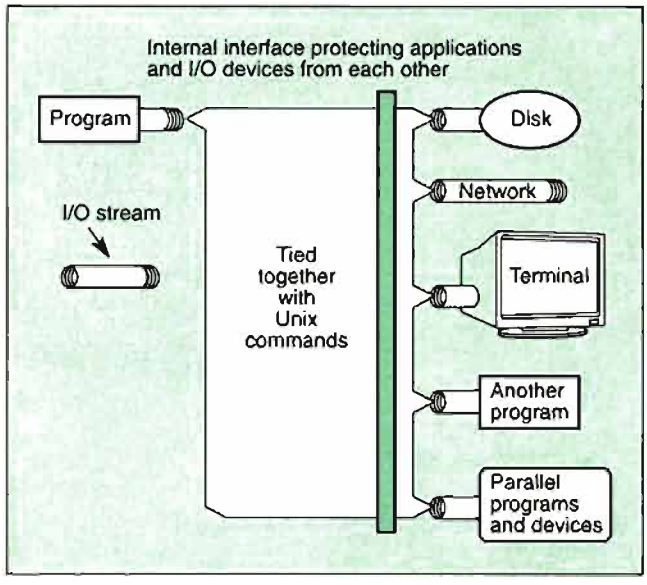

Figure 3. I/O streams in Unix.

time. In contrast, some scalable algorithms run on thousands of processors and use hundreds of disks. Simple extensions of the Unix API can put these algorithms into an API that is scalable by design.

The bulk of this article concerns deriving a scalable API compatible with Unix. We chose Unix because it is the most popular operating system among current users of parallel computers, although the basic ideas can apply more widely.

Extending Unix

Figure 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 knowing a program's source language. For these reasons, parallel source is compiled into the common executable.

Figure 3 shows the Unix abstraction for connecting application programs to one another and to I/O devices. Unix has several features for managing I/O that we collectively call I/O streams. One such feature, device-independent I/O, lets a program perform I/O to a variety of I/O devices without a change in source code. All programs and I/O devices communicate through a common interface, or I/O stream. A second feature — variously called pipes, sockets, and fifos — allows two programs to communicate directly by using the I/O streams model. 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 hard-copy terminals and CRT character displays because there were no windowing systems then. Since old-style terminals performed I/O using an I/O stream, windowing systems were designed to emulate the older terminals and allow older programs to run today in windows. I/O streams allowed hundreds of programs to upgrade to a window-based user interface. This accelerated the acceptance of window-based user interfaces by automatically providing hundreds of utility programs.

We can accelerate the acceptance of parallel processing if parallel programs and parallel I/O use an extension of the standard Unix I/O stream, allowing existing Unix programs to run on new parallel computers. Like the window-based user interfaces, this will provide a limited upgrade.

The Unix API allows programmers to treat all I/O as a byte stream. This means programs do not need to know the physical representation of files and output devices (such as tape and disk blocking and printer carriage control). Conversely, I/O devices do not need to know how a program interprets data (such as text or binary data). Instead, programmers imagine and manipulate data in the most natural way. On-screen text (illustrated in Figure 4) or images (illustrated in Figure 5) are

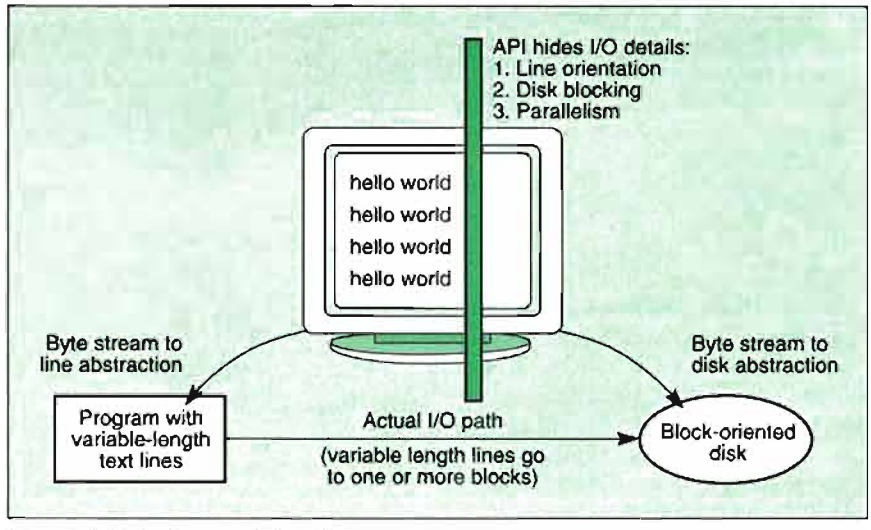

Figure 4. Byte stream abstraction.

