I/O in Parallel and Distributed Systems

David Kotz Department of Computer Science, Dartmouth College

It has become increasingly apparent that I/O performance, rather than CPU performance, may be the key limiting factor in the performance of future systems. This has led to a growing and systematic study of the *I/O bottleneck* in parallel and distributed systems.

The I/O bottleneck arises for three main reasons. First, while the speeds of CPUs have been increasing dramatically in the past few decades, the speed of I/O devices, being limited by the speed of mechanical components like disks and arms, has been increasing at a much slower rate. For example, while CPU speeds have been increasing at 50-100% per year, magnetic disk access time has decreased by only about one third in ten years, and these trends are likely to remain qualitatively unchanged. Second, in parallel and distributed systems multiple CPUs are employed simultaneously, thus exacerbating this speed mismatch. Finally, new application domains, such as multimedia, visualization, and Grand Challenge problems, are creating ever-increasing I/O demands. Gibson [Gib92] provides a historical review of the I/O bottleneck, as well as a discussion of the underlying technology trends.

The likely growth and severity of the I/O bottleneck for parallel and distributed systems demands attention at all levels of the system design, including applications, algorithms, compilers, operating systems, and architecture. These solutions should be scalable, as the size of the systems and applications grow, and as technology changes. We consider each of these areas in the sections below.

1 Applications

There have been two major application domains where I/O in parallel computer systems has traditionally been found to be a bottleneck. One is scientific computing with massive datasets, such as those found in seismic processing, climate modeling, and so forth [dC94]. The second is databases [DG92].

The I/O bottleneck continues to be a serious concern for scientific computing, particularly Grand Challenge problems, where it is now commonly recognized as an obstacle. Many scientific applications generate 1 GB of I/O per run [dC94], and applications performing an order of magnitude more are not uncommon: applications in computational physics and fluid dynamics are projected to require I/O on the order of 1 TB [dC94]. It seems clear that these total I/O requirements will keep increasing as scientists continue to study phenomena at larger space and time scales, and at finer space and time resolutions. Since the response time that humans can tolerate for obtaining computational results— no matter how comprehensive and detailed— is always bounded, the I/O rates required will continue to increase also. Thus while current applications require I/O rates of tens of MBps for secondary storage, in the near future they will

require I/O rates in the region of 1 GBps for secondary storage [dC94].

A similar trend can be seen in the area of databases, particularly for applications such as data mining [DG92]. New applications, such as mapping the human genome, turn out to involve large-scale database searches on gigabytes of data, and eventually terabytes of data.

Meanwhile, new classes of applications that are rapidly becoming ubiquitous are image visualization and multimedia information processing. It seems likely that multimedia information will be found in many, if not all, computing environments in the future. Multimedia information systems not only impose much higher throughput demands than traditional computer applications (e.g., 81 MBps for HDTV, or 100 MBps for 200 concurrent 0.5 MBps MPEG video streams from a video server) but also introduce additional constraints, such as real-time and synchronized data transfers, not found in the traditional applications.

2 Algorithms

An important consideration for any solution that addresses the I/O bottleneck is that the fundamental algorithms used by applications be asymptotically efficient in terms of the I/O activity they generate.

For sequential computers, the asymptotic efficiency of algorithms is considered in terms of the RAM model of computation. The success of the RAM model is based on its ability to realistically capture the fundamental characteristics of a wide range of sequential machines, while remaining sufficiently abstract to be tractable in the design and analysis of algorithms. The model assumes, however, that the data required by the computation is available in the main memory of the machine before the computation begins. This assumption is reasonable if the memory is large enough to hold the data, or the time required for I/O is small relative to the time required for computation. It has long been recognized that for many applications and system architectures, neither assumption holds, particularly as the size of data sets increases. Furthermore, the RAM model assumes that all memory locations are accessible at equal cost, which is increasingly not true in today's multi-level cache-memory hierarchies. Nonetheless, the RAM model remains a good first approximation for the analysis of many in-core algorithms on sequential machines.

