INPUT-OUTPUT PROGRAMS

INTRODUCTION

Individual processor speeds are increasing at a very high rate both in the commercial and scientific arenas of the computing worlds and are consumed by highly demanding large-scale applications. These large-scale applications also store, retrieve, and process huge quantities of data, which in turn emphasize the need for a powerful input/output (I/O) subsystem. Unfortunately, advances in I/O subsystem technology have not kept pace with those of the processors, leading to poor overall performance of I/O-intensive applications. Database processing, climate prediction, computational chemistry codes, and computational physics codes all perform I/O intensive operations and make optimizations and tuning to the I/O subsystem a necessity. Metacomputing is a fast emerging area where heterogeneous platforms process and feed data into one another; usage of resources scattered over the Internet, and using supercomputers located in different geographical locations but connected through high-speed networks to solve the same application are some examples of this environment. In addition to requiring high-speed networks, most of such applications also need a high-performance I/O subsystem. High-performance visualization systems and multimedia applications require archival storage on the order of terabytes and high bandwidths in their I/O subsystems. In addition to scientific applications that have high I/O requirements, legacy codes used in the commercial sector have surging storage and bandwidth requirements. Data mining and data warehousing are fast becoming important commercial sector application areas. Along with the current changes in Web technology and accompanying development of programming languages like Java, the commercial sector is pushing the limits of performance of I/O subsystems. Small-scale parallel machines such as symmetric multiprocessors (SMPs) built with a small number of commercial off-the-shelf microprocessors are permeating the commercial market, thereby making parallel processing a feasible alternative. Lessons learned and progress made in bridging the gap between the processor and I/O performance can benefit all these areas.

It is well known that different types of programs have different I/O requirements. Although in recent decades both storage capacity and speed of I/O hardware have increased considerably, these enhancements still lag far behind the central processing unit (CPU) and memory system performance. So it is imperative that software systems aimed at optimizing I/O behavior of applications be developed. After giving a brief overview of I/O architectures, in this article we present an overview of approaches with which to handle I/O from within programs. Due to the size of the applications considered, we will concentrate on parallel architectures and parallel software rather than sequential machines.

I/O HARDWARE

Sequential machines have simple I/O architectures: an I/O device connected to the CPU via a controller. For parallel architectures, however, the I/O subsystem can be more sophisticated. We present two representative examples for the I/O subsystem. In the architecture shown in Fig. 1(a) each processor has its own locally attached disk space. This disk subsystem architecture is very similar to that of a network of workstations with local disks. The disadvantage of this architecture is that the only way to share the disk resident data is through explicit communication. This means that the data will be first read by the owner processor (source) and then will be communicated to the requesting processor (destination) over the interconnection network. The obvious advantage is that the I/O accesses to local disk(s) are relatively fast.

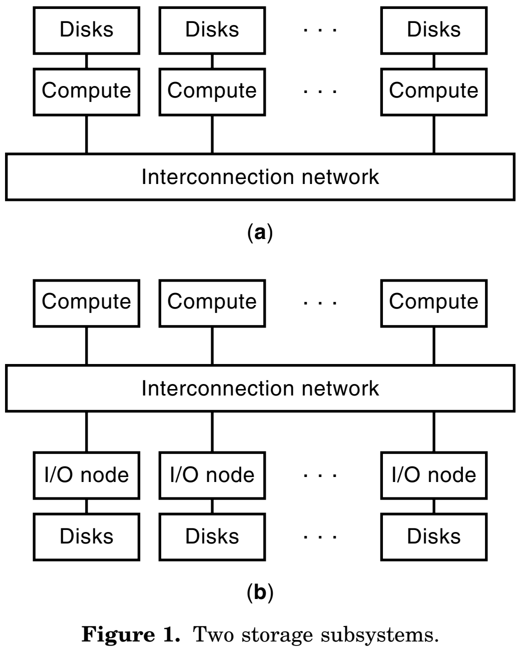

Figure 1. Two storage subsystems.

The architecture shown in Fig. 1(b), however, contains dedicated nodes to perform I/O. Essentially the disk space is shared across all compute nodes, resulting in the possibility of sharing data over the disk subsystem without any communication. The problematic issues, such as keeping the shared data consistent across processors and preventing contention on the hot spots in the disk area, are the main drawbacks.

Modern parallel systems like those of Intel Paragon and IBM SP-2 generally have hybrid I/O architectures that are some type of combination of Figs. 1(a) and 1(b).

It is known from Amdahl's law that the performance of a computer system is determined by the slowest part of it. The rate of increase in the speed of processors and memory components is much higher than that of I/O subsystems. For example, over the past 20 years disk-related factors such as access times and rotational latencies have improved only minimally. Despite this fact, to address the problems with slow mechanical parts, several new mechanisms, such as arrays of disks, disk caches, and intelligent network interfaces, are being introduced. Although these improvements eventually are reflected in the execution times of applications, we believe that there is much more to be done for the I/O problem from the software side. In the next section we concentrate on software that is related to I/O in some ways. This includes I/O-intensive applications as well as system software to overcome the I/O bottleneck.

SOFTWARE ISSUES

Throughout the years, software designers encountered and handled the I/O problem in several levels, including but not limited to applications, file systems and operating systems, runtime libraries, languages and compilers, multimedia, and databases. Following subsections present information about these areas.

I/O-Conscious Application Programming

Many high-performance applications have huge I/O demands as well as substantial computational requirements. The term *grand challenge* (1) is used to characterize an important subset of these applications whose storage requirements currently reach up to 100 Tbytes. These I/O requirements are bound to increase with the onset of faster processor architectures and I/O hardware as well as new applications. It is not hard to predict that within a reasonable period of time, the I/O requirements of a typical high-performance application will reach the range of a few petabytes. Examples of such applications include programs from computational physics, computational biology, high-performance simulation, climate modeling, data assimilation, and computational fluid dynamics.

