Modular Scalable I/O

Erik P. DeBenedictis

Scalable Computing
204 Canoe Court
Redwood City, CA 94065

Juan Miguel del Rosario

Syracuse University
Syracuse, NY 13244

We describe design issues for scalable I/O systems. I/O system modularity is the central issue. We identify the new information and algorithms responsible for high performance in scalable I/O. We show how to assign them to the parts of a scalable computer system so the result is modular. We built a modular I/O system and it is commercially available. We describe its design and performance benchmarks using it.

1. SCALABILITY

We clarify the idea of scalability before extending it to I/O. Fig. 1 shows a scalable architecture. The figure is halfway between a machine design and an algorithm. If we supply actual values for *n* and *m*, then we can convert the architecture into a machine design. Like an algorithm, however, a scalable architecture shows the behavior of the resulting machines as *n* and *m* vary. One easily sees the number of building blocks grows linearly and the depth of the interconnection network grows logarithmically with *n* and *m*.

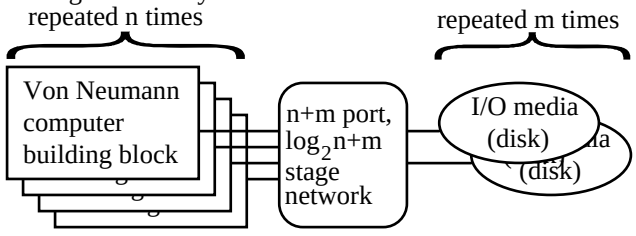

Fig. 1: A scalable I/O architecture

The U. S. Government has an initiative to produce a TFLOPS (one million MFLOPS) computer around 1995. While we don't yet know the exact design of such a machine, a likely candidate is Fig. 1 with *n*=10,000 and 100 MFLOPS processors. This paper deals with strictly with scalable I/O, where *n*=10,000 is a typical value.

2. I/O REQUIREMENTS OF A SCALABLE COMPUTER ARCHITECTURE

A rule of thumb in the supercomputer industry is "a megabyte per MFLOPS is balanced." This means that computers where the I/O bandwidth in megabytes per second equals the computing capacity in MFLOPS have a good I/O balance. A scalable architecture with *n* processing elements will have a computing rate proportional to *n*. This means the I/O rate also should be proportional to *n*.

Fig. 1 shows the scalable I/O architecture. The figure has *n* Von Neumann computing elements and *m* disk drives (or other media). The ratio of *n* and *m* controls the I/O balance and the size of *n* and *m* controls the overall performance.

Adding scalability does not guarantee scalable performance. Say we find a scalable way of doing 99% of the activities in an application. The remaining 1% will continue to run at their original speed, however. For *n*=10,000 the total speed improvement is 99, much less than the 10,000 we expect! This is Amdahl's Law [1]. A system with fully scalable performance must be scalable to one part in *n*, or 99.99% for the *n*=10,000 example. This paper is not about "adding scalability," but "removing the last vestige of nonscalability."

Since we have already presented the hardware architecture, the remainder of the paper presents software algorithms for a scalable I/O system. We describe new or changed algorithms so the reader can see that they are scalable. For additional support, however, we end with results from benchmark programs. Each result tests the scalable architecture for one combination of *n* and *m*. Where performance varies with *n* and *m* in the expected way, one gets additional support for scalability.

Many other algorithms can stay in their current form because they are not in time critical parts of the system. We do not mention these because they are unchanged from a regular computer.

3. EXISTING WORK

There are dozens of commercially available parallel computers with shared memory and multithreaded operating systems. These systems have full-featured and parallel I/O, but they have values of *n* between two and about thirty. These values are too small to be scalable.

While Intel's Concurrent File System™ [12] uses the hardware architecture shown in Fig. 1, some software algorithms are nonparallel. The result is increased performance but no claim of scalability.

Thinking Machines' CM-2 [14] had a scalable I/O system, but a SIMD architecture. The CM5 [15] will have a MIMD scalable I/O system, although it is still under development.