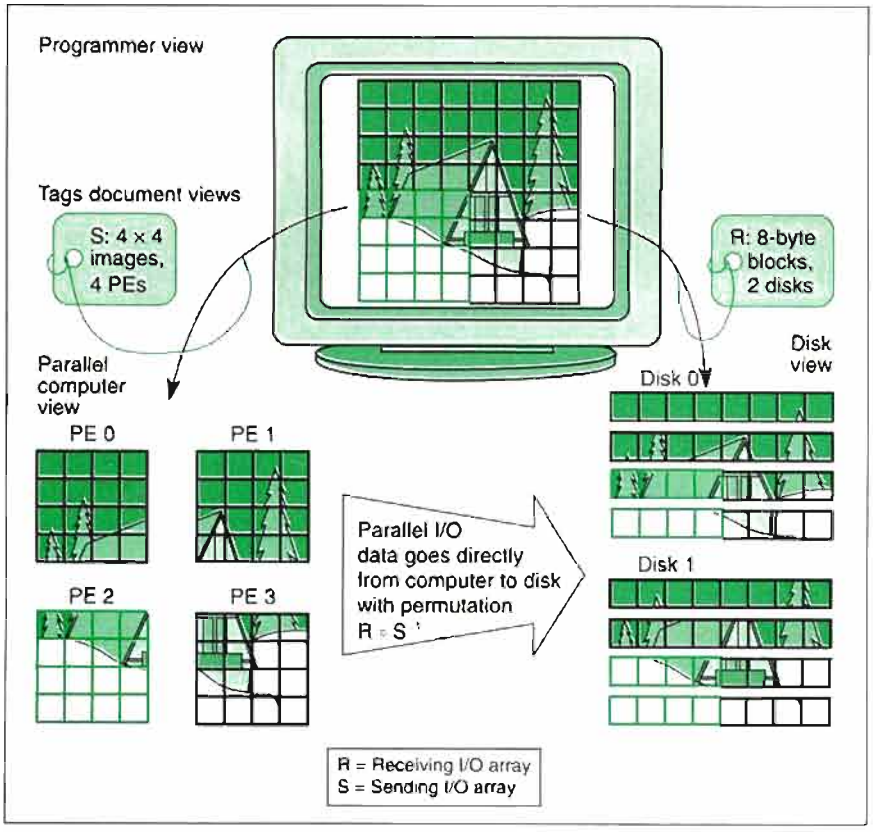

Figure 5. Parallel system example.

common models. While writing programs that print lines of text, programmers imagine adding lines to the bottom of a document. This model holds even when the program is outputting to disk.

Byte streams are abstractions because the system does not really perform I/O that way. Instead, data actually moves from a program to a device in one step as shown in Figure 4. Writing a line of text to disk, for example, can change the end of one disk block, get an unused disk block from the "free list," add that block to the file description in the directory, and change the beginning of the new block. (See the sidebar "Other approaches to parallel operating systems.")

Unix extensions for data distribution. Figure 5 also shows how data flows in a prototypical parallel I/O activity. In the following sections, we package these dataflows into the Unix model. Figure 5 is an example of a parallel program in its execution environment. The program is a parallel image-processing application that writes the image to a parallel disk. The parallel program on the bottom left uses a programming paradigm variously called single program, multiple data (SPMD); loosely synchronous;[7] or data parallel.[8] In such a program, each of four PEs runs the same program. Furthermore, the PEs execute similar sequences of instructions, but on different data. All PEs execute system calls in the same sequence.

The programmer views the data as an 8 × 8 image, as shown at the top of Figure 5. (We labeled each pixel with a small image so that correspondences are easy to see.) Image processing applications commonly divide data into roughly square two-dimensional regions and distribute these regions to the processors, where they are viewed as a 2D array. This distribution excels for image-processing algorithms that access the neighbors of each pixel. The arrow from the image to the sending (S) processor array represents this distribution. We placed a tag on this arrow with the words "S: 4 × 4 images, 4 PEs" to document this first distribution.

The figure shows normal striping for the two-disk storage system on the right. The storage system views data as a 1D sequence of 8-byte blocks, with the blocks distributed round-robin to the disk drives. This kind of distribution gives good results over the general mix of I/O accesses in a computer system. The arrow from the image to the receiving (R) I/O array and the arrow's tag document this second distribution.

As with nonparallel Unix I/O, parallel I/O is not actually done in this way.

Other approaches to parallel operating systems

The most widely used approach to parallel operating systems is multithreading.[1] This approach, now used on tens of thousands of symmetric multiprocessors, lets each Unix process have multiple execution threads, as shown in Figure A. Each thread has a separate CPU, but all threads in a process share the same memory.

During a computation phase, calculations are performed by all CPUs in parallel. In early versions of this technology, only one CPU could perform a system call at once. This was equivalent to having an "I/O position" that only one CPU could occupy. Since the outside world sees a process only through system calls, such a process looks just like the single CPU in the I/O station. More recent operating systems allow multiple processors to perform certain system calls at the same time. This technology requires identifying all interactions between system calls and dealing with each case, so that the result looks like a single CPU. The industry is still dealing with the complexity explosion that results from this approach. Commercial multiprocessors limit the number of CPUs to around 32, although some research computers have 256 processors.

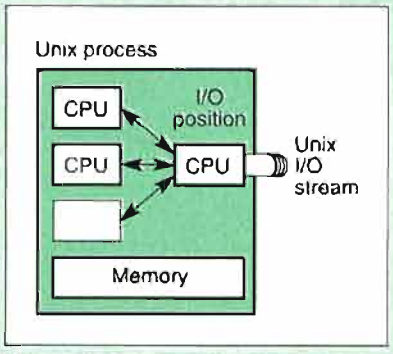

Figure A. Multithreading in operating systems.

Another approach derives from the libraries of parallel functions developed for distributed-memory parallel computers.[2] The first distributed-memory parallel computers had no system software. Users wrote library packages with the minimal

To achieve scalable transfer rates, the operating system sends data directly from the processors to the disks. Throughout Figure 5, the pixels in PE 2 are outlined in color. 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 repeats on 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.

Tags S and R define the pattern of data movement. To preserve the byte stream abstraction, the API must hide the program's tag from that of the I/O device, and vice versa. (See the sidebar "Parallel I/O for supercomputing" on the next page.)