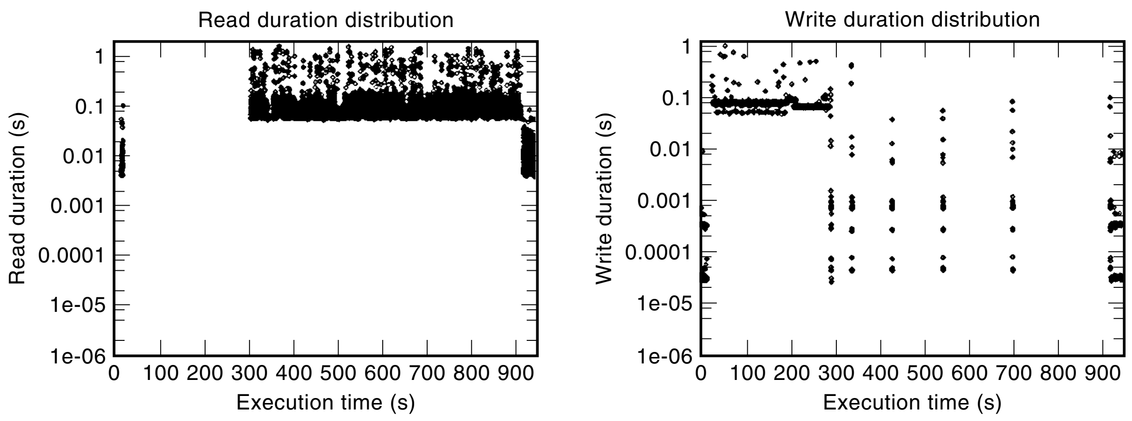

Figure 2. Read and write operation durations of the HF application.

As a point in case, a quantum chemistry application based on the Hartree-Fock (HF) method (2) performs I/O in the order of up to several hundred gigabytes per processor (3). In fact, $\mathcal{N}$ being the number of the basis functions, the I/O performed by this application is on the order of $\Theta(\mathcal{N}^4)$. Moreover, the type of I/O access varies across different phases of the applications. Figure 2 shows the read and write duration distribution of the application across the execution time for $\mathcal{N} = 118$. Table 1 presents the I/O summary of the application, whereas Table 2 shows the read and write size distributions for the same input. Three different phases can easily be identified from Fig. 2. Initially (first 50 s) there are small reads and writes for the purpose of initialization of some database files as well as application parameters. The second phase (between 50 and 300 s) consists mainly of large write operations to data files on the disk. Following this, the third and last phase consists of large read operations and itself comprises of several subphases, corresponding to the iterations in the program. Also in this phase a few write operations to runtime database files are performed. Since the second and third phases constitute the main bottleneck for the program, the programmers use application-level memory buffering to perform I/O operations in large chunks instead of many small reads/writes. In general, the reason that an application performs I/O can be a combination of the following:

- *Data Storing/Retrieval.* Many high-performance programs store/retrieve large amounts of data to/from disk subsystem and/or archival storage. These data may be temporary or persistent. While the temporary data are used for execution of the current application only, persistent data are "alive" across different executions. The main data structures used by the HF application mentioned earlier, for example, fall under the temporary data category. Additionally, the data accessed by an application can be *local* or *remote.* The data resident on the local I/O subsystem are considered local, whereas the data accesses to a remote location over an interconnection network are said to be remote.

  Out-of-core data also fall into this category. Out-of-core applications have data structures that are so large that they cannot fit entirely in the aggregate memory of even the parallel machines. Consequently, data should be staged into memory in smaller chunks called data tiles. As will be explained later, there is some work from the software community on optimizing specifically out-of-core applications.
- *Checkpointing.* Since many applications take several hours or even days for completion, they need to store some information on disk and/or tape to recover in case of system or program failure. Although this process can increase the execution time of the application and put an additional burden on the I/O subsystem, in many cases it is proven to be useful and practical. As an example, an iterative astrophysical hydrodynamics application (4) performs I/O for checkpointing and restarting. In this application, up to six arrays over twenty thousand iterations are written in the checkpointing stage and read during the restarting stage. Overall data transfer from the disk subsystem is on the order of several gigabytes.

Table 1. I/O Summary of the HF Application

Operation	Operation Count	I/O Time (Seconds)	I/O Volume (Bytes)	Percentage of I/O Time	Percentage of Execution Time
Open	19	3.13		0.20	0.08
Read	14,521	1489.07	909,301,536	93.76	39.28
Seek	1,018	17.0		1.07	0.45
Write	2,442	78.01	57,477,540	4.91	2.06
Flush	50	0.44		0.03	0.01
Close	14	0.52		0.03	0.01
All I/O	18,064	1,588.17	966,779,076	100.0	41.9

Table 2. Read and Write Size Distribution of the HF Application

Operation	Size < 4K	4K < Size < 64K	64K < Size < 256K	256K <= Size
Read	646	3	13,872	0
Write	1,572	3	867	0

- *Monitoring/Visualization.* Applications dealing with real-time planetary data, for example, can transmit huge quantities of data that need to be rendered at a rate requiring up to 200 megabytes per frame (5). Since that amount of data far exceeds the I/O capacity of current machines, intelligent programming approaches to optimize the real-time I/O are extremely important.

In general, all I/O-bound applications may need to make use of a combination of secondary (disk) and archival storage.

File Systems

Traditionally, file systems present the user with a high-level interface to access low-level and architecture-dependent I/O routines. At the point where the size of data exceeds the size of virtual address space of the machine, file systems are needed to facilitate the interaction between the I/O hardware and application software.

While the traditional file systems for serial machines have, in general, simple and easy-to-use interfaces, they are oriented toward specific sequential access patterns. The workload studies for IBM mainframes, Unix workstations, and some grand challenge applications as well as other scientific applications show that sequential access patterns are highly regular in terms of both granularity and strides.

Another area of work is distributed file systems, where the file services are generally provided by "file servers," which are processes that run on dedicated machines. The main issues in a distributed file system design are access control and authorization, transparency, file naming, and file sharing.