General ideas for parallel I/O appear in Crockett [3] that are compatible with the approach in this paper.

Other publications by the authors [2, 4, 6, 8] report other aspects of this system.

4. MODULAR SCALABLE I/O

The modularity property of an I/O system is directly responsible for ease of use. It lets one connect any program to

any I/O device easily and without chance of error. Curiously, the technical basis of modularity is not elaborate algorithms. Instead, modularity is the result of assigning each algorithm to the correct part of the computer. By parts, we mean the user's program, operating system kernel, device drivers, and I/O libraries. Correctly assigned algorithms will work for any assembly of the parts. The key in designing a scalable I/O system is to keep the performance advantages of scalability while building a modular I/O system.

5. EXAMPLE

In making the I/O system, one must first identify any new information and algorithms that arise from parallel processing. These are then assigned to make the system modular. In Fig. 2 we show a scalable I/O task to introduce these new features. The figure shows a scalable program outputting an image to a scalable disk system. The center of the figure shows the image within an outline of a CRT display. Imagine the image is 8×8 bytes or pixels.

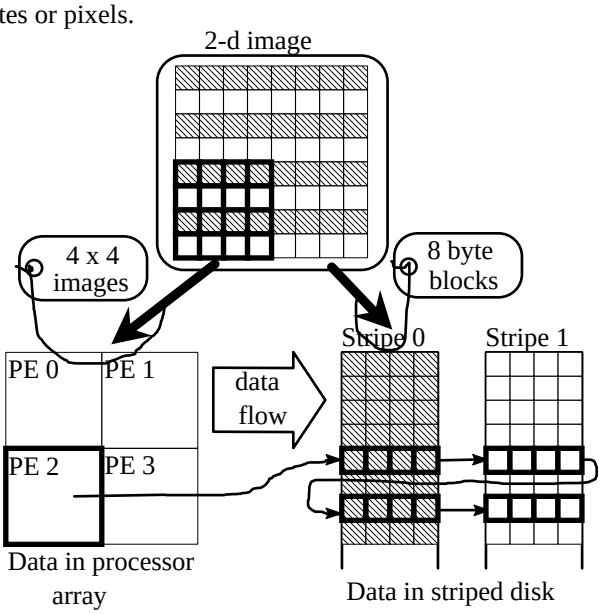

Fig. 2: Parallel I/O example

5.1. New information

On the left of the diagram are four processors of a parallel program. As is common in image processing applications, one breaks the image into square regions with the regions assigned to the processors of the parallel computer. We have highlighted Processing Element (PE) 2 and the part of the image mapped to this processor. We have a tag with the words "4×4 images" to describe the form of data distribution from the image to the parallel program. The example generalizes if the tag can represent any distribution popular in parallel programming [9].

On the right-hand side we show the parallel disk system. On each of two disk drives we show a stripe file for this data. In disk storage systems, one views data as a one dimensional stream of blocks, with a round-robin assignment of blocks to disks. The tag on the right is an example of this by saying "8 byte blocks." We divide the image, imagined in raster order, into eight byte blocks and map them to the drives. The figure patterns the contents of one drive and the portion of the image that goes to that drive. We generalize the example by letting the tag represent any popular storage distribution [11, 13].

Now we can see the first new feature of scalable I/O. The tags representing data distributions are new in scalable I/O and have no analog on a conventional computer.

5.2. New algorithms

The hand-drawn line at the bottom of the figure illustrates the second new feature. Output from a program on a workstation, for example, has simple behavior. The data goes to one file starting at the beginning and filling the file without gaps until the end. We highlight the flow of data from PE 2. It highlights the processor, its part of the image, and the corresponding data on the disk. One sees the output goes to two disks, not starting from the beginning, getting stored with gaps, and not ending the files. A new algorithm must be present in this scalable computer to create this behavior. Each byte of output data gets targeted by the algorithm to the proper destination device and position in that device. We call this the conversion algorithm, because it converts data distributed in one form to another. This algorithm has no analog in a conventional system.