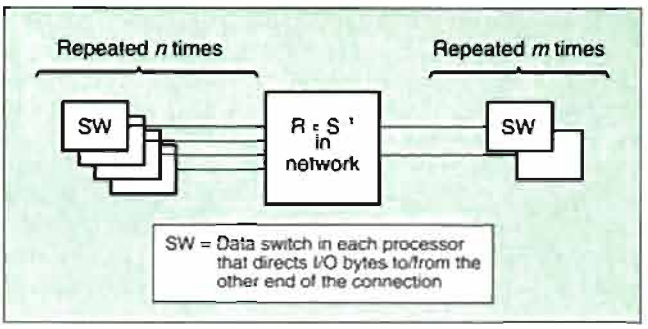

Figure 6. Parallel algorithm to convert distributions.

Scalable I/O streams. Figure 6 shows a scalable algorithm suitable for an I/O stream. This diagram overlays the processors, network, and I/O units of the scalable architecture shown in Figure 1. We show data switches on the left and right sides of the figure. These switches are placed in each processor and I/O device. As the data passes through the left-hand switches, the switch routes individual bytes or blocks to the proper channel on the right. The switches on the right order the arriving data into a single stream. The network in the middle conveys the data blocks to the proper unit.

The algorithm in Figure 6 is the new feature needed in an I/O stream to make the bandwidth scalable. The overall effect of the algorithm is to redistribute data from the left-hand distribution S to the right-hand distribution R. This is $R \circ S^{-1}$ in functional notation (using the functional composition operator $\circ$, defined as $(f \circ g)(x) = (f(g(x)))$.

The algorithm in Figure 6 is both theoretically and practically scalable. As detailed later, 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. Because of the way the algorithm uses the network, the algorithm does not impair scalability either. The networks used in multicomputers experience slowly degrading latency but constant bandwidth per port as the network scales. The rate at which a processor produces data does not change as the number of processor grows. It matches the network's constant bandwidth per port exactly. Data also may be pipelined through the network, diminishing the effect of network latency. Latency increases execution time by just one network latency time per I/O block, which may be thousands of bytes.

While data-switching algorithms are straightforward, the operating system must know S and R to configure the algorithm. The next section describes how the system obtains S and R from executing programs through system calls. Also, we propose (although this is not standard yet) that nonparallel programs and devices receive a default tag that says "no distribution, one processor." This would make the algorithm handle connections between parallel and nonparallel programs and devices automatically.

Interface to emerging parallel languages. Figure 7 illustrates an interface to 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 S and use them for distributing arrays and calculations on the arrays. Until recently, parallel compilers discarded this information, making it unavailable to other parts of the system. Now, parallel compilers are being enhanced so that they place this information into the executable file and generate system calls to pass this information to the operating system before performing I/O. This gives the operating system the information necessary to use the I/O stream extension previously described.

A similar situation exists with data distributions in I/O devices. The system administrator selects R based on the number of disks on the system and other requirements. In current I/O

number of functions needed to run application programs. These library packages included functions for message communications, memory allocation, and arithmetic exception processing. Over time, these library packages evolved into the system calls of a new operating system. Adding Unix system calls has become popular. The resulting operating system has the structure shown in Figure B, with two types of system calls: Unix and parallel. Each type has a standard interface; data is interchangeable only with the same type of interface on any other program.

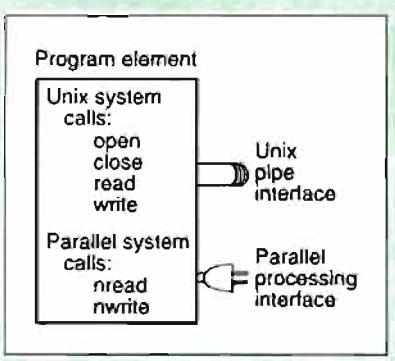

Figure B. Parallel processing library package.

The difficulty with this approach is that Unix and parallel data are incompatible with each other. Scalable rates are possible by using parallel system calls, but this makes the large body of existing Unix programs useless. Similarly, full Unix compatibility is possible by using the Unix system calls, but Unix limits data rates.

References

1. Open Software Foundation, *OSF/1 Programmer's Reference, Revision 1.0*, Prentice Hall, New York, 1991.

2. J. Salmon, "Cubix: Programming Hypercubes Without Programming Hosts," *Hypercube Multiprocessors*, M. Heath, ed., SIAM, Philadelphia, 1987.

device technology, data distribution is entirely the responsibility of the storage subsystem and is unavailable elsewhere. This is equivalent to the parallel compiler's discarding data-distribution information. Making R available in the data switch is crucial to producing scalable I/O to the device.

A system call delivering distribution information to the operating system is the primary extension needed for executable files. If the default is "no distribution, one processor," the new system call does not affect existing programs. That is, existing Unix programs, written without knowledge of parallel processing, become valid parallel programs with one processor. Parallel programs make the new system call and thus enable parallel I/O.

Unix extensions for parallel execution. If users are to run parallel programs with Unix commands, the parallel environment must closely follow Unix abstractions. Surprisingly, we found Unix support for various types of local area networks (LANs) to have the basic abstraction needed for parallel processing. Users need only add instances of this abstraction to support

Parallel I/O for supercomputing

Although there are other forms of parallel I/O, you can view them as variants of the example in Figure 5. (We refer readers to Crockett[1] for a similar taxonomy.) Specifically:

• Each PE may need to access the entire data set. Examples of this include reading a configuration file or accessing a database. Our system supports this mode of access by letting a processor that specifies no distribution tag access the entire file.

• PEs can append asynchronously to the end of a data set. This occurs when you write messages to the end of a debugging file, for example. We support this access scalably by adding a shared read/write pointer, using a message-based distributed addition algorithm based on Gottlieb et al.[2] for the shared read/write pointer.