Recently there has been more work on *parallel* file systems (PFSs), which show that simple extensions of Unix-like interfaces for parallel architectures and parallel I/O subsystems are often inadequate and may result in inferior performance. Throughout the years several commercial parallel file systems have been designed and implemented. In the following, we present somewhat detailed information on one example parallel file system.

The PFS (6) is designed to provide the high bandwidth necessary for parallel applications on Intel Paragon. This is accomplished by striping the files across a group of regular Unix file systems (UFSs) that are located on distinct storage devices and by optimizing accesses to these file systems for large transfers. Any number of PFS file systems may be mounted in the system, each with different default data striping attributes and buffering strategies. Stripe attributes describe how the file is to be laid out via parameters such as the stripe unit size (unit of data interleaving) and the stripe group (the I/O node disk partitions across which a PFS file is interleaved). Currently supported buffering strategies allow data buffering on the I/O nodes to be enabled or disabled.

When buffering is disabled, a technique called *fast path I/O* is used to avoid data caching and copying on large transfers. The file system buffer cache on the Paragon OS server is bypassed, as is the client-side memory mapped file support used by default in the UFS file systems. Instead, fast path reads data directly from the disks to the user's buffer and writes from the user's buffer directly to the disks. Also, the file system performs block coalescing on large read and write operations, which reduces the number of required disk accesses when blocks of the file are contiguous on the disk.

The Paragon PFS provides a set of file access modes (Fig. 3) for coordinating simultaneous access to a file from multiple application processes running on multiple nodes. These modes are essentially *hints* provided by the application to the file system that indicate the type of access. These hints allow the file system to optimize the I/O accesses based on the desired file layout, the degree of parallelism, and the level of data integrity required. The I/O mode can be set when a file is opened, and the application can also set/modify the I/O mode during the course of reading or writing the file.

The various I/O modes are as follows:

- `M_UNIX` is the default mode for sharing files and conforms to the standard UNIX file sharing semantics for different processes accessing the same file. Each node that shares the file maintains its own file pointer, and there is no synchronization between the nodes. The nodes access variable-length and unordered records.
- In the `M_LOG` mode, all nodes that share a file use the same file pointer. The node accesses are not synchronized. The nodes can access variable-length and unordered records.
- In the `M_SYNC` mode, all the nodes sharing a file use the same file pointer and the node accesses are synchronized. The node accesses are always satisfied in the node order. Node ordering is used to synchronize the node accesses.
- In the `M_RECORD` mode, all the nodes that share the file have unique file pointers and the nodes are not synchronized. Nodes access fixed-length records, and files created in this mode resemble files created in the `M_SYNC`

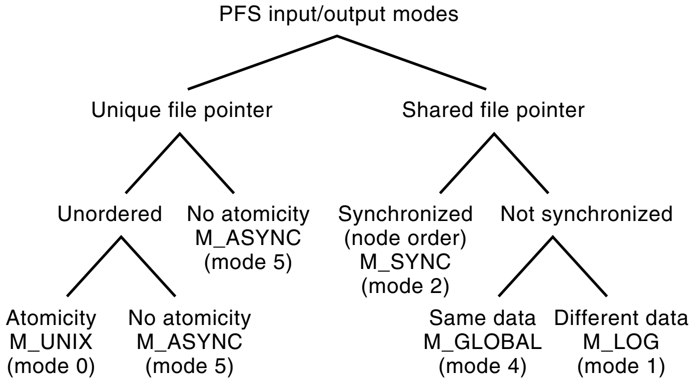
PFS input/output modes
Unique file pointer
Shared file pointer
Unordered
No atomicity M_ASYNC (mode 5)
Synchronized (node order) M_SYNC (mode 2)
Not synchronized
Atomicity M_UNIX (mode 0)
No atomicity M_ASYNC (mode 5)
Same data M_GLOBAL (mode 4)
Different data M_LOG (mode 1)

Figure 3. Paragon parallel file system I/O modes.

