Parallel I/O: Getting Ready for Prime Time

Dan Reed, University of Illinois
Charles Catlett, National Center for Supercomputing Applications
Alok Choudhary, Syracuse University
David Kotz, Dartmouth College
Marc Snir, IBM T.J. Watson Research Center

International Conference on Parallel Processing
August 15–19, 1994
St. Charles, Ill.

During the *International Conference on Parallel Processing*, held August 15–19, 1994, we convened a panel to discuss the state of the art in parallel I/O, tools and techniques to address current problems, and challenges for the future. The following is an edited transcript of that panel.

Dan Reed

Welcome! I've always been a firm believer that panels where the members talk for the whole time and then leave three minutes for questions at the end aren't very interesting. We're going to try to keep the presentation reasonably brief and leave the rest of the time for you to talk about what issues are important to you, ask questions, voice opinions, and suggest things. Vegetable throwing is optional, but certainly fair game.

So, why are we here? Well, think about the hardware and software configuration you might see on a parallel system—lots of disk arrays, software disk caching, virtual memory, multiple high-speed network interfaces, and a hierarchical file system that involves multiple secondary and tertiary storage devices. What kinds of I/O questions can we answer in this context? The list is pretty short! The list of questions we can't answer is pretty long.

I believe we need to revisit many of the issues that we viewed as closed in the past. A huge flurry of I/O research occurred as part of the classic operating system work in the sixties and seventies. With many of those issues, the same answers may not apply to parallel systems.

To whet your appetites, I wrote down a few questions that sprang to mind while I generated slides. One of them is, what kind of I/O patterns can we expect? Going to a river and counting the number of swimmers each day to decide if you should build a bridge is not a particularly good metric. Looking at what people do now is not always a good predictor of what they would like to do. Most systems' I/O is more limited by the art of the possible than by the art of the desirable. You have to determine what people want to do as well as what they can do.

There are other issues related to the kind of support we should provide to application developers. For example, do we need language interfaces so that people can specify how to distribute data across storage devices? Should we give people control over caching and prefetching policies? How does the changing balance of network and disk speeds affect things? Likewise, the information superhighway means that there will be and already are lots of distributed data archives.

When we look at storage devices, it's clear that data densities are rising faster than access times are decreasing. That also has important implications. There are already individual applications that have terabyte-size data sets. And your 1-Tbyte disk farm holds a file—one file—so there are tertiary storage problems to face there as well. And, once you start mixing scientific data, compressed audio, and compressed video, a host of interesting real-time constraints arise.

Now let me tell you what we've been doing, and then I'll let the rest of the panel do the same.

I/O CHARACTERIZATION

The first question I posed earlier was understanding application I/O access patterns. There are really two sides to that problem. First, there are the application stimuli—what kind of demand are we seeing—and then there is how the system responds to those stimuli. In the first case you would like to understand what people are doing, and what they might do, and in the second you'd

like to understand how systems are responding to those access patterns. This tells you what is broken—and plenty of things *are* broken—and it shows you both ends of the spectrum so that you can study how to fix it. If you have both stimulus and response when you modify the file system or the application, you can study the "before" and the "after" and see the effects of your file system changes.

We've been retargeting the Pablo environment and building tools to capture application I/O access patterns. Of course, real codes run a long time, with "a long time" measured in hours or days on high-performance machines. Sometimes you can capture a detailed trace of every I/O activity; other times you can't. In fact, with some codes, trying to trace the I/O activity may generate more I/O than the I/O does—that usually indicates that the code wasn't written very well! In other cases, the I/O operations are larger and less frequent. So, it's pretty obvious you need a variety of ways to characterize I/O activity. Sometimes you want to capture the trace; other times you just want a summary.

Let me show you just one I/O characterization example. As part of a collaborative National Science Foundation Grand Challenge project with Caltech, we obtained an electron-scattering code that models low-temperature plasmas.[1] The results of this model are used in semiconductor foundry work. For our purposes, the application's details aren't really important, except that the problem its developers are solving and the problem they would like to solve are vastly different. The kind of problem they solve now might run all night on a 500-processor parallel system. However, the problem they would like to solve might run a week on that machine, with an equally impressive scaling of the I/O.

Here's the bad news. This code is indicative of the problem I mentioned earlier: namely, the difference between what people would do and what they can do. It turns out that the straightforward way to do I/O with this code is to parallelize both reads and writes—it can be completely parallel with very little synchronization. But, the current system software makes that difficult.