• Sometimes PEs will perform I/O asynchronously rather than in the single program, multiple data (SPMD) model. We have a user-changeable flag that declares to the operating system that I/Os are in SPMD mode. This cues the operating system to employ certain timing optimizations. With the flag off, the operating system performs asynchronous operations.

• Data is often formatted as variable-length text lines. Nonparallel computers face this issue as well and resolve it by using library packages for formatting and parsing. We believe library packages are also right for parallel computers. A parallel library would transfer data to and from the operating system using a fixed distribution. The library code on multiple PEs would work collectively to format or parse the data.

There are nonobvious criteria for selecting I/O data distributions. Limiting the maximum overhead is the key to a good I/O system. "Having low overhead," however, is different from "being fast." Only programs with simple data distributions can produce data at the fastest rates. To limit overhead, we need fast algorithms for these simple distributions. Conversely, dealing with complex data distributions slows a program and its I/O rate. The system designer can use slower algorithms here without introducing high I/O overhead. This means the I/O system must be most efficient only for those data distributions that correlate with high I/O rates.

Our experience shows that only 1D and 2D distributions need direct support. Of the few programs we found with distributions of three or more dimensions, all but one were computationally intensive anyway. Overhead in redistributing the data here is tolerable. The one exception was out-of-core FFTs, which remain an exception. A 1D distribution views data as a sequence of blocks that are distributed round robin to the units. The distribution of an array or dense matrix by column or row is 1D.[3] So are the interleaved forms of these distributions. We need unequal block sizes when the number of data items is not a multiple of the number of processors, however. The most even division of the data involves some blocks of a particular size and some blocks 1 byte larger.[4] Block-oriented data movement is a 1D distribution. This includes system I/O by formatting or parsing libraries, I/O to a striped disk,[5] and data handling in RAID (redundant arrays of inexpensive disks).[6] Parity block generation in RAID systems is not a form of data distribution because data goes to more than one place.

Computer graphics and visualization make 2D distributions important, although they occur occasionally in other applications. Imaging programs are often computationally simple, yet have high bandwidth requirements. This makes controlling overhead important. A typical distribution for images divides a CRT screen into an array of nearly square subimages. These subimages are mapped to processors. Interleaving is the most common variant. In other applications, distributing a dense matrix by block is a 2D operation. So are the interleaved forms of these distributions. Furthermore, parallel compilers occasionally generate 2D distributions from 1D arrays.[4]

Subsequent to the writing of this manuscript, Ncube and other vendors have refined the system in this article (described in detail in the next sidebar). Ncube now allows data to be distributed in k dimensions by specifying k-1 parameters (the data becomes a stream in the last dimension). Each I/O transfer becomes a k-dimensional subregion. The National Lab's Parallel Data-Exchange Group is developing a system based on a list of distributions. In this system, distributions are identified by the terminology in this article, but a program may supply a list of distributions. Two-dimensional (and higher) data can therefore be represented, although the amount of data required for the representation may be large.

References

1. T.W. Crockett, "File Concepts for Parallel I/O," *Proc. Supercomputing '89*, IEEE CS Press, Los Alamitos, Calif., Order No. 2021M (microfiche),1989, pp. 574-579.
2. A. Gottlieb et al., "The NYU Ultracomputer – Designing an MIMD Shared-Memory Parallel Computer," *IEEE Trans. Computers*, Vol. C-32, No. 2, Feb. 1983, pp. 175-189.
3. G. Fox et al., *Solving Problems on Concurrent Processors*, Prentice Hall, New York, 1988.
4. S. Hiranandani, K. Kennedy, and C. Tseng, "Compiling Fortran D for MIMD Distributed-Memory Machines," *Comm. ACM*, Vol. 35, No. 8, Aug. 1992, pp. 66- 80.
5. K. Salem and H. Garcia-Molina, "Disk Striping," *IEEE Int'l Conf. Data Eng.*, IEEE Press, Piscataway, N. J., 1986, pp. 336-342.
6. D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks," *Proc. ACM SIGMod Conf. Management Data*, ACM, New York, 1988, pp. 109-116.

parallel execution (such as adding two new network types to existing network types like Ethernet).

Parallel execution model. Figure 8 is a defining example of how we extended Unix on the Ncube parallel computer. The example first compiles the well-known Hello World program with the Ncube C compiler. The additional command line switch "-n 4" results in a parallel program with four processors. The second line runs the resulting a.out file. Running a C program in parallel simply runs the program on each processor.

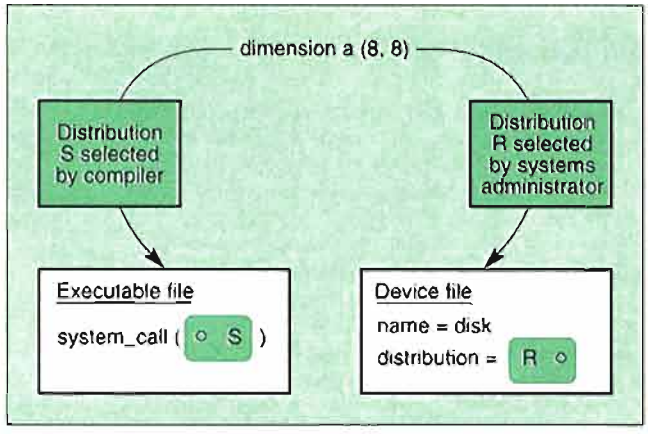

Figure 7. Flow of data-distribution information.

Users could also compile a real parallel program, which also loads and runs the same executable on each processor. However, a real executable does more than just print the same "answer" four times. Each processor first asks the operating system which of the four processors it is. It then uses this information to select the portion of the distributed data it must compute. Following the computation, each processor prints only the part of the answer that corresponds to its part of the data.