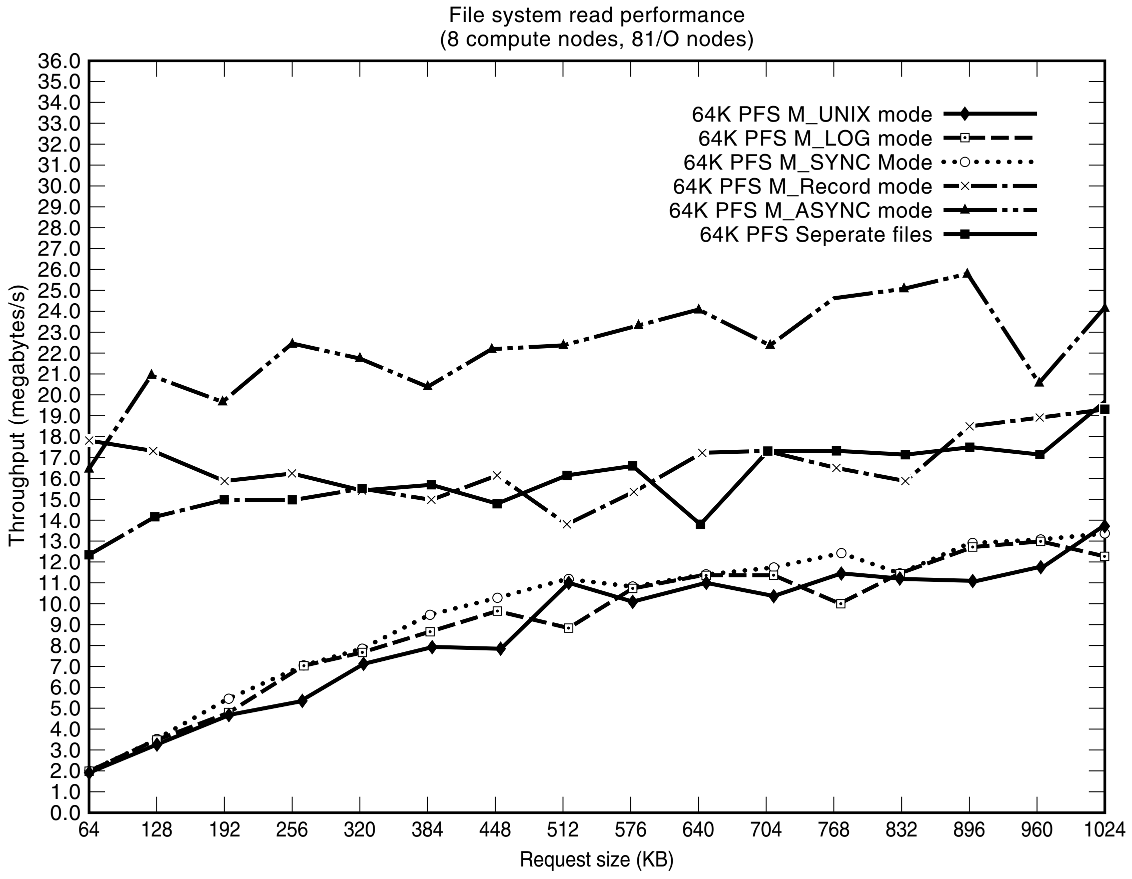

Figure 4. Read performance of the PFS I/O modes.

mode (that is, the data appear in node order). However, this is a highly parallel mode that can allow multiple readers and multiple writers because all the nodes read/write from/to distinct parts of the file. Since all the nodes can access the file in parallel, it offers better performance than M_UNIX, M_LOG, and M_SYNC modes (6).

- In the M_GLOBAL mode, all the nodes that share the file have the same file pointer and all the nodes access the same data. Instead of accessing the disk individually for each request, this mode coalesces the multiple identical I/O requests to the same file into a single request and increases the throughput for the user.
- The M_ASYNC mode is very similar to the M_UNIX mode, except that multiple readers and multiple writers are allowed, which implies that standard UNIX file sharing semantics for different processes accessing the same file are not preserved. I/O operations are not guaranteed to be atomic.

Figure 4 displays the read performance of most of the various PFS I/O modes supported by the PFS. These results were obtained on a Paragon with eight compute nodes and eight I/O nodes, with all compute nodes reading a single shared file. Each I/O node was configured with a single SCSI-8 (Small Computer Systems Interface) card and RAID array; it should be noted that SCSI-16 hardware is also available that effectively quadruples the bandwidth available on each I/O node. In the graph, data for the "Separate Files" case are also presented for comparison with the I/O mode data; in this case each compute node accesses a unique file rather than opening a shared file.

M_RECORD mode read performance is better than those of M_UNIX, M_LOG, and M_SYNC modes. All the nodes in the M_RECORD mode access the file using unique file pointers since they always access separate areas in the shared file. Even though M_ASYNC has the highest performance of all the I/O modes, this mode does not guarantee I/O operations to be atomic.

Other parallel file systems also have similar file access modes. The most recent ones, like PIOFS [], which runs on IBM SP machines, provide the user with logical views (partitioning) of the data in files and support a limited class of collective I/O operations.

A study conducted by Cormen and Kotz (7) has shown that existing parallel file systems have limited functionality. To name a few, some of them cannot give the user access to disk blocks independently; some of them do not offer control over data declustering and stripe attributes; and apparently none of them support user-level different types of data distributions and access patterns. The experiments conducted by Nieuwejaar and Kotz (8) demonstrated that many parallel applications exhibit highly regular but nonconsecutive I/O accesses patterns. Since commercial parallel file systems cannot capture those types of accesses effectively, they proposed some extensions to the standard file system interfaces. The proposed extensions support strided, nested-strided, and nested-batch I/O access requests.

In general, a file system fulfills one of the main functionalities of the operating system. Among the other responsibilities of operating systems is transfer of data between protection domains. Unfortunately, many operating systems are inefficient in transferring large amounts of data across different domains. The main problem is that they introduce unnecessary copy operations, which in turn degrade the performance (in many cases significantly). Container shipping (9) is a new technique for efficient data transfer between domains and involves no physical copying.

Runtime Systems and I/O Libraries

In general, file systems are difficult to use as they are bound to several I/O parameters that are dependent on the underlying architecture. In comparison, runtime libraries are attractive development environments for both users and compiler writers as they offer a level of insulation from the operating system and file system software.

There is some work on developing runtime libraries that provide a number of functionalities to perform I/O in sequential as well as parallel applications. In particular, recently there have been a number of projects on parallel runtime libraries and systems. First we mention some of the optimizations performed by these libraries and then we present an overview of some of the current projects.

Collective I/O. Data parallel programs, where all processors perform similar operations on different data sets, constitute an important class of programs in the scientific community. If all processors perform I/O independently, the result may be a large number of low granularity requests that may arrive from different processors in any order. Instead, processors can cooperate in reading and writing data in an efficient manner. This process is known as collective I/O.

Data Reuse. It has been observed that in many applications, a portion of the current data set fetched from the disk is also needed for computation on the next data set. Instead of reading the data again, they can be reused by caching the data in either the client or server side.

Prefetching. The time taken by a program can be reduced if it is possible to overlap computation and I/O in some fashion. *Asynchronous* or *nonblocking* I/O calls supported by several file systems provide this capability. Data prefetching is achieved by issuing an asynchronous I/O read request for the next data set immediately after the current data set has been read. In parallel machines, it may also be possible to overlap communication, computation, and I/O. Notice that in comparison with data reuse, prefetching does not eliminate or reduce I/O latency, but rather hides it. Figure 5 shows how the I/O reads are eliminated if prefetching is used for the HF application mentioned earlier.

Current Projects. PASSION (Parallel and Scalable Software for Input-Output) (10), SOLAR (Scalable Out-of-Core Linear Algebra Computation Library) (11), Jovian (12), and PANDA (13) are runtime libraries that have been developed for out-of-core and/or I/O-intensive applications. They provide software support for performing I/O accesses from the user program with a high-level interface using the native file system calls of the parallel machine. Most of the libraries provide support for some subset of major optimizations, such as prefetching, overlapping, reuse, data sieving, collective I/O, disk-directed I/O (14), two-phase I/O (10), and buffering.