Let me show you some performance data to illustrate that (see Figure 1). In this version of the electron-scattering code, all processors attempt to concurrently read some initialization files. Then, they begin a cycle of computation and writes, where a single processor writes data to a group of files. Performance measurements show a great deal of contention when 128 processors are concurrently fighting to open and read the initialization files.

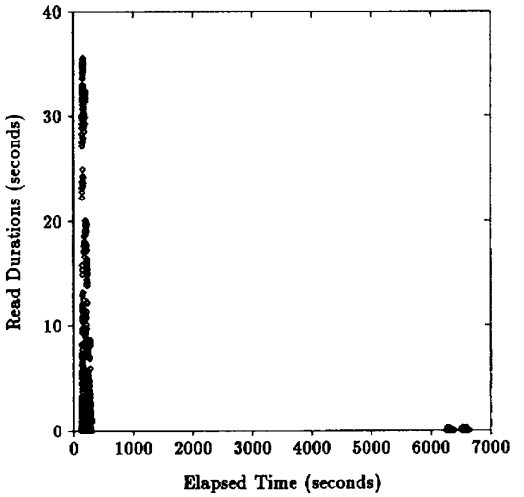

Figure 1. Electron-scattering code read durations.

We showed the application developers this performance data, and they said, "We can fix this," and they did. Having found that reads were a bottleneck, they sequentialized those and parallelized the writes. This fixed the read problem and broke the writes! So, what can you conclude from that? Basically, the things you think would be reasonable don't always work—application developers can spend a great deal of time programming their way around problems rather than doing the science they want to do.

This also has implications for performance instrumentation. Getting performance data is harder than it looks—to get data you are using the same I/O facilities the application developers are using. If they are having trouble doing I/O, you as a performance analyst are having trouble doing I/O too.

From looking at a range of application codes,[2] we've concluded that there is wide variability in I/O access patterns and that performance is highly sensitive those patterns. The software implication is that, in the short to medium term, we need mechanisms that let users control how the system manages I/O, how it distributes data across the storage devices, when and what it prefetches, and what caching algorithms it uses.

Implications

Let me conclude with three thoughts. First, many of the problems are economic, not technical. I don't mean to minimize the significance of the technical problems. They are real, they are severe, and they deserve lots of work, but many of the problems are economic. In an academic context it is possible to work on I/O problems, but it is difficult because you need access to real systems—the problems of interest are problems of large scale.

The second is really a variation on what I have, with some self-aggrandizement, taken to calling "Reed's Law": the more you are willing to pay for a system, the more limited the software options are. This observation relates to the size of the user base. If you buy a PC you can go to a mall and buy all the software you want at $50 a copy. If you are willing to spent $30 million for your machine, you will probably have to write most of the software yourself. That is just the hard economic reality: at the high end, economics drive the software's capabilities in some fairly significant ways. This makes collaboration among academic researchers, government laboratories, and vendors especially important.

Finally, some of us, including several people on this panel, are involved in a new project called the Scalable I/O Initiative. It consists of a group of operating systems, language, performance analysis, and applications researchers who are mounting a concerted attack on the I/O problem, looking at it from end to end. There are two major testbeds—the Intel Paragon XP/S and the IBM SP-2—but other vendors are involved as well. On that note, I'll let Charlie Catlett talk.

(For information on performance

instrumentation and the Scalable I/O Initiative, access http://www-pablo.cs.uiuc.edu/ and http://www.cgsf.caltech.edu/SIO/SIO.html.)

Charles Catlett

I would like to talk about three general application areas where the National Center for Supercomputing Applications has encountered I/O problems. One is transaction processing, another is bulk transfer, and the third is real-time interaction between computers and between computers and users. In these areas, we want to see improvement in I/O from a host-interface and an operating-systems standpoint.

Transaction processing

The enormous growth of the World Wide Web and of NCSA's Mosaic browser is evident in the growth of Web traffic on the Internet and at the NCSA server, the busiest Web server on the Internet. Since Mosaic's release in early 1993, Web traffic on the Internet has increased exponentially, from under 100 Mbytes to over 1.2 Tbytes per month, with continuing rapid growth. At the NCSA Web server, the number of connections per month has grown from under 500,000 in July 1993 to over 10,000,000 in July 1994.[3] The growth is essentially linear, with no evidence of a plateau in sight. Each connection is a typical TCP connection. Somebody clicks on something with Mosaic or some other browser. The browser then starts a TCP connection to the server, grabs the data, and then closes the connection. The amount of data being transferred varies from a few tens of bytes to approximately 18 Mbytes.