When the application's data do not fit in main memory, however, the situation is quite different. The I/O time may dominate the performance of the application, so the RAM model is a relatively useless tool for analyzing performance. In the following we discuss some models of I/O complexity that have been developed; see Shriver and Nodine [SN96] for more details. We also discuss some of the issues these models raise.

In the *unrestricted parallel model* [AV88] the computer system is modeled as a single CPU connected to a main memory capable of holding M records, which

in turn is connected to an external memory (disk) capable of storing at least the N records that are the data set of the algorithm. Any D blocks, each consisting of B contiguous records, can be transferred in a single I/O operation, and it is assumed that $1 \leq DB \leq M \leq N$. The unrestricted parallel model is a two-level model of memory hierarchy, unlike the single level of memory in the RAM model, and captures two different forms of parallelism in the data transfer: block transfer, since a single I/O operation transfers B records simultaneously, and parallel disk transfer, since D blocks can be transferred simultaneously. Block transfers are quite important in practice, since disk seek times often dominate the total I/O time of a block. The model is termed unrestricted since *any* D blocks can be transferred simultaneously.

The *parallel disk model* [VS90] extends the unrestricted parallel model by requiring the D parallel block transfers to be from D separate disks, in which consecutive blocks are stored on consecutive disks, and an I/O operation can transfer at most one block per disk. The parallel disk model is clearly more realistic than the unrestricted parallel model, which allows the algorithm designer to ignore the critical issue of partitioning and allocating a large data set across multiple disk drives so as to balance disk loads. Numerous algorithms have been developed using the parallel disk model (PDM), from sorting and permutation primitives to computational geometry.

An important factor not captured by I/O complexity models is the ability to overlap I/O with computation. Several techniques have been developed that attempt to obtain this overlap, such as the use of write-behind caching policies, log-structured file systems, user-level data prefetching hints, and language and compiler techniques.

The development of these techniques, as well as the observation of the usefulness of such overlap in practice, underlines the importance of validating the theoretical models of I/O complexity against careful experiments. As noted by Shriver and Nodine [SN96], despite the numerous algorithms that have been developed for these models, researchers have only recently attempted experimental validation of the model or algorithms. Some have found that many applications built using the parallel disk model (PDM) are not I/O bound and thus performance does not necessarily track the PDM parameters, particularly D and B. It was found to be particularly important to overlap I/O and computation. Further experimental work is needed to validate the underlying I/O models and discover the regimes where they are most useful in practice. Hopefully, this experimental work will help to define a new, tractable model that permits I/O, computation, and communication to be considered at the same time, and allow development of algorithms that balance the time taken by these three basic activities. Perhaps a model that allows representation of these different activities to different degrees of precision, to match the characteristics of the problem and the architecture under consideration, would overcome this difficulty.

3 Language and compiler support

In addition to developing specialized I/O-efficient algorithms for fundamental operations such as sorting, it is important to extend parallel compiler technology to automatically generate I/O-efficient code for important classes of applications. Unless programmers are provided higher-level language constructs for specifying the I/O requirements and access patterns of their programs, however, it is difficult and tedious to manually optimize program I/O to take advantage of domain-specific information.

There have been several efforts at extending existing languages to provide constructs that allow a compiler to attempt to optimize the I/O of individual programs. Typically these languages, like High-Performance Fortran (HPF) assume a data-parallel programming model, in which the same sequence of operations is to be applied to all the elements of a large data structure (e.g., an array or vector). The data-parallel programming model is well suited to many types of regular scientific computations.

In data-parallel languages like HPF, the user can issue compiler directives that specify how data arrays are to be partitioned for parallel access by multiple processors. The directives include constructs for specifying common data alignment and access methods (e.g., block, cyclic, block-cyclic). The compiler uses these directives to partition the computation and to generate the appropriate communication and synchronization code for permitting parallel data access. Bordawekar and Choudhary [BC96] provide further details on issues in compiling I/O-intensive problems expressed in data parallel languages.