Disk-directed I/O (14) allows the disk servers to determine the flow of data for maximum performance. The simulation results show that impressive performance gains are possible.

The PASSION library (10) performs collective I/O using a two-phase method. In this method, I/O is performed in two phases. In the first phase, processors cooperate to read data in large contiguous chunks, whereas in the second phase data are redistributed among processors using the interprocessor communication network available. Two main advantages of the two phase method are high-granularity data transfers and use of innerconnection networks instead of the I/O network, which, in general, has much lower interconnectivity.

PASSION supports the notion of abstract storage models and it classifies it into two types—namely, the local placement model (LPM) and the global placement model (GPM). Each processing node is connected to a separate disk system in the LPM, and data are shared between processors only by communication. Individual processors read or write data from/to the disk through the in-core local array data structure. On the other hand, the GPM supports a global file view where all the processors share a single file that can be accessed by all the processors, called the global array file. The

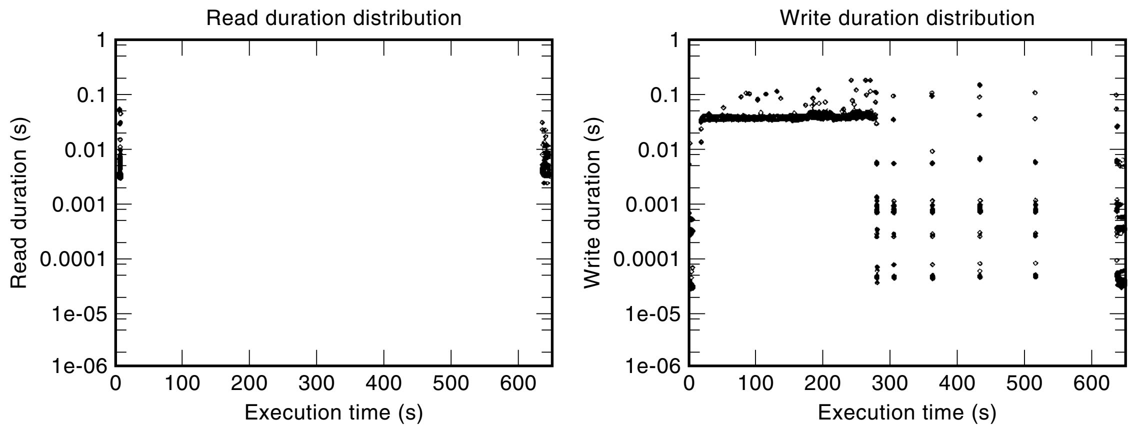

Figure 5. Read and write operation durations of the HF application with prefetching.

I/O routines that are used by the various PASSION library calls are implemented using the native parallel file system's I/O calls. PASSION also performs I/O optimizations such as prefetching, data sieving, data reuse, overlapping, different array layouts, and collective communication for optimizing the total time spent in I/O. Data sieving results in large grain and parallel I/O transfers that are preferable in I/O systems. Data reuse promotes the reuse of data that are already present in memory and fetching only the absent data. Prefetching uses the asynchronous I/O support of the parallel file system to overlap computation with I/O.

SOLAR (11) is primarily designed to handle out-of-core dense matrix computations providing out-of-core functionality similar to the in-core BLAS and LAPACK for shared memory machines and the in-core ScaLAPACK for distributed memory machines. A MIOS (matrix input output subroutine) is associated with each matrix created and it identifies a "primary block"—aligned accesses that yield parallel reads/writes to the disks and achieve the best possible I/O bandwidth. Two-phase I/O and disk-directed I/O are used by the MIOS routines to perform efficient I/O accesses. Multiple layouts and pipelined accesses are supported.

Collective I/O is the emphasis of Jovian (12), where all the I/O nodes cooperate and coordinate to perform the I/O accesses to access the disk with fewer coalesced accesses. The researchers (12) define "coalescing processes" that are responsible for distinct parts of the global data structure that is stored in the I/O system and are analogical to the server processes in database management systems. The I/O accesses create one-to-one or many-to-one mapping between the application processes and coalescing processes. The researchers present two different views for the collective I/O model—namely, the global view and the distributed view. In the global view, the global subset of the out-of-core data structure distributed across the disks is copied to or from the global subset of in-core data structure, which is distributed across the processors by the I/O library. In the distributed view the application process must convert the local in-core data structures into global out-of-core indices before making I/O library calls, which increases the load on the application process.

The PANDA library (13) uses "server directed I/O" that performs large sequential I/O accesses. The compute node clients perform I/O through the server I/O nodes by sending the appropriate I/O access requests. Once the I/O requests are received by the server I/O nodes, they are performed collectively with all I/O nodes cooperating with each other.

In addition to work in I/O libraries and runtime systems, there has been a significant effort both from the academic and vendor communities to standardize the I/O library calls. Each parallel machine has its own native I/O library and parallel file system that prohibits portable applications. The MPI-IO standard (15) addresses portability and efficiency in developing applications that perform I/O and provides a high-level interface to the application programmer hiding the underlying complicated details of the parallel machine. Similar to file systems that optimize the underlying disk accesses for various patterns of I/O accesses, MPI-IO allows expression of data partitioning across the processors using MPI file types. These are patterns in a file that are replicated in the entire file and are used to tile the file data. File types are built from some basic data types, and they could contain holes or blanks as part of them that can enable multiple processors to share the file with different file types. Unlike the file systems' I/O modes, multiple patterns can be used simultaneously on the same file and highly out-of-order, flexible, and portable patterns can be optimized for using the MPI-IO interface. MPI-IO interface also provides global and individual file pointers and asynchronous I/O accesses. It can allow for overlap of communication or computation with I/O. It provides communicator groups for global data accesses where the accesses are done in a collective fashion.