Initially, the NCSA server was a single workstation. At a million connections per month we went to a server-class system. Shortly thereafter we moved to multiple servers in a cluster. We changed it each time because we thought we were running into load problems on machines, in the traditional sense of CPU or I/O load. Because users couldn't connect to the servers, we thought these machines were overloaded. Yet, if we looked at the performance the way we looked at a file server or a compute server, the servers appeared to be very lightly loaded. Actually, a very large number of people were requesting transactions, which were hitting all at once.

The point is that the problem is not I/O in terms of throughput and it is not CPU speed. It is the way the operating system is built, the way the network protocol stack is implemented, and the assumptions implementors made about the number of TCP connections that people would request during a short period of time. Today's systems are not built to take on 50, 60, or 100 requests for TCP connections in a few seconds.

We got around that with some very coarse parallel I/O. We put a cluster in place. We changed the domain name system (the local copy of the BIND software) so that when a Web client wants to connect to our server and goes into the DNS to get the IP address, we perform round-robin distribution among multiple machines. So, we spread the load across these machines in one sense, but because of the caching in the DNS, it is not done on a per-client basis. Rather, the load-leveling round robin occurs on a per-site basis. When a client—say, at Stanford—asks for the IP address of www.ncsa.uiuc.edu, the Stanford DNS system caches that information for other Stanford clients. The local site (in this case, Stanford) determines the length of time that information is cached, but the information sent out also has a recommended time-to-live. We initially used a time-to-live of two hours but eventually lowered that to 15 minutes.

Bulk processing

At the NCSA computing facility, major machines are connected with both FDDI and Hippi switches. In this environment, people do their computation on a supercomputer, store data in the archive system, and then post-process the data using a visualization environment—for example, our Cave Automatic Virtual Environment (CAVE), driven by a Silicon Graphics Onyx system.

Over a month, there are many instances where a user will transfer on the order of 50 Gbytes in or out of the archive during a single day. Bulk transfer is more important as we get to larger data sets. For example, our Thinking Machines CM5 has 140 Gbytes of disk space, or what we call *scratch area*. One of the grand-challenge groups using the CM5 has an application they want to run for 12 hours, and it produces enough data to fill that scratch area several times. Even if that application fills a larger portion of the supercomputer disk space, another application will come later and want to produce a similar amount of data. This means we've got to move the first application's output into the archive in a short time. When we are talking about moving 100 Gbytes, much less a terabyte, bulk throughput becomes very important.

Unfortunately, we actually took a step back in I/O when we went from traditional vector machines to massively parallel machines such as the CM5 or the Intel Delta or Paragon. Back in 1989, when we started thinking about what we could do with gigabit networks, we assumed that the 400-Mbits-per-second user-to-user throughput we were getting between two Cray supercomputers was the starting point. We assumed that as technology advanced, I/O would also improve, or at least remain constant. We were dead wrong. Although we went in five years from a Cray Y-MP that runs a little over a gigaflop to a CM5 that is about 50 times faster, it wasn't until the CM5 was two years old (late 1993) that we could get I/O on it at faster-than-Ethernet rates. Even now with Hippi on the CM5, we are not anywhere near where we were four years ago with the Cray Y-MP. So, compute power has increased, memory capacity has gone way up, and disk farms are much larger, but I/O is still a real problem.

When we consider applications we could attack with high-speed I/O, we face the following dilemma. Back when we were using the Cray Y-MP we had plenty of I/O, but not enough compute power to take advantage of it. Now we have plenty of compute power to take advantage of 400 or 800 Mbps, but we can't get in and out of the machine. When this gets fixed we are really going to be set!

Real-time interaction

On the Blanca gigabit network testbed, our applications typically involve a user sitting

at the University of Wisconsin on a Silicon Graphics machine with a code that runs on a supercomputer at NCSA in Illinois. In some cases, we want to run part of a distributed application on the SGI at Wisconsin and part of it on an SGI at NCSA. As the pipe's bandwidth increases, we can make tradeoffs between bandwidth and latency—actually hiding latency by clever prefetch schemes, for example. In other cases—for example, the CAVE virtual environment—we want the SGI at Wisconsin to be able to respond in real time to a command given by a user in the CAVE at NCSA. This means being able to get control information through the network at very low latency. This is not to say we think we can do anything about the speed of light, but we certainly want to minimize delays due to queueing or packet/cell loss and retransmission.