While the data-parallel paradigm is undoubtedly important and widespread, many parallel applications do not have a regular structure. There has been a lot of activity in designing computational techniques for irregular parallel computations, particularly in scientific applications such as computational fluid dynamics. So far there has been little attention paid to developing compiler support for the I/O behavior of irregular parallel computations. Similarly, high-level parallel languages and programming environments often do not contain support for specifying the I/O activities of the program, although the recent incorporation of I/O constructs into the MPI message-passing standard is promising.

4 Run-time libraries

Perhaps one of the most active areas of software development in parallel-I/O research has been in run-time libraries. Run-time libraries are the quickest way to provide I/O support to a wide range of applications on a wide range of platforms, avoiding the need to modify compilers or operating systems. Several libraries have been developed to support applications wishing to encode algorithms from the parallel disk model, others are designed to support a compiler, , and still others are oriented toward scientific applications in general, or for

specific application domains like computational chemistry. Although few attempt to be standards, these libraries allow the application programmer to take advantage of carefully tuned algorithms and proven techniques.

5 Operating systems

The operating system of a parallel or distributed computer has to strike a delicate balance when it comes to I/O. On one hand, it must provide the programmer facilities that ease the programming task and hide the details and complexities of coordinating and efficiently utilizing the underlying I/O hardware and devices; on the other hand, it must allow the programmer sufficient control to efficiently use the rich resources of the system.

There has been a recent surge of activity in parallel file systems and parallel I/O interfaces that address some of these requirements. One common feature of many parallel file interfaces is that the programmer to specify the access pattern for each file. Typically, this access pattern is specified with one or more parameters (sometimes called *modes*, *filetypes*, or *templates*). Then, when the usual file read and write operations are invoked by multiple processors, the semantics of the operation (and the actual set of file bytes to read and write) are determined by the declared file mode. The most common modes can be classified as

- *Broadcast-reduce,* where all processes collectively access the same data,
- *Scatter-gather,* where all processes collectively access a sequence of data blocks, in order,
- *Shared offset*, where all processes operate independently but share a common file pointer, and
- *Independent*, where the programmer is allowed complete freedom to specify access.

Other studies show that most common parallel file-access patterns can be captured in terms of simple file partitioning schemes and access modes [NKP+96]. Determining the right set of abstractions is still an issue, however. Some file systems like Vesta and Panda allow the programmer to choose parameters that determine the mapping of file data to disks, and file accesses to file data. HFS takes a different approach, allowing the programmer to construct a file abstraction from object-oriented components, such as a distribution component and a replication component. The Galley file system chose a low-level philosophy, in which the file system provides only a low-level interface with highly flexible primitives. Other interfaces and abstractions, such as MPI-2, Vesta, Panda, or HFS, can be built on top of these lower-level interfaces.

Underneath the interface, much of the research in parallel file systems involves techniques for high-performance implementations. Techniques like disk-directed I/O (in which the I/O is scheduled to best suit the disks, with internal data transfer scheduled to match), caching and prefetching, and access-pattern classification (in which the file system observes the file-access pattern, classifies it according to a set of common patterns, and then applies the policy appropriate for the matching pattern). These techniques can often lead to orders of magnitude better performance.

Some features of parallel computer operating systems may not be well-suited to support high data-rate applications such as multimedia information systems. One such feature is the large amount of data copying that takes place to perform data transfers. Thus it is not unusual for a single transfer from an I/O device to an application process to involve a copy from the device to the device I/O buffer, another copy from the device buffer to a kernel buffer, and a third copy to a user process buffer; the sequence of copies may be repeated in reverse for a process-to-device copy. While the nominal bandwidths for most workstation buses are 100 MBps or more, measured bandwidths for copying un-cached data are almost an order of magnitude less. Repeated data copying further reduces the effective bus throughputs and severely impacts the response times for applications such as digital video and audio. Bypassing some of this copying can produce significant performance improvements.