Language Support and Compilers

Despite the fact that the parallel file systems and runtime libraries for out-of-core computations provide considerable I/O performance, they require a considerable effort from the user as well. As a result, the user-optimized parallel I/O-intensive applications consume the precious time of the programmer, who instead should focus on higher aspects of the program, and are not portable across a wide variety of parallel machines.

In this subsection, we concentrate on compiler techniques to optimize the I/O performance of scientific applications. In other words, we give the responsibility of keeping track of data transfers between disk subsystems and memory to the compiler. The main rationale behind this approach is the fact that the compiler is sometimes in a position to examine the overall access pattern of the application and can perform I/O optimizations that conform to application's behavior. Moreover, a compiler can establish a coordination with the underlying architecture, native parallel file system of the machine, and I/O libraries so that the optimizations can obtain good speedups and execution times. An important challenge for the compiler approach to I/O on parallel machines is that the disk use, parallelism, and communication (synchronization) need to be considered together to obtain a satisfying I/O performance. A compiler-based approach to the I/O problem should be able to restructure the disk resident data and computations, insert calls to the parallel file systems and/or libraries, and perform some low-level I/O optimizations.

As compared with the compilation techniques designed to optimize memory performance, designing compiler techniques to optimize I/O performance is more difficult. To elaborate more on the difficulty of designing efficient compiler optimizations, let us consider an I/O-intensive data parallel program running on a distributed memory parallel machine. The primary data sets of the program will be accessed from files stored on disks. Assume that the files will be striped across several disks. We can define four different *working spaces* (16) in which this I/O-intensive parallel program operates: a *program space,* which consists of all the data declared in the program; a *processor space,* which consists of all the data belonging to a processor; a *file space,* which consists of all the data belonging to a local file of a processor; and a *disk space,* which contains some subset of striping units belonging to a local file. An important challenge before compiler writers for I/O-intensive applications is to maintain the maximum degree of locality across those spaces. During the execution of I/O-intensive programs, data need to be fetched from external storage into memory. Consequently, the performance of such a program depends mainly on the time required to access data. To achieve reasonable speedups, the compiler or user needs to minimize the number of I/O accesses. One way to achieve this goal is to transform the program and data sets such that the localities between those spaces are maintained. This problem is similar to that of finding appropriate compiler optimizations to enhance the locality characteristics of in-core programs; but due to the irregular interaction between working spaces, it is more difficult. To improve the I/O performance, any application should access as much consecutive data as possible from disks. In other words, the program locality should be translated into spatial locality in disk space. Since maintaining the locality in disk space is very difficult in general, compiler optimizations attempt to maintain the locality in the file space instead.

Early work on optimizing the performance of I/O subsystems by compilation techniques came from researchers dealing with virtual memory issues. The most notable work is from Abu-Sufah et al. (17), which deals with optimizations to enhance the locality properties of programs in a virtual memory environment. Among the program transformations used are loop fusion, loop distribution, and tiling (page indexing).

More recent work has concentrated on compilation of out-of-core computations using techniques based on explicit file I/O. The main difficulty is that neither sequential nor data parallel languages like High-Performance Fortran (HPF) provide the appropriate framework for programming I/O intensive applications. Work in the language arena offered some parallel I/O directives (18) to give hints to the compiler and runtime system about the intended use of the disk resident data. Since implementation of these language directives strongly depends on the underlying system, there has been no consensus on what kinds of primitives should be supported and how. There are generally two feasible ways to give compiler support to I/O intensive programs: (1) using parallel file systems, and (2) using parallel runtime libraries. The research has generally concentrated on using runtime libraries from a compilation framework (18).

An I/O-intensive program can be optimized in two ways:

- Computation transformations (19,20)
- Data transformations (21)

The techniques based on computation transformations attempt to choreograph I/O, given the high-level compiler directives mentioned previously. The computation transformations used by compilers for handling disk resident arrays can roughly be divided into two categories: (1) approaches based on tiling, and (2) approaches based on loop permutations. Unlike in-core computation, where main data structures can be kept in memory, in I/O-intensive applications tiling is a necessity. The compiler should stage the data into memory in small granules called data tiles. The computation can only be performed on data tiles currently residing in memory. The computation required for other data tiles should be deferred until they are brought into memory (19). By using the information given by directives, the compiler statically analyzes the program and performs an appropriate tiling. After tiling, the compiler has to insert the necessary I/O statements (if any) into program. Another important issue is to optimize the spatial locality in files as much as possible. This can basically be performed by permuting the tiling loops in the nest. Alternatively, permutation can be applied before the tiling transformations are performed. Given the fact that the accesses to the disk are much slower than accesses to processor registers, cache memory, and main memory, optimizing spatial locality in files to minimize the number as well as volume of the I/O transfers is extremely important.

Although for many applications transformation based on reordering computations is quite successful, for some applications in order to obtain the best I/O performance, data in files should also be redistributed (21). Unfortunately, while the computation transformations can benefit from the work that has been done for cache memories, there has not been much interest on data transformations until recently. This is especially true for disk resident data consumed by parallel processors. The main issue here is to reach a balance between optimizing locality and maintaining a decent level of parallelism. More advanced techniques requiring unified data and computation transformations are necessary if future compilers for I/O-intensive applications are to be successful.

Of course, all the compiler transformations performed to optimize disk performance of I/O-intensive programs should be followed by techniques for optimizing the accesses to data tiles currently residing in memory. Fortunately there are lots of efforts in academia for optimizing the main memory and cache performance (22,23).

To see the effect of the transformations, let us consider Fig. 6, which is obtained on different numbers of processors on Intel Paragon for a simple program that uses 128 megabyte arrays. Slab ratio corresponds to the ratio of size of the local memory to the total size of the out-of-core local arrays. Each figure shows four bars for each slab ratio. The bars correspond to the normalized execution time of the unoptimized version, two optimized versions with computation transformations only (assuming column-major and row-major file layouts), and an optimized version that uses both data and computation transformations, respectively. As can be seen from the figure, although optimizations based on computation transformations improve the execution time, the impact of the unified approach is impressive.