5.3. Other modes

Fig. 2 is a key I/O paradigm, but we need a host of variants. Sometimes each processing element needs to access an entire data set, not just a piece of it. Defeating the left distribution tag in Fig. 2 does this. In addition, multiple processors sometimes need to do I/O using a common file pointer [7]. This is a common mode in nonparallel computers. Finally, a nonparallel program (like a text editor) may need access to parallel files. Fig. 2 already implies this. If the program in Fig. 2 is a text editor, it will have only one processor and data distribution will be irrelevant. Although these variants in the nCUBE implementation, there is no further mention of them because they are all based on Fig. 2. There is no guarantee, however, that other I/O paradigms will not appear in the future that require other modes.

6. ASSIGNING THE NEW FEATURES

6.1. Before execution

Fig. 3 shows the situation before execution. Before execution, data distribution information must be in executable files. To see why this is, imagine switching a user program's output to a terminal from a disk. This removes the disk driver module and replaces it with a terminal driver module. The resulting configuration does not need knowledge about disk striping but must know that the terminal requires sequential data. Since the device driver is the only part of the system to change, this information must be in the device driver.

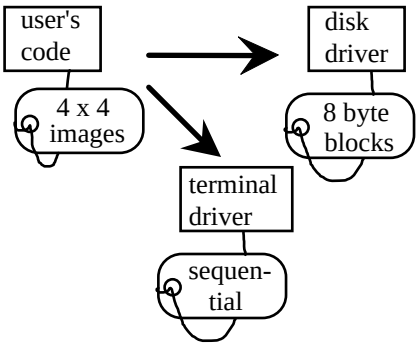

Fig. 3: Assignment before execution

6.2. During execution

Fig. 4 shows the situation during execution. System performance controls the behavior during execution. We see this by asking "what is the bandwidth requirement on the conversion algorithms during execution?" The program produces data at a rate proportional to *n*, the number of processors. Similarly, the disk system consumes data at a rate proportional to *m*, the number of disk drives. For arbitrarily large values of *n* and *m*, the bandwidth of the conversion algorithm will be arbitrarily large. This precludes a nonparallel implementation.

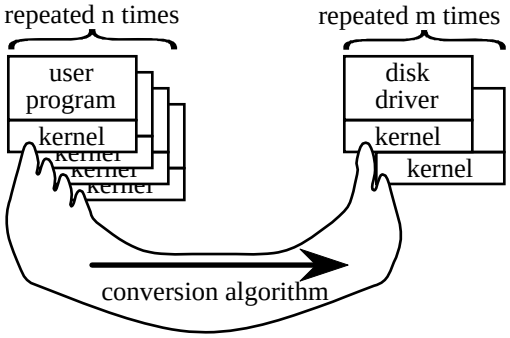

Fig. 4: Assignment during execution

In general, one must distribute the conversion algorithm among a scalable number of processors. Our conversion algorithm includes a subroutine in the kernel of all processors. Since the subroutines get the CPU resources of their processors, the total resources available to this algorithm are *n+m*. As *n* and *m* grow, the resources available to the conversion algorithm grow at the same rate, giving it greater conversion capacity. The conversion is more than series of isolated subroutines, however. Variables in the subroutines identify the other end of the connection. This makes the conversion algorithm look like an ameba. The ameba has pseudopodia extending into each processor, yet has a single structure.

6.3. Conversion algorithm

Fig. 5 illustrates the subroutine in each kernel in greater detail. The kernels of all the processors contain a generic conversion algorithm. This algorithm converts any distribution to any other distribution. Setting up an I/O connection customizes a subset of these to do a specific conversion. This brings the data distribution tags together from the executable files and plugs them into the conversion algorithms as parameters. The algorithm also must have the processor number of the parallel program it is running on plugged in. Having been so customized, the conversion algorithm can target a byte of data appropriately. The algorithm receives the position of a data byte in the output stream as the *byte index*. The conversion algorithm then computes a *unit number* and *byte index* that direct that byte. For efficiency, the conversion algorithm emits a *block size*. This specifies the number of bytes that are adjacent at both ends of the connection and form a block for transmission purposes.