The parts of the parallel program also must communicate at times to carry out some computations. Since a parallel program should run on any set of processors, it is inappropriate for a program to use real network addresses for communications. Instead, the processors address each other using nonnegative integers, with the operating system translating the integers to network addresses. To support this communication, parallel-computer vendors typically add a whole new messaging system that takes integer addresses. Unfortunately, the abstractions in these system calls duplicate the ones already in Unix for networking support. To achieve a more concise description, we will describe these features as though there were no unnecessary duplication. The Ncube-2 system likewise eliminates unnecessary duplication.

Program networks. The new messaging systems have the same effect as the program network illustrated in Figure 9 (we discuss the channel network in the figure later). The program network interconnects a specific group of processors using nonnegative integer addresses. The start-up procedure for a parallel program creates one of these networks to connect the processors. This network is then used for interprocessor communications. We emphasize that a program network does not have dedicated hardware but shares the scalable interconnect.

Splitting the Unix process abstraction. The parallel execution model adds a new abstraction to Unix. We see this if we try to apply Unix names to the parts of Figure 8. We associate the term "program" with the file produced by a compiler. We also would say that the Hello World program writes "hello world" on the terminal once. Since Figure 8 has one a.out file but writes "hello world" four times, the two definitions of program conflict. We resolve this by splitting the Unix program abstraction into two parts. The a.out file becomes a parallel program (PP) that runs on multiple processors. We call the code that runs on each processor a program element (PE), even though it is not a complete program. These two abstractions merge for current Unix programs, since all run on one processor. (The sidebar "Program elements and threads" on the next page elaborates on PE terminology.)

As shown in Figure 10, many features of a Unix process become attributed to either a PP or PE. A few receive more sophisticated treatment. Program executables and the *exit* sys-

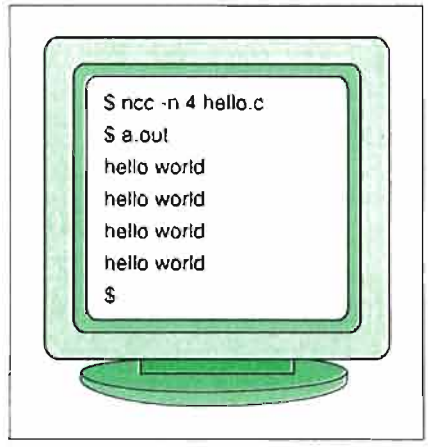

Figure 8. Execution model.

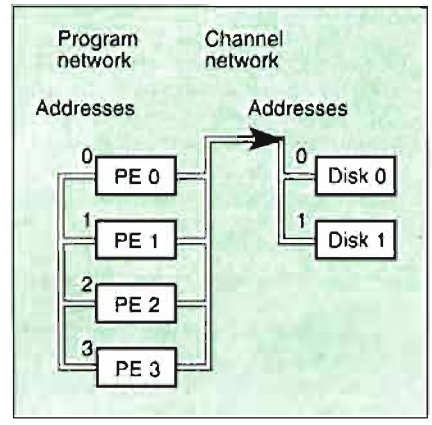

Figure 9. New network types.

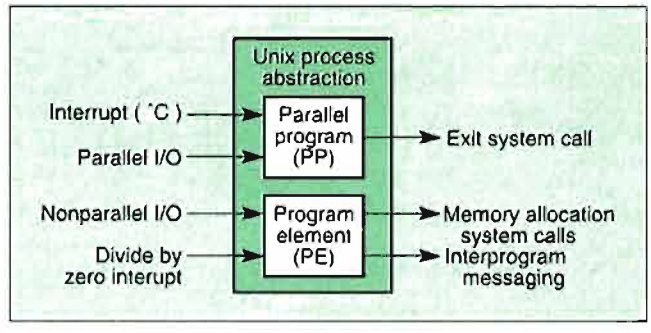

Figure 10. Splitting Unix processes.

Program elements and threads

Multithreaded Unix operating systems for shared-memory parallel computers[1] also split the process abstraction. These systems keep the name *process* for what we call a parallel program and add a new entity called a thread. Threads and PEs are different, although they occupy the same level of abstraction. However, a thread has only an execution context, while a PE has memory as well.

We are deliberately giving the acronym PE two meanings. We chose the term program element as a compromise between operating system and parallel processing terminology. The designers of scalable computers devised the term processing element (PE). It referred to the single-tasking microprocessor and memory that was replicated to form early scalable computers. Over time, programmers expanded this term to refer to the software that runs on the hardware with the same name. We now face a dilemma: You cannot use a hardware term for the parts of a program in the context of operating systems. By using the term program element instead of processing element, we fix this problem while retaining the widely used acronym PE.

Reference

1. Open Software Foundation, *OSF/1 Programmer's Reference, Revision 1.0*, Prentice Hall, New York, 1991.

tem call become attributed to PPs and can affect many processors. Memory allocation, in contrast, applies to a single PE. Interrupts can receive both treatments, depending on the particular interrupt. The user-generated interrupt signal (^C) aborts an entire PP, while a divide by zero interrupts only the PE where the division occurred. Except for I/O, Unix features fit easily into the parallel model without changing their current behavior.

A full treatment of this allocation would be more complex than described here, yet simpler than the corresponding treatment in multithreaded Unix, which additionally deals with communication between threads using shared memory. These communications increase the complexity of interactions between system features.