I/O in Multimedia

Multimedia systems operate with a combination of information, such as video, voice, audio, animation, and graphics. In addition to requiring large processing power, applications running on these systems also require large storage capability, fast access rates, low latency, and high network bandwidths. Also, continuous video and audio transfers that occur in real time require constant data transfer rates. Multimedia applications include teleconferencing, group working, multimedia electronic mail, and playback applications. The I/O requirements of these applications can be classified into two types: (1) Conferencing applications require very small latencies in delivery, and (2) playback applications need constant real-time I/O throughput.

A request in a multimedia server undergoes three phases—namely, disk, processor-I/O bus, and processor. The first two phases fall under the purview of the I/O subsystem. When I/O accesses from the application go through the SCSI bus to the disk with deadlines, the disk employs various scheduling algorithms to order the requests for service. Deadline is the total time taken to release a request and the period of the request. EDF (Earliest Deadline First) is a traditional scheduling algorithm that serves disk accesses based on their deadlines. This could result in high seek times and low disk

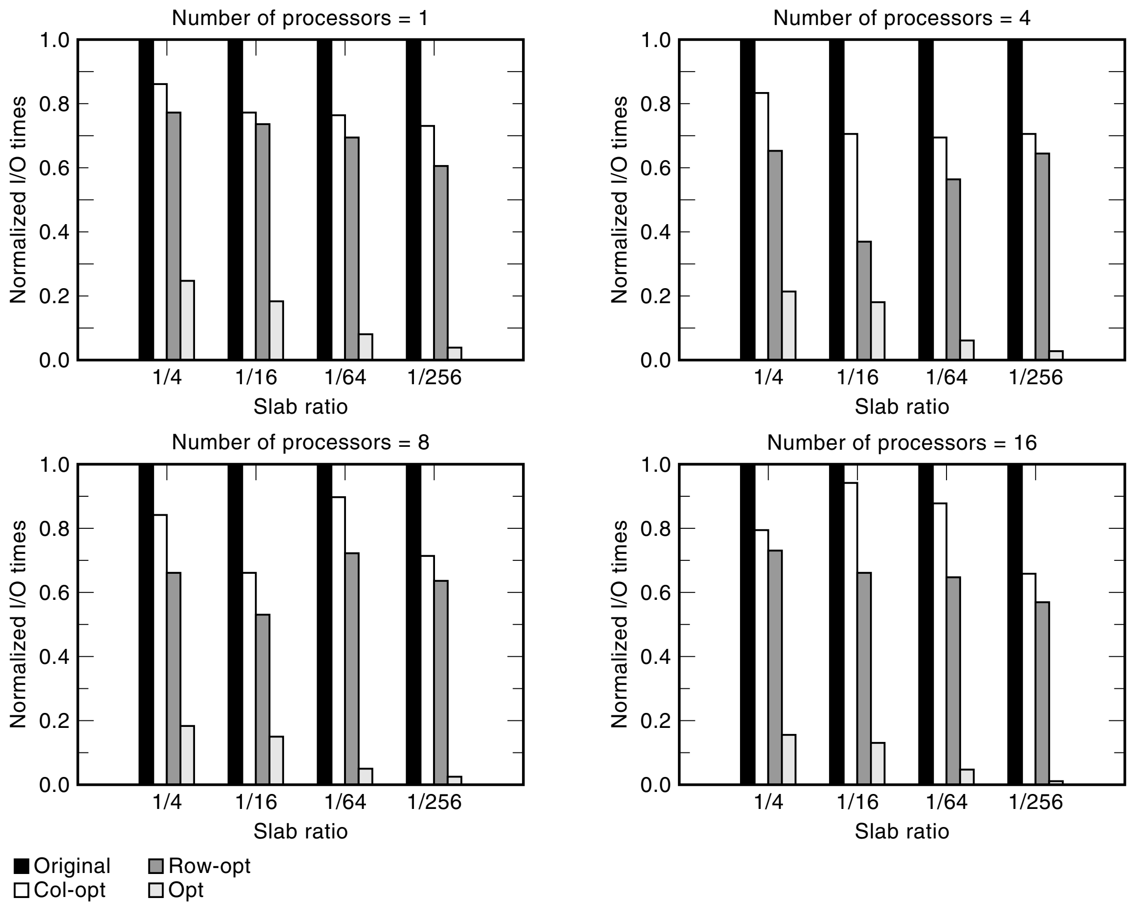

Figure 6. Normalized I/O times for a simple program with 128 megabyte arrays on Intel Paragon.

utilization. This algorithm also assumes that disks are preemptable (current disks are not). To serve aperiodic or bursty requests, some number of such requests are given special priority to avoid starvation and thereby provide reasonable real-time responses. CScan (Circular Scan) service requests in the scanning direction of the disk head. If the disk is moving from the outermost track to the innermost track, it services all the requests on its way. The outcome of this policy is seek optimization, but there is no concept of deadlines. Scan-EDF offers the benefits of both EDF and CScan policies. The general policy is to serve requests in the EDF order, and when the requests have the same deadlines, CScan policy is used to give seek-time optimization. Thus the technique can be made more efficient by giving several requests the same deadline. Aperiodic requests are served as in the EDF policy. In Ref. 24 a comparison of the forementioned scheduling policies for an IBM 3.5 inch 2-gigabyte cat disk was performed. The authors found that CScan supports the most number of streams and supports real-time requests better than aperiodic requests due to its predictable seek-time optimization, but that EDF supports the least number of streams as it gives higher priority to aperiodic requests. SCan-EDF supported almost as many streams as the CSan method and at the same time gave good response times to aperiodic requests as the EDF policy. Also of importance is the contention on the SCSI bus, and this can reduce the streams supported by a disk by a factor of 3. Buffering, file system block size, scheduling algorithm, and the bus system affect the overall performance of the multimedia application and need a lot of further investigation.