FINAL THOUGHTS

Transaction processing is by far the most important of these three areas. Improving transaction processing means fundamentally changing the assumptions made when an I/O system is developed. A global information server on the Internet will be subjected to very different demands than will a local group file server or desktop workstation. Parallel systems are very fast, but they are useless for bulk processing if you cannot get data into and out of them at rates commensurate with their compute capabilities. For real-time interactions between systems over networks, mechanisms that hide latency on wide area networks will be very useful for distributed computing, particularly if they are transparent at the application level.

Alok Choudhary

I will focus on the specific I/O problem of running an application on a single parallel computer. Users demand only high performance, not parallel I/O. Given the technology, hardware-level parallelism is essential. So, I would like to concentrate on the technology's software aspects. The I/O problem is like the federal budget deficit. With the budget, the interest part of the deficit burdens economic growth. Parallel computers can do fast computations, but I/O software is pulling their performance down.

What are the solutions? How do we achieve them? Hardware solutions alone do not work, if we take lessons from cache memory or virtual memory on uniprocessors. Software support—for example, compiler and runtime support—is necessary. For example, to utilize the cache better, a compiler has to do a lot of work.

Improving transaction processing means fundamentally changing the assumptions made when an I/O system is developed.

To achieve high-performance for I/O in parallel computers, we have to break away from traditional sequential views such as "stream of bytes." We use that view for convenience. If you have the sequential view, you can say something about the relationship between two different elements in your file. Dan made this point earlier. Access information must flow from a program to the data-management (for example, files) system. Information must flow from the application program to the data-management system, and back. The hardware underneath has fast interconnect and a lot of disks. The problem is the layer of software on top of the hardware. To exploit the hardware's available bandwidth, the software must use the access-pattern information.

The current implementation of file systems is a naive extension of sequential files. Access and prefetching do not consider any information about interleaved access patterns by different processes. This results in a lot of thrashing, lost bandwidth, and poor performance. Therefore, there must be some way for this access-pattern information to flow to the file (and the runtime) system. That is, a notion of collective accesses should be built in. At the same time, the interface should not be very rigid for this information flow. It should be flexible enough to easily support different access patterns.

The type of information provided is also important. Users can only specify a certain type of information; for example, in High Performance Fortran-type languages, users provide distribution information. Compilers provide only static-access information based on the computation and distribution. The runtime systems provide dynamic information, and so on. Each level of software needs to incorporate the most suitable access information to obtain high performance at the application level.

At the file system level, we may need metadata describing common access patterns likely to be seen on the data stored in the file. This approach can be extended to I/O in a high-performance distributed-computing environment using a high-speed network. Let's say you have a gigabytes-per-second network, but to communicate on two sides you have to sequentialize the data and then send it over. Obviously, there is a bottleneck where the data is sequentialized. You still want to communicate in parallel even though fragments may go sequentially on this high-speed network. You can achieve this by associating some information with the data. This information specifies how this data is organized, so it can be reassembled on the other side.

I believe that the runtime system has to do the most work because compilation technology is difficult and is useful only for static patterns. We have demonstrated how the runtime system can enhance I/O performance by taking the distribution and access-pattern information into account for scheduling accesses.[4,5] The runtime system takes information from the application about the application's distribution requirements, and information from the file system about the data distribution. Using this information, it performs accesses collectively, rather than letting individual nodes do it all.

This is the "two-phase" access strategy. For a read operation, in the first phase data is accessed according to a conforming distribution. In the second phase, the data is

permuted into appropriate processor, based on the application's distribution requirements. This provides consistent performance, mostly independent of the individual node-access patterns.

In summary, to satisfy users, software support should provide high performance without burdening them with the low-level details of the underlying parallelism.

David Kotz

The preceding panelists discussed either I/O hardware or parallel I/O software. I'll tell you a little about each.

In the past, the problem was that we had no parallel I/O hardware. Now we have adequate parallel I/O hardware designs—for example, separate I/O nodes with RAIDs attached to each node. However, the new problem is that most machines aren't configured with *enough* I/O hardware to meet their needs. I guess people aren't willing to spend enough money. The solution is to face reality and spend the money to buy enough disks and I/O nodes.

As I see it, the real problem is in the software: performance, reliability, and usability.

With current software, as others here have mentioned, you usually get only a small fraction of the full bandwidth. So, even if you do face reality and buy enough parallel I/O hardware to provide the bandwidth you need, you cannot access that bandwidth.