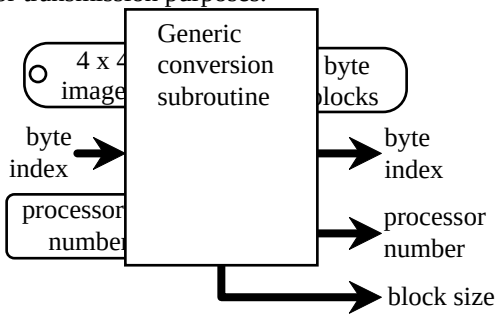

Fig. 5: Data conversion algorithm

7. IMPLEMENTATION ON NCUBE

In this section we expose general design issues using the nCUBE scalable I/O system as an example. We describe the data structure we chose to represent data distributions and the algorithm that converts the distributions defined by these data structures. In describing the capabilities and limitations of the nCUBE system, we create guidelines for designing future systems. We restrict our discussion only to points about scalability, however. See the nCUBE technical documentation [5] for a complete description.

7.1. Data distribution functions

Fig. 6 shows a data distribution function. The function maps each byte position in the file to a byte position in one unit. For programs, the functions map each byte to a position in the I/O stream of a processor. For a striped disk, the functions map each byte to a position in a particular disk unit.

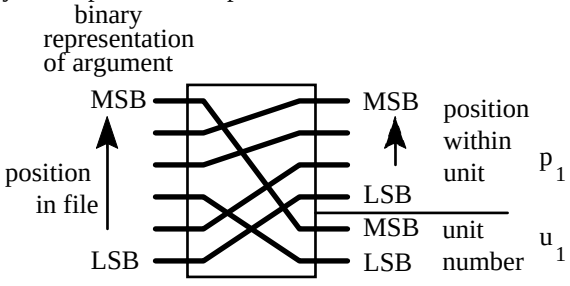

Fig. 6: Data distribution function

The distributions work as follows [2]. 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.

7.2. Representation of Common Data Distributions

The sections below show that these functions can represent the necessary range of data distributions. There is a limitation,

however. The limitation is that the number of units and block sizes must be powers-of-two. Del Rosario [8] has a system that alleviates this restriction.

Fig. 7 illustrates the most common form of data distribution. This distribution views the file as a sequence of blocks. The blocks get distributed round-robin to the units of the parallel entity. The permutation pattern controls the block size and number of units. Striped disks [13] use this mapping, as do parallel programs distributing a dense matrix by column or row [9]. The data distribution in parity protected disk arrays [11] follows this paradigm, although the parity blocks must be generated by a postprocessing step. The block size in the example is eight bytes, corresponding to the disk mapping in Fig. 2.

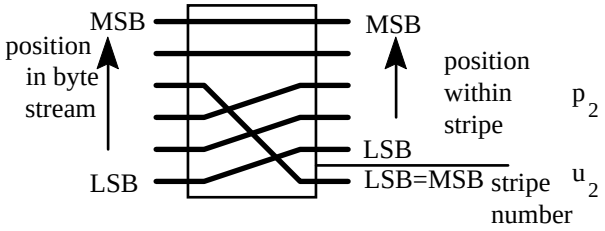

Fig. 7: Mapping for a striped disk

Fig. 6 illustrates the two-dimensional data distribution from Fig. 2. The distribution views the file as a two dimensional array scanned in raster order. Two dimensional subarrays are then mapped to processors [9]. The permutation pattern controls the size of the subarray in each dimension and the number and arrangement of the processor array. The figure illustrates a 8×8 matrix of bytes distributed over a 2×2 array of processors.