Implementation on Ncube

We wrote system software for the Ncube parallel computer on the basis of the ideas presented in this article. This software, now version 3.0 of Ncube's system software, is commercially available. All features in this article are available except for preconnecting a parallel file to a program. This release is not a Unix port, however. We implemented operating system features in the C and Fortran compiler libraries running on top of Ncube's proprietary communications kernel. This sidebar describes some design choices in the Ncube implementation and shows that scalable operation is compatible with Unix.[1]

Figure C shows a data-distribution function. The function maps byte positions in an I/O stream to a byte position in one of the replicated units. We represent the position of a byte as a binary number and apply it to the left of the function. The action of the function is to permute the order of bits as shown by the lines in the figure. The right-hand side of the function divides the bits into two groups: The bottom group represents the unit number; the upper group represents the byte position within that unit. The specific function shown is the distribution on the left side of Figure 5. With the proper permutation, you can express any n-dimensional distribution with or without interleaving. The limitation is that block sizes must be powers of 2.

This class of functions has useful mathematical properties. The data switch uses these properties to compute destinations and block sizes. Figure D illustrates these properties. The figure shows S from Figure C mirrored about the vertical axis, forming S^{-1}. The middle and right parts of the figure show the composition of S^{-1} and R. (R is the data distribution for the disk in Figure 5.) We compose S^{-1} and R by tracing each bit from the left side of S^{-1} through the common boundary to the right side of R. We call the resulting function $R \circ S^{-1}$ a composite data distribution. The data switch uses it to direct data from the sending side directly to the receiving side.

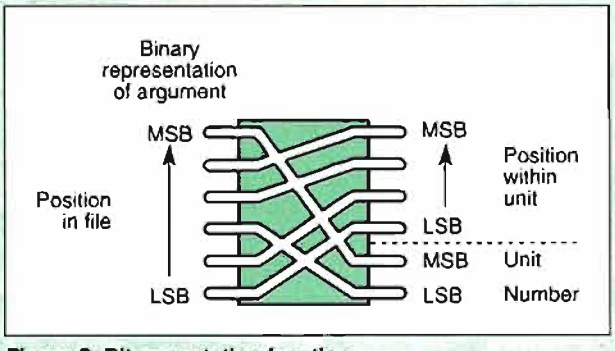

Figure C. Bit permutation function.

This approach is particularly efficient for sending data in blocks rather than one byte at a time. DeBenedictis and del Rosario[1] show a method of computing block size. The only aspect of the block size computation actually related to the block size involves counting the number of parallel traces in the composite data distribution. With n parallel traces, you can send a 2^n-byte block. These data structures and algorithms should result in a small computational overhead, which we show below.

We ran numerous 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. The detailed results appear in DeBenedictis and del Rosario[1] and in the Ncube technical documentation.[2]

We include one performance result to show that the high-level approach taken in this article can be efficient. Figure E shows a parallel pipeline, or two programs running simultaneously on a parallel computer. The output of one program becomes the input of the other. In the trial runs, the programs input or output a 1,024 × 1,024-byte matrix. We distributed the matrix over a varying number of processors using row-, column-, and block-oriented distributions. The parallel code is similar to that shown in Figure 11, except there is no "open" command. To provide a context for the performance numbers, this computer has 2-Mflops processors connected as a hypercube with 2.22 Mbyte/s communication paths. Table A shows performance results.

The results of these tests meet expectations perfectly. Since these tests use a fixed-size data set, the running time decreases as the number of

Link to I/O. Let's discuss parallel I/O enhancements for the example in Figure 5. Figure 11 illustrates some code required for our approach to this example. The key issue in opening files is whether the entire parallel program is opening a file collectively or a PE is opening one individually. The behavior differs, for example, if the file-opening operation creates a file. If several PEs attempt to create the same file individually, only one can succeed. The others will fail because the file already exists. We reconcile this by letting a PE open or create a file in global mode. In this mode, only one PE actually creates the file; it then relays the resulting handle to the other PEs.

In our system, the Unix system call *ioctl* moves information about data distribution from the executable file into the system. Ioctl performs the function selected by the middle argument Cmd (command) on the specified file. The third argument is a pointer to an arbitrary block of data containing an encoding of the tags we have been using in this article.

The write statement starts a series of activities that can move data at scalable rates. Before any data can move through the I/O stream algorithm shown in Figure 6, we must configure the data switches to perform R S^{-1}. The ioctl call cannot do this because the ends of the channel can receive data tags at different times. Both S and R will be available before the first data movement, however, because the system swaps the S and R information between the ends of the channel. This configures the data switches, as shown

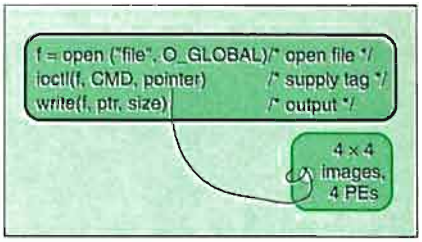

Figure 11. Parallel program example.

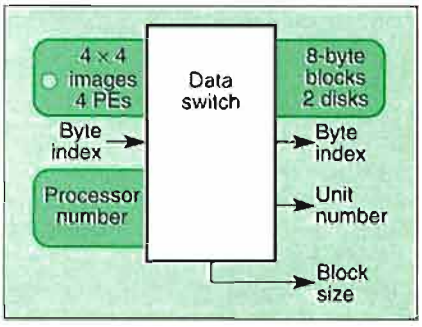

Figure 12. Data switch.