CONCLUSIONS

In this article we present some of the key software activities to improve the performance of the I/O subsystems (specifically the secondary storage). It is emphasized that the I/O bottleneck can be handled in different layers of software. Application programmers try to optimize the I/O performance of their programs by a combination of I/O-conscious programming techniques and low-level optimizations such as buffering and caching. File systems and runtime systems present similar functionalities to the user and/or compiler. There are several tradeoffs here concerning ease of use and efficiency. We argue that a compiler for I/O-intensive programs may, in some cases, have a global view of the I/O behavior of the programs and restructure it so that a good coordination between I/O hardware and system software is established.

We hope that ongoing research will give us more information regarding the demands placed by a specific program on I/O subsystems, so that software designers can decide where to place a functionality. This will lead to a better defined coordination between application programs and systems software, which in turn improves the I/O performance of the application.

BIBLIOGRAPHY

1. A. Brenner, et al., Survey of principal investigators of grand challenge applications: a summary, *Proc. Workshop Grand Challenge Applications Softw. Technol.,* ANL, Chicago, 1993.
2. A. Szabo and N. S. Ostlund, *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory.* 1st rev. ed., New York: McGraw-Hill, 1989.
3. M. Kandaswamy et al., Optimization and evaluation of Hartree-Fock application's I/O with PASSION, *SC'97 Conf. (formerly known as Supercomputing 1997),* San Jose, CA, November 1997.
4. R. Thakur, W. Gropp, and E. Lusk, An experimental evaluation of the parallel I/O systems of the IBM SP and Intel Paragon using a production application, *Proc. 3rd Int. Conf. Austrian Center Parallel Computat. (ACPC) Special Emphasis Parallel Databases Parallel I/O,* September 1996. Lecture Notes in Computer Science 1127, Springer-Verlag, pp. 24–35.
5. J. M. del-Rosario and A. N. Choudhary, High performance I/O for parallel computers: Problems and prospects, *IEEE Comput.,* **27** (3): 59–68, March 1994.
6. B. Rullman, *Paragon Parallel File System, External Product Specification,* Santa Clara, CA: Intel Supercomputer Systems Division.
7. T. H. Cormen and D. Kotz, Integrating theory and practice in parallel file systems, *Proc. 1993 Symp. Darthmouth Inst. Advanced Graduate Studies Parallel Comput.,* Dartmouth College, Hanover, NH, 1993, pp. 64–74.
8. N. Nieuwejaar and D. Kotz, Low-level interfaces for high-level parallel I/O, *Proc. IPPS '95 Workshop Input/Output Parallel Distributed Syst.,* pp. 47–62, April 1995.
9. J. Pasquale, E. Anderson, and P. K. Muller, Container shipping: operating system support for I/O intensive applications, *IEEE Comput.,* **27** (3): 84–93, March 1994.
10. A. Choudhary et al., *PASSION: Parallel and Scalable Softw. Input-Output, NPAC technical report SCCS-636,* Sept. 1994.
11. S. Toledo and F. G. Gustavson, The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations, *Proc. 4th Annu. Workshop I/O Parallel Distributed Syst.,* May 1996.
12. R. Bennett et al., A framework for optimizing parallel I/O, *Proc. 1994 Scalable Parallel Libraries Conf.*
13. K. E. Seamons et al., Server-directed collective I/O in Panda, *Proc. Supercomput. '95,* San Diego, CA, December 1995.
14. D. Kotz, Disk-directed I/O for MIMD multiprocessors, *ACM Trans. Comput. Syst.,* **15** (1): 41–74, 1997.
15. P. Corbett et al., Overview of the MPI-IO parallel I/O interface, *Proc. 3rd Workshop I/O in Parallel Distributed Syst.,* IPPS '95, Santa Barbara, CA, April 1995.
16. R. Bordawekar, Techniques for compiling I/O intensive parallel programs, Ph.D. dissertation, ECE Dept., Syracuse University, Syracuse, NY, May 1996.
17. W. Abu-Sufah et al., On the performance enhancement of paging systems through program analysis and transformations, *IEEE Trans. Comput.,* **C-30**: 341–355, 1981.
18. P. Brezany, T. A. Mueck, and E. Schikuta, Language, compiler and parallel database support for I/O intensive applications, *Proc. High Performance Comput. Networking 1995 Europe,* Milano, Italy, 1995, Springer-Verlag.
19. M. Paleczny, K. Kennedy, and C. Koelbel, Compiler support for out-of-core arrays on parallel machines, *Proc. IEEE Symp. Frontiers Massively Parallel Computat.,* February 1995, pp. 110–118.
20. R. Bordawek et al., A model and compilation strategy for out-of-core data-parallel programs, *Proc. 5th ACM Symp. Principles Practice Parallel Programming,* July 1995.
21. M. Kandemir, R. Bordawekar, and A. Choudhary, Data access reorganizations in compiling out-of-core data parallel programs on distributed memory machines, *Proc. Int. Parallel Process. Symp.,* April 1997.
22. M. Wolf and M. Lam, A data locality optimizing algorithm, *Proc. ACM SIGPLAN 91 Conf. Programming Language Design Implementation,* June 1991, pp. 30–44.
23. M. Wolfe, *High Performance Compilers for Parallel Computing,* Reading, MA: Addison-Wesley, 1996.
24. A. L. Narashima-Reddy and J. C. Wyllie, I/O issues in a multimedia system, *IEEE Comput.,* **27** (3): 69–74, March 1994.
25. J. Rosario, R. Bordawekar, and A. Choudhary, Improved parallel I/O via a two-phase run-time access strategy, *Workshop Parallel I/O, Int. Parallel Process. Symp.,* April 1993, pp. 56–69.

Mahmut Kandemir
Meena Kandaswamy
Syracuse University
Alok Choudhary
Northwestern University

INSERTION LOSS MEASUREMENT. See Attenuation measurement.

INSPECTION. See Computerized monitoring.

INSPECTION ALLOCATION. See Corporate modeling.