7.3. Distribution conversion algorithm

In building the conversion algorithm, the first step is to understand how to move single bytes between the ends of the connection. There are two distributions involved in Fig. 2, which we will call D1 and D2. Functions D1 and D2 represent the distributions of data in the sending and receiving entities respectively. We know a byte's unit and position within the unit (u_1, p_1) and need to compute the receiving unit and position within the unit (u_2, p_2). Computing the position in the file as an intermediate value, however, is the initial strategy. Fortunately, we can use the bit permutation functions in both directions. Fig. 6 represents function D1 with (u_1, p_1) applied to the right-hand side. We can find the position in the file by permuting the bits in a right-to-left direction. We can then apply this result to Fig. 7 to compute (u_2, p_2). When we can evaluate a mapping function in the reverse direction, we have a method, possibly slow, for doing scalable I/O.

Symbolic manipulation of distribution functions improves the previous method. Perhaps counter intuitively, the position of a byte in the file is irrelevant. Fig. 8 shows D1 mirrored about the vertical axis, forming $D1^{-1}$. The middle and right parts of the figure show the composition of $D1^{-1}$ and D2. We trace each bit from the left side of $D1^{-1}$ through the common boundary to the right side of D2. At this point, the position in the file disappears. The name of the function on the right side is a composite data distribution. It represents the data permutation from the sending side directly to the receiving side. Evaluating the composite function with (u_1, p_1) on the left side makes (u_2, p_2) appear on the right side.

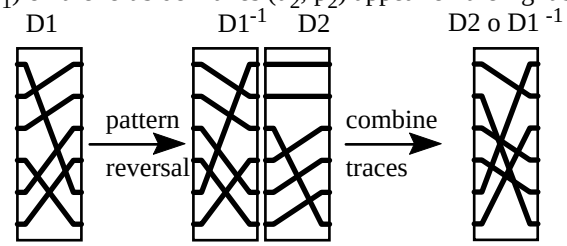

Fig. 8: Generation of a composite mapping function

7.4. Blocking of Parallel Data Transfers

One improves data transfer efficiency by sending data in blocks, instead of byte-at-a-time. Symbolically evaluating composite functions yields more information than just where one byte goes. One also finds out how many bytes are adjacent on both the sender and receiver. The system can send these bytes as a block.

Fig. 9 starts by illustrating the composite mapping function from Fig. 8. Let us follow the example started in Fig. 2 by following output from PE 2. Say PE 2 has already output the eight bytes in the upper half of its image and is now outputting the second eight bytes. Now, let us apply (u_1, p_1) to the left side of the mapping function, but let us do it symbolically. Specifically, the unit number is two, resulting in $u_1=10_2$, and the byte positions are 1000_2 through 1111_2, resulting in $p_1=1xyz_2$. Here *x*, *y*, and *z* are variables that match either a 0 or 1 in a binary number. The illustration shows these patterns applied to the left, and shows the pattern that results from the permutation of the symbols on the right. Note that the unit field u_2 has pattern x_2, which matches single bit integers 0 and 1. This means that the specified transfer will send data to disk units 0 and 1, consistent with Fig. 2. Repeating this analysis for position field p_2, we find the pattern $110yz_2$ representing the integers 24 to 27. The output operation will therefore send four bytes from the program to each disk platter, consistent with Fig. 2.

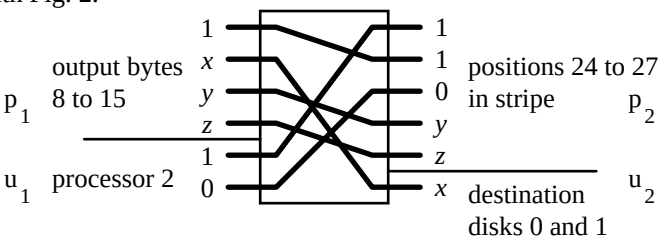

Fig. 9: Composite mapping function

The final analysis step is to observe that the two least significant bits of the position in the sending processor (yz_2) are the same bits and in the same order as the two least significant bits of the position in the disk stripe (yz_2). This final condition allows one to send a four-byte block from sender to receiver. The overhead in symbolic evaluation is likely to exceed the efficiency improvement of four byte blocks. In realistic situations blocks of 512 to 65536 bytes are common and blocking makes a dramatic difference.