Even if you have RAIDs, which protect against disk failure, the overall system reliability is poor. The problem is that the software is either buggy or not tolerant of hardware faults, and so is not very reliable.

In most systems today, the file system and language interfaces are hard to use. For example, it is very difficult to express complicated mappings of data from the file into program data structures that are distributed among many processor memories. Even if you choose one of the optimal parallel-I/O algorithms developed in the theory community, it is very difficult to express it in a way that translates into high performance. I think Dan said theory and practice clash sometimes.

In addition, today's parallel file systems don't integrate very well with existing systems, network file systems, and archival file systems, so it is not easy to get data in and out of a specialized parallel file system.

The file system interface is an important issue. What do programmers see as a way to get at their files? Most systems have been based on a Unix-like "read," "write," and "seek" kind of interface, but that is very awkward to use. You often need to have each processor calculate seek offsets so that it can read and write its own portions of the file. This constant seeking also tends to stress the file cache. The cache can get requests for different parts of the file from any of the compute nodes at any time. There is little semantic information flow from the application level down to the file system level. All the processor sees is a couple of reads, writes, and seeks—not any higher-level information about what is really going on.

In recent characterizations of real applications we found that this kind of interface leads to programs that make very small requests. For supercomputer applications, the typical request size is on the order of megabytes. In the parallel scientific applications we traced, the typical request size was fewer than 200 bytes. We believe that these patterns arose because programmers were trying to distribute the file data across many processors, using an interface that limited them to contiguous requests. In many patterns, therefore, even though a processor needed a large portion of the file, each request for each contiguous piece of the file was very small. Fortunately, because of the regularity of the data distributions these requests were very regular, with a fixed request size and stride between requests.

Many parallel file systems, like Intel CFS, extend the traditional interface with "I/O modes" to let users express some of these patterns. These extensions are too simple to express what users need, and we found that users never used them. As a result of our studies, we feel that systems need to support strided I/O requests so that programmers can ask for a large (but not necessarily contiguous) amount of data in one request. IBM's Vesta and the proposed nCube file systems provide this capability by letting users set up a mapping function between the file and memories.

Collective I/O interfaces would also help. Collective I/O means all the processes in an application cooperate to make one (collective) I/O request to the file system. If you are using a single-program, multiple-data programming model, generating collective I/O requests shouldn't be too hard. A collective request gives the file system even more information, and lets it do some additional optimization—an example of when synchronization can be a "good thing." (In parallel computing we usually try to avoid synchronization at all costs. However, in some I/O-intensive applications, when the processes get unsynchronized, the cache starts thrashing. If you keep them synchronized, they can take advantage of interprocess locality in the cache.)

It is important to find ways for the language, compiler, runtime system, operating system, and hardware to cooperate to allow semantic information to flow. The best answer to a problem such as parallel I/O is likely to be a holistic solution arising from cooperation among experts at all levels of the system.

One idea that seems to work is to let the disks control the timing of data flow, a technique I call disk-directed I/O. Memory is a random-access device: it doesn't care in what order you do the transfer. Disks are not really random-access: their performance depends a lot on the order in which you do accesses. So, let the disks decide the order of accesses, and make the memory respond to random transfer requests. The disks can also set the pace according to what is convenient for their buffering, rather than trying to let the cache dynamically deal with whatever requests come at it.

As others have said, we should encourage application-specific I/O policies. One size does not fit all here. A generic cache policy isn't going to fit all access patterns. Perhaps we need to program the I/O nodes, which until now have generally been off-limits.

Are we stuck with Unix forever? Unix has a nice file system, but it is really inap-

propriate for all the things we are trying to do with parallel computing.

Should we have file systems at all, or should we have a scientific database or perhaps a persistent-object system?

Reliability is still a problem. There is more to the solution than RAID.

Regarding virtual memory: what we call "out-of-core" programs do not use the file system for persistent storage but rather for temporary storage. Should we have separate mechanisms to support out-of-core programs, or should they continue to use "files"? Should we support demand paging? Again, there is typically one management policy—a mistake. You need to let the programmer or the compiler tell you when to move things in and out. Go back to the fundamental problem here and try to deal with it.

If I/O was the orphan of high-performance computing, graphics and network I/O are the orphans of I/O research.

Here is a URL for an archive of parallel-I/O resources, including pointers to many other research projects, a bibliography, anecdotes about people's use of parallel I/O, and even some sample programs: http://www.cs.dartmouth.edu/pario.html.