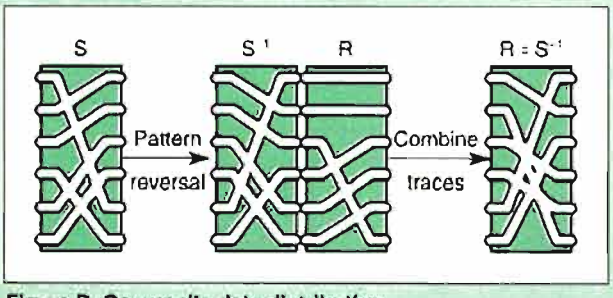

Figure D. Composite data distribution.

processors increases. The decrease is not exactly linear because there is a small overhead independent of the size of the data transfer (4 milliseconds, in this example). This causes the rate per node to decline. Parallel programs without I/O have equivalent behavior under the same circumstances, which led to the "scaled speedup" model.[3]

Since the trials with mixed distributions perform data permutation as well as data movement, lower performance could be expected. We selected the row-to-column trial specifically because it was pathological, as its performance results reflect. In this case, each processor sends a small message to every other processor. Known scalable algorithms for matrix transposition[4] would avoid this performance problem.

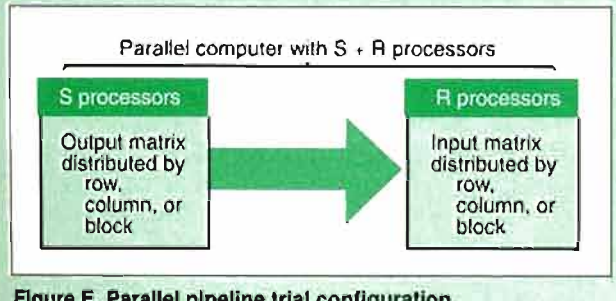

Figure E. Parallel pipeline trial configuration.

Table A. Pipelines between programs of various sizes.

S, R	Sender distribution	Receiver distribution	Rate per node (in MBps)	Aggregate bandwidth (in MBps)
1	row	row	2.20	2.20
2	row	row	2.18	4.36
4	row	row	2.14	8.56
8	row	row	2.07	16.6
16	row	row	1.94	31.4
32	row	row	1.72	55.0
64	row	row	1.68	108.0
64	row	block	0.36	23.2
64	row	column	0.03	2.11

References

1. E. DeBenedictis and M. del Rosario, "Modular Scalable I/O," *J. Parallel and Distributed Computing*, Vol. 17, No. 1, Jan. 1993, pp. 122-128.
2. *Ncube-2 6400 Series Supercomputer*, Ncube Corp., Foster City, Calif, 1989.
3. J. Gustafson et al., "Development of Parallel Methods for a 1,024-Processor Hypercube," *SIAM J. Scientific and Statistical Computing*, Vol. 9, No. 4, July 1988, pp. 609-638.
4. G. Fox et al., *Solving Problems on Concurrent Processors*, Prentice Hall, New York, 1988.

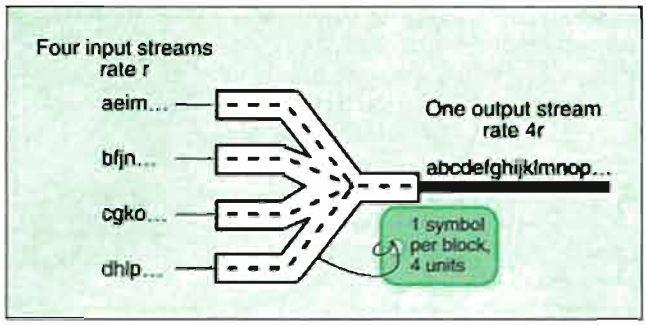

Figure 13. Stream combiner.

by the tags in Figure 12.

The easiest explanation of data movement assumes that each byte is moved individually. The process starts with the index of a byte within the PE's I/O stream. After receiving the tags shown in Figure 12, the switch can compute the destination for a byte. The destination consists of a destination unit number and an index within that unit. Here, a unit is a disk or a processor, depending on the nature of the connection. The byte is then sent to this destination in one of two protocols, as described in the next section. This expanation applies in concept to most scalable I/O systems but details vary widely.

Actually, to obtain acceptable performance, data must move in blocks rather than one byte at a time. We enhanced the data switch to emit a block size as well. Block size computations require unexpectedly complicated math, however.

Channel networks. We propose channel networks as a way of representing scalable I/O streams in the Unix I/O stream model. (Figure 9 shows such a network.) Like a program network, a channel network connects a specific group of processors and uses integer addresses. It connects two parallel entities, however, such as a PP and a parallel disk, both having addresses starting at 0. Instead of circulating within the sending parallel entity, messages go to the other end. This means that messages coming from the left side are delivered to the right, and vice versa. We enhanced the operating system to set up a channel network for each I/O stream open at startup. The data switch is straightforward when used with a channel network. It simply sends the data onto the network by using the unit number as the address.

We need two protocols for various I/O devices, corresponding to character and block devices in Unix. One protocol deals with potentially random-access disk requests. Here, the user's PE must send messages that identify exactly where in the file to read or write. This protocol is similar to the network file system (NFS) protocol for workstation disk accesses over a network. For a device not supporting random access (like a communications line), the data's position in the stream is implicit. Tagging data with its position is unnecessary. Here, however, the receiver must have a data switch to know which sender to obtain the data from next. (Refer to the sidebar "Implementation on Ncube.")

Scalable I/O hardware

While Tflops computers are under construction, the fastest external networks at the same stage of development operate at about 1 gigabyte per second. This is under 0.1 percent of the bandwidth required for a traditional I/O balance. By using the software described earlier and the "stream combiner" architecture in Figure 13, you could build a scalable network that any Unix program can drive. You could then scale the network to a Tbyte/second level to balance a computer scaled to Tflops operation. The same architecture could drive any fast medium, such as a radar antenna or a video display.