8. PERFORMANCE BENCHMARKS

We made a computing environment using the methods just described and ran benchmark programs in it. The first set of programs exercised a parallel pipeline. In a parallel pipeline, two programs run simultaneously on a parallel computer. Output of one program becomes input of the other. This is a paradigm of increasing popularity on parallel computers [10]. In the benchmarks, the programs input or output a 1024×1024 byte matrix. We distributed the matrix over processors using row, column, and block distributions. We show the benchmark in Fig. 10 and performance results in table I.

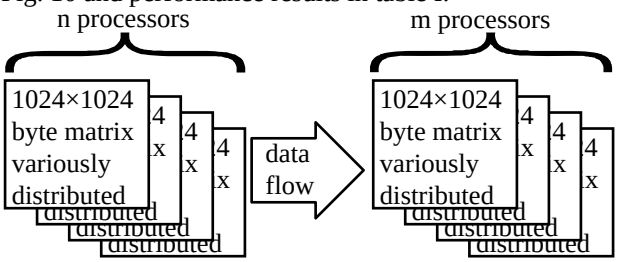

Fig. 10: Parallel pipe trial configuration

n=m	sender distri-bution	receiver distri-bution	rate per node (MBytes per second)	aggre-gate band-width
1	row	row	2.20	2.20
4	row	row	2.14	8.56
16	row	row	1.94	31.4
64	row	row	1.68	108.0
64	column	column	1.68	108.0
64	row	column	.03	2.11
64	row	block	.36	23.2
64	block	block	1.68	108.0

Table I: Pipeline benchmark

The first four simple benchmarks are of the same program on differing numbers of processors. The system initially does a brief calculation that concludes that the data distributions are the same. Then, pairs of processors exchange a single 1/*n* megabyte message. The *n*=1 case sends a one megabyte message that approaches the peak hardware rate of 2.22 megabytes per second. As *n* increases, the messages get shorter and message transmission overhead to become noticeable. Nevertheless, with *n*=64, the I/O speedup is nearly 50.

The last four tests vary the data distributions. The tests where the sender and receiver distributions are the same have the same results. This is because actual flow of data is the same as in the first four tests. In the two tests where the distributions differ, performance is much lower. This is because the system is performing a data permutation as part of the I/O operation.

In the second set of benchmarks, a program outputs a 1024×1024 byte matrix to a parallel disk. The code is the same as that developed in the previous section. Fig. 11 shows the benchmark and table II shows performance results.

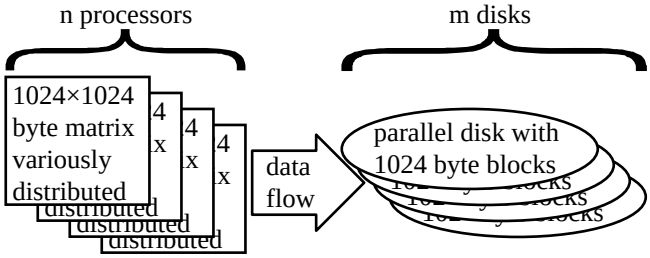

Fig. 11: Parallel output trial configuration

n	output distri-bution	m	disk bandwidth (MBytes per second)
64	row	1	1.30
64	row	2	2.49
64	row	4	4.96

Table II: Parallel disk benchmark

9. CONCLUSIONS

This paper discusses making I/O both scalable and modular. There is a well-known procedure for making an I/O system modular, but nobody has applied it to a scalable architecture before. We identify new information and algorithms characteristic of scalable computers and repeat the procedure.

The result is that before execution, the new data distribution information must be separate for modularity. During execution, data distribution information gets brought together for use by the new data distribution algorithms.