Març Snir

Parallel I/O is a multifaceted problem that stretches from hardware to systems and compilers. There is also the issue of external connectivity and how that takes advantage of parallel I/O. I am not going to address all of these issues in nine minutes. I am not even going to address all the work going on at the IBM T.J. Watson Research Center on these issues. I am going to focus on our activity in parallel I/O interfaces to compilers and applications. This is the research group's work on the Vesta parallel file system for the SP machine, which involves five to six researchers. This technology is now moving into the SP product, so a fairly large product-development group is also involved in parallel I/O.

Our perspective on parallel I/O relates to the SP system's structure. This system has a fast switch with many attached nodes, with a flexible node configuration. Compute nodes may have few disks attached and no persistent data on those. But storage nodes can have a large number of disks attached and hold persistent data. Long-term storage of persistent data is outside the SP on some storage or archival system, so gateways are needed. The internal storage is a cache to the external storage, and an import-export interface moves data across.

One moves a file to be stored not on one node, but across multiple storage nodes. One does this because those nodes, which are supporting the parallel file system, should provide access to the file in parallel with high bandwidth from multiple compute nodes, which are doing parallel computation. This is a clear design goal. Parallel I/O should match parallel computing to get high performance; one wants parallel access to one logically shared but physically distributed file.

Another important goal is to avoid a lot of system activity on the nodes doing the computation. One does not want to see system activities happening randomly on compute nodes. In some benchmarks, we found that two to three percent of system activity occurring randomly on the computation nodes can reduce performance by 40% at the parallel-application level. So, off-loading the overhead from system calls to I/O nodes is an important consideration.

So, what is the Vesta file system? Basically, it allows one to create files that are strided across multiple storage nodes. It gives a fair amount of control in describing how many storage nodes store the file, what the basic striding block's size is, and so on. When one opens the file, it gives one control of what is opened. One does not need to open the entire file; one can open a subfile. Basically, one can view a window into the file, so each processor can open a separate subfile, all processors can open the entire file and share it, or anything in between.

Vesta maintains atomicity and serializability of I/O accesses even if they are spread across multiple I/O nodes: conflicting accesses at different nodes will occur in the same order. This happens without locking and without communication between the client compute nodes. Serializability is achieved by a protocol that involves communication between the servers on the storage nodes. The amount of communication between clients and servers is minimal. Basically, for each access from a client to a server there will be two messages for a write and three for a read. The system supports asynchronous I/O, which is essential to off-load I/O overheads to the storage nodes.

With Vesta, one can add more storage nodes to get more I/O bandwidth, until one saturates the switch. We are also working on implementing MPIO, a programming interface on top of Vesta that allows a message-passing style of I/O and that supports collective I/O, which is also important.

I don't want to go into more technical detail. Instead, I would like to share what we learned from this research project and, more important, from our collaboration with a development group. Sure, Unix stream files are the wrong thing for parallel I/O. The sequential nature of stream files really inhibits the optimizations one can do if one relinquishes the Unix semantics for parallel files. But, in a product environment, the idea of having a file system that is not Unix-compatible raises a lot of horror, and not only in a product-development environment. Users want compatibility with existing file systems. So, any deviation from Unix semantics of stream files is extremely painful.

When we worked on the Vesta file system we developed a non-Unix file system, which had a lot of performance advantages. We provided an import-export interface to Unix files, and we provided a lot of the Unix file system functions. But the product development people did not buy that. The product version of Vesta is a mounted file system that supports Unix file system interfaces with no change. The new functions that Vesta introduced are now available through I/O controls, which is messier. But this is a worthwhile tradeoff.

I would also suggest that we stop thinking of I/O as communication between a job and a file. I/O is really communication between one job and another job. It is communication in space when jobs run concurrently and communication in time

when jobs run one after another. This perspective might lead to more innovative approaches to parallel I/O.

Discussion highlights

Reed: We've talked for an hour, so it is your turn to ask questions.

Is there any ongoing work with parallel I/O and persistent-object storage?

Choudhary: Dave Dewitt at the University of Wisconsin is working on persistent-object stores and parallel I/O. At the University of Illinois at Chicago, Bob Grossman is also looking at persistent object-stores. We are discussing with Mark Snir how to provide information to the file system and keep that information around so that we can improve the performance of parallel accesses.

I know there is a lot of work going on in persistent-object stores, but I am not aware of any that is designed to maximize throughput. Were you talking specifically about that kind of work, or simply about general research in persistent-object storage?