The device in Figure 13 works like the section of roadway between a toll booth and a bridge. Since speed at toll booths is slower than in traffic lanes, a two-lane bridge may need 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 cars become closer together and traffic speeds up. This lets the bridge run at full capacity though individual toll booths are slow. The stream combiner we illustrate works like four toll booths and a one-lane bridge. While cars merge randomly on a roadway, the stream combiner merges the data according to the simple interleaving pattern shown. Curiously, several parallel computer vendors have I/O devices that look like the one in Figure 13, but the appearance is superficial because they use a different concept.

The merging pattern shown corresponds to the 1D data distribution described earlier. This lets you create a data-distribution tag for the stream combiner. If you then treat stream combiner inputs as the units of a parallel entity, the software described earlier will work. However, you must tailor a stream combiner to each specific medium. For example, an engineer must pick a size for the symbols in Figure 13 and understand the flow-control requirements of the medium.

This article outlined the evolution of software that will make parallel computers as easy to use as nonparallel ones. This evolution involves some new ideas but does not require developing an operating system from scratch. Most of the code now written for Unix will run unchanged on a parallel computer.

Here is what a Tflops supercomputer will look like. Parallel-computer vendors will construct Tflops processors and balance them with scalable secondary storage. Parallel I/O to fast networks and special I/O devices will be available as options. The standard configuration will include nonparallel LAN interfaces.

The standard configuration will also include an operating system like the one described here and data-parallel C and Fortran compilers. The operating system will include the standard set of Unix tools and utility programs compiled (but not rewritten) to run on one processor. This suite of several hundred programs will provide the networking, electronic mail, and other traditional tools that make Unix attractive.

The user would add parallel applications, like weather prediction or molecular modeling. These would be Tflops programs with balanced I/O. Users could freely mix and match the parallel programs with the Unix utilities. For example, you could direct the output of

a parallel program to a workstation running data visualization software via a LAN.

While recompiling the hundreds of well-known Unix utilities would provide a great deal of software at low cost, this approach has limits. As the application programs become more powerful, some nonparallel Unix utilities would become bottlenecks. For example, the Unix copy command would be the only tool available to copy a Tbyte file distributed over 10,000 disks. Since the Unix utilities run on only one processor, the operation would be very slow. As they become bottlenecks, these Unix utilities would be rewritten as scalable programs, while providing the same interface.

We now have a computer with parallel processing "under the hood." Just as new cars have familiar controls, Tflops supercomputers will have familiar commands for editing files and sending electronic mail. Running or writing computationally intensive applications with the familiar commands yields a surprise: The computer runs 10,000 times faster than a workstation. ■

Acknowledgments

We thank Mike del Rosario of Ncube for doing the coding for much of the I/O system and all the performance trials, Marina Chen of Yale University for joint work on the compiler interface (in 1988), and the Ncube development staff for developing the rest of the software releases that this work became part of.

References

1. E. DeBenedictis and S. Johnson, "I/O for Tflops Supercomputers," *Proc. Sixth SIAM Conf. Parallel Processing Scientific Computing*, SIAM, Philadelphia, 1993, pp. 751-758. (This article is an expanded version of this paper.)
2. *Ncube-2 6400 Series Supercomputer*, Ncube Corp., Foster City, Calif., 1989.
3. *The Connection Machine CM-5 Technical Summary*, Thinking Machines Corp., Cambridge, Mass., 1991.
4. *Paragon XP/S Product Overview*, Intel Supercomputer Systems Division, Santa Clara, Calif., 1991.
5. L. Ni and P. McKinley, "A Survey of Wormhole Routing Techniques in Direct Networks," *Computer*, Vol. 26, No. 2, Feb. 1993, pp. 62-76.
6. J. Van Zandt, *Parallel Processing in Information Systems*, John Wiley and Sons, New York, 1992.
7. G. Fox et al., *Solving Problems on Concurrent Processors*, Prentice Hall, New York, 1988.
8. P. Hatcher et al., "Data-Parallel Programming on MIMD Computers," *IEEE Trans. Parallel and Distributed Systems*, Vol. 2, No. 3, July 1991, pp. 383-388

Erik P. DeBenedictis is president and cofounder of Scalable Computing, a start-up company applying parallel processing technology to computer networking. His interests are the reduction to practice and commercialization of parallel computer technology. In 1981 he designed the first hypercube multiprocessor, called the Cosmic Cube, at the California Institute of Technology. At Bell Labs, he received a Gordon Bell Prize in 1988 and began research work on parallel I/O. During 1991, he put this technology into the system software of the Ncube parallel supercomputer.

DeBenedictis received a BS from the California Institute of Technology in 1978, an MS from Carnegie Mellon University in 1979, and a PhD in computer science from Caltech in 1983. He is a member of the IEEE Computer Society.

Stephen C. Johnson is chief executive officer and cofounder of Scalable Computing. His interests include applying constraint-based programming to the design of graphical user interfaces and parallel software. He spent 25 years pioneering Unix at AT&T Bell Laboratories and Stardent Computer. His technical efforts produced the portable C compiler and the YACC compiler generator. He managed the first release of C++ from AT&T and was responsible for a wide range of system software products at Stardent. In 1990 he joined Ncube as vice president of software, where he reorganized and expanded the software organization. He is president of Usenix, the Unix users' group.

Johnson received an AB in mathematics from Haverford College in 1963 and a PhD from Columbia University in mathematics in 1968.

Readers can contact the authors at Scalable Computing, 204 Cantoe Crt., Redwood City, CA 94065, (415) 591-1442. DeBenedictis' e-mail address is 71553.714@compuserve.com; Johnson's is scj@usenix.org.