Data distributions need a range of properties, which are summarized below: They must be

1. representable as a data structure,
2. capable of representing common distributions in parallel programming and I/O device design,
3. have efficient algorithms for functional inversion and composition,
4. allow symbolic evaluation to support computing block size.

We built a system to test these ideas. The system is available from nCUBE in release 3.0 of system software. We ran benchmarks on this system showing the scalability of I/O.

The principle limitation in release 3.0 is that all block sizes and numbers of units must be powers-of-two. Release 3.1 of the nCUBE system software has a variant that alleviates this deficiency [8].

The primary area for further work is in the representation of data distributions. Experience shows that one and two dimensional distributions are probably sufficient, but the power-of-two limitation draws criticism, however. We suggest developing a new class of data distributions that address these points while preserving the properties described earlier.

This paper did not discuss scalable I/O with a common file pointer or I/O to devices other than secondary storage. Other works by the authors [5, 7] address these issues, however.

10. REFERENCES

1. Amdahl, G., Validity of the single-processor approach to achieving large-scale computer capabilities. *AFIPS Conference Proceedings*. Spring 1967, pp. 483-485.
2. Chen, M., and DeBenedictis, E., Separate Compilation and Dynamic Linking of Parallel Programs. Seminar at Yale University, May 1988.
3. Crockett, T., File Concepts for Parallel I/O. *Proceedings of Supercomputing '89*. Pp. 574-579.
4. DeBenedictis, E., and Madams, P., nCUBE's Parallel I/O with Unix Compatibility. *Proceedings of the Sixth Distributed Memory Computing Conference*. May 1991.
5. DeBenedictis, E., and del Rosario, J., Scalable I/O. nCUBE Technical Report, nCUBE-TR-001-911015.
6. DeBenedictis, E., and del Rosario, J., nCUBE Parallel I/O Software. *Proceedings of the 1992 International Phoenix Conference on Computers and Communications*. April 1992, pp. 117-124.
7. DeBenedictis, E., unnamed. Submitted to *IEEE Proceedings*, April 1993.
8. del Rosario, J., High Performance Parallel I/O on the nCUBE 2. *IEICE Transactions (English Edition)*. Japan, August 1992.
9. Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D., Solving Problems on Concurrent Processors. Prentice-Hall 1988.
10. Gustafson, J., Benner, R., Sears, M., and Sullivan, T., A Radar Simulation Program for a 1024-Processor Hypercube. *Proceedings of Supercomputing '89*. Pp. 96-105.
11. Patterson, D., Gibson, G., and Katz, R., A Case for Redundant Arrays of Inexpensive Disks (RAID). *1988 Proceedings of the International Conference on the Management of Data*. June 1988, pp. 109-116.
12. Pierce, P., A Concurrent File System for a Highly Parallel Mass Storage Subsystem. *Fourth Conference on Hypercubes, Concurrent Computers, and Applications*. March 1989, pp. 155-160.
13. Salem, K., and Garcia-Molina, H., Disk Striping. *IEEE 1986 International Conference on Data Engineering*. 1986, pp. 336-342.
14. Thinking Machines Corporation, Model CM-2 Technical Summary. Thinking Machines Corporation technical report HA87-4, April 1987.
15. Thinking Machines Corporation, The Connection Machine CM5 Technical Summary. Thinking Machines Corporation, 1991.

ERIK P. DEBENEDICTIS has focused his career on research, design, and practical applications of parallel computers. After designing the Cosmic Cube at Caltech in 1981, he went to Bell Labs. There, he received the Gordon Bell award 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. He is now the founder of Scalable Computing, a company formed to develop flexible and easy-to-use parallel computing environments.

DeBenedictis received his Ph. D. in Computer Science from Caltech in 1982, a M. S. from Carnegie-Mellon in 1979, and a B. S. from Caltech in 1978.

JUAN MIGUEL DEL ROSARIO received a B. S. degree in physics math and chemistry in 1989 and an M. S. degree in computer science in 1992, both from the University of San Francisco. Mike is now a graduate student at Syracuse University. His research interests are operating systems and file system for parallel computers.