Choudhary: We are interested in improving the throughput and the performance of accessing data from parallel-object storage. Of course, a lot of work is just for storage management.

Snir: There are really two issues. First, if a parallel file is not just a stream of bytes, but a structure, then it is a persistent object. In that case you need a persistent-object-management system to manage the file information, making sure that you transform it to the right format.

The more general question is, if you are moving to the brave new world of object-oriented programming, what is parallel object-oriented programming? What does it mean for an object to be distributed? How do you manage this distribution? This last question is more general than I/O, but the real general question is how to combine parallelism and object-oriented approaches, which don't seem to marry well.

Kotz: There is another project, led by Andrew Grimshaw at the University of Virginia, that proposed an object-oriented interface in the file system. The idea was to build different structures for files, and the object methods for reading and writing would have structure-specific caching and prefetching. The Hurricane file system for the Hector multiprocessor at the University of Toronto also had an object-oriented interface.

Will putting the disks or I/O nodes in control of the order of I/O clash with a carefully ordered computation?

Kotz: You use that method when the application wants to read a large amount of data. With current interfaces, you usually must request many small pieces of a file. With a collective I/O-request mechanism, if you make a request to the file system, the file system manages that transfer from the disks to the memory or vice versa in an order that is convenient for the disks. If you want to go beyond that and overlap that I/O with some computation, you have to compromise between the order that is convenient for the disk system and the order that is convenient for the computation. That is something I want to consider more.

Reed: Often, we needlessly take our sequential models into the parallel context. With a straightforward extension of sequential access to multiple processors, there are many cases where you really don't care what order your requests come back from the file. The file is just a repository of chunks of data, and you want one of them; you don't care which one you get back. That flexibility lets the file system disk scheduler respond to requests in an order that is efficient for it.

Should the application manage the data layout on the disk?

Choudhary: Most parallel applications, at least those using the SPMD model, exhibit a highly correlated access pattern. Once you have highly correlated access patterns you also have information you can use to perform those accesses. If you provide that information with the data storage, the software can use that information. If you apply this approach to a model where you assume you have a set of independent tasks doing I/O, this won't work. But, for all the scientific and information-processing applications where you have highly correlated access, this would work, in my opinion.

Snir: In a sense we are going back in time. With mainframes of twenty years ago, you had a lot of control over block size, how things were laid out on disk, and how the file system optimized accesses. Now we've moved to Unix, where there is some control, but nobody knows how to exercise it. We are trying to build high-performance systems on top of Unix interfaces, and it doesn't work.

Reed: I agree completely with Marc. We need to go back and think about the control we used to have. I remember writing job control for MVS. At the time I thought it wasn't the most elegant language in the world, but it provided a phenomenal amount of control over data layout on storage devices and how data was accessed. We've come from a Unix world where we think the Unix way is the only way to access data. We need to take our blinders off and look at the problem again.

Can you comment more on the Scalable I/O Initiative (SIO)?

Reed: SIO is based on the recognition that I/O is really the limiting factor for many applications, and that the problem is getting worse rather than better. The project evolved from a fairly substantial number of meetings over two years, and involves 30 to 40 people. SIO brought together enough people with a broad range of expertise so that it wouldn't suffer from single-person myopia. It involves compiler experts, operating-systems researchers, performance analysts, application developers, and parallel system vendors, with the goal of transferring ideas into practice.

The project takes an end-to-end approach, beginning with application and system characterization. This lets us see what people are doing now and understand what they would like to do if I/O were better. Given that information, along with the known problems and a couple of system testbeds, SIO has three goals. First, provide better parallel language and file system support. Second, determine what hardware configurations maximize I/O

performance. Third, involve the applications people in the process to keep the rest of us honest. In the end, we want to emerge with a prototype that can influence future-generation production systems. For additional details, access http://www.ccsf.caltech.edu/SIO/SIO.html.

In languages such as High Performance Fortran, we are accustomed to data-layout specifications that help a particular program reference memory more efficiently. There is a problem when you try to do that across programs or even across multiple loops in one program. What is optimal for one program or loop might not be optimal for another. What work has been done about

- *transforming one layout into another, based on a requested structure of the accesses, and*
- *finding a way to use knowledge of a file's storage layout to go backward through the compiler and restructure the code's access pattern to make it better match the file layout?*

Choudhary: The first is essentially a communication-scheduling problem. (I am talking about data that is in memory and not about out-of-core redistribution.) There is a lot of work going on in this area; people at Ohio State University, the University of Illinois, Argonne National Laboratory, and other places have developed techniques to redistribute the data from one distribution to another.

The second question was, are there techniques that let compilers determine the distribution of the file on disks and restructure the accesses in a program to better match the data layout on the disks? You must be assuming you know the file layout when your program is compiled; that is not necessarily true. The data may come over the network, or you may not have control over how the data arrives. The way the data arrives might depend on how efficiently you can do that part of the communication. However, at runtime you can restructure your accesses if you know the layout of the file.

Kotz: There is a bunch of work on out-of-core algorithms for permuting data on secondary storage to improve the performance of future accesses. Depending on the kind of permutation, there are different time bounds on its cost. Most of that work is not known outside the small community of theoretical parallel-I/O algorithms people, but information is spreading. You can find information on these techniques at the URL I mentioned earlier. I know of at least one project that is trying to make a compiler recognize permutations and pick the right algorithm from the known set of optimal algorithms.

References

1. C. Winstead and V. McKoy, "Studies of Electron-Molecule Collisions on Massively Parallel Computers," in *Modern Electronic Structure Theory*, D.R. Yarkony, ed., World Scientific, Singapore, 1994.
2. P.E. Crandall et al., "Characterization of a Suite of Input/Output-Intensive Applications." To obtain a copy, contact Dan Reed at reed@cs.uiuc.edu.
3. T.T. Kwan, R.E. McGrath, and D.A. Reed, "User Access Patterns to NCSA's World Wide Web Server." To obtain a copy, contact Dan Reed at reed@cs.uiuc.edu.
4. R. Bordawekar, J. Del Rosario, and A. Choudhary, "Design and Evaluation of Primitives for Parallel I/O," *Proc. Supercomputing '93*, IEEE Computer Society Press, Los Alamitos, Calif., 1993, pp. 452–461.
5. A Choudhary et al., "Passion: Parallel and Scalable Software for Input-Output," Tech. Report CRPC-TR94483-5, Center for Research on Parallel Computation, Rice Univ., Houston, Tex., 1994.

Dan Reed is a professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign, where he holds a joint appointment with the National Center for Supercomputing Applications. He has authored a plethora of research papers on algorithms, architectures, and performance evaluation techniques for high-performance computing. He serves on the boards of *IEEE Transactions on Parallel and Distributed Systems*, *Concurrency Practice and Experience*, and the *International Journal of High-Speed Computing*. He is also the treasurer for ACM Sigmetrics and a member of the NASA RIACS Science Council. He received the 1987 National Science Foundation Presidential Young Investigator Award. He received his BS in computer science from the University of Missouri at Rolla in 1978 and his MS and PhD, also in computer science, from Purdue University in 1980 and 1983. He can be contacted at reed@cs.uiuc.edu.

Charles Catlett is the associate director for computing and communications at the National Center for Supercomputing Applications, where he is responsible for the strategic planning, architecture, installation, and management of a national supercomputing facility and local high-performance computing environment. He is also a principal investigator on the Blanca gigabit testbed and a member of the coordination committee for the ARPA/NSF national program in gigabit technology. He received his BS in computer engineering from the University of Illinois in 1993. He can be contacted at catlett@ncsa.uiuc.edu.

Alok Choudhary's biography can be found on page 39.

David Kotz is an assistant professor of computer science at Dartmouth College. His research interests include parallel operating systems and architecture, multiprocessor file systems, transportable agents, single-address-space operating systems, parallel computer performance monitoring, and parallel computing in computer-science education. He received his AB in computer science and physics from Dartmouth College in 1986, and his MS and PhD in computer science from Duke University in 1989 and 1991. He is a member of the ACM, the IEEE Computer Society, and Usenix. His e-mail address is dfk@cs.dartmouth.edu

Marc Snir is a senior manager at the IBM T.J. Watson Research Center, where he leads research on scalable parallel software and on scalable parallel architectures. He led the initial design and prototyping of the parallel software for the IBM SP1 and SP2. He coauthored the High Performance Fortran and the Message Passing Interface standards. He worked on New York University's Ultracomputer project from 1980–1982. He has published on computational complexity, parallel algorithms, parallel architectures, interconnection networks, and parallel programming environments. He received his PhD in mathematics from the Hebrew University of Jerusalem in 1979. He is a member of the IBM Academy of Technology, a senior member of IEEE, and a member of ACM and SIAM. He can be reached at snir@watson.ibm.com.