Separating the data from the control information about the data (or "meta-data") can help bypass some copying and reduce CPU involvement by allowing some transfers to be done by DMA. It can also help to enable effective data-transfer scheduling. There are several operating system-level, parallel-I/O scheduling algorithms, that are intended for use in an operating system that handles I/O requests for multiple applications, in systems where multiple I/O transfers can take place simultaneously. Thus they differ from traditional disk-scheduling algorithms (which schedule disk arm movements at the level of individual disks) and the application-specific I/O-scheduling algorithms (which schedule I/O operations of individual programs, e.g., for out-of-core sorting). Parallel-I/O scheduling is required because even if each individual disk schedules I/Os to minimize arm movement, and each individual application issues a minimal number of I/O requests, the simultaneous I/O requests of multiple applications for data residing on multiple disks can result in conflicts that, unless properly resolved via scheduling, can result in long delays and inefficiencies.

6 Architecture

Possibly the area that has received the most attention in terms of the I/O bottleneck has been the disk subsystem architecture. The use of *low-level parallelism* in the service of I/O in schemes such as disk interleaving, striping, RAID, RADD, etc., is well known and is briefly reviewed by Kotz [Kot96]. The gains

provided by the low-level schemes can be overwhelmed, however, unless scalable algorithms, smart compilers and appropriate operating systems mechanisms are used to increase the I/O parallelism at higher levels of the system; these higher-level techniques in turn lead to additional requirements upon the architecture. For instance, the use of I/O-optimal algorithms for sorting and matrix multiplication imposes architectural requirements, such as the ability to perform independent parallel disk accesses and to turn off parity, that are typically not supported by architectures employing such low-level schemes. In particular, if disk striping is used, where the read/write heads of all the disk drives move synchronously, the I/O complexity of the optimal algorithms increases by more than a constant factor.

While most efforts date have focused on parallel I/O for parallel applications running on parallel computers, there is increasing interest in the use of workstation clusters in place of parallel computers. Although the hardware is typically comprised of commodity components, so that the cluster hardware is roughly equivalent to that in a typical "distributed system" running on a LAN, the software is much different. For example, a parallel file system distributes the data of each file across many disks on many "nodes" (workstations), and provides an API sufficient for parallel applications (which execute simultaneously on many nodes) to express their accesses efficiently. Since the network is a cluster tends to be slower than the custom network found in a parallel computer, scheduling and other network optimizations are even more important in cluster implementations.

The emergence of multimedia applications such as digital video and audio also motivates further architectural requirements. The high data rates and timing constraints of digital video and audio necessitate better control over low-level timing of I/O transfers. In particular, system I/O channels should be interruptable and it should be possible to schedule the data transfers across the channels. It should also be possible to perform DMA to and from all devices and memory, using all addresses (i.e., not just word-aligned or block-aligned addresses). Although some buses like the IBM Microchannel do have this capability, they are typically not used in this fashion. Device controllers should be capable of large-grained burst-mode transfers, and should have relatively large memory buffers to help smooth out the jitter between different media streams.

7 Summary

The I/O bottleneck continues to be a serious concern for scientific computing and database applications, and the emerging areas of multimedia and visualization bring new I/O challenges. The I/O behavior of all application types needs to be studied in more detail, and more attention paid to designing applications to directly deal with the I/O bottleneck.

There are sophisticated models for I/O complexity used to develop I/O-

efficient algorithms for important functions like sorting and FFT. Current I/O models may or may not be realistic or representative of the I/O behavior of important applications on real machines. More work needs to be done to validate the I/O complexity models to determine the machine classes or operational regimes where they are appropriate. In addition, it is desirable to have models that capture computation, communication, and I/O in an integrated fashion, yet remain tractable.

Several efforts extend existing languages to provide constructs that allow a compiler to optimize the I/O of individual programs. Typically these efforts have focused on languages that assume a data-parallel programming model, which has been particularly useful for scientific computations. It would be useful to extend language and compiler support for the I/O performed by non-regular computations, or non-scientific applications, and to provide this support in higher-level (e.g., graphical) programming languages and environments.

There has been a recent surge of activity in parallel file systems and parallel-I/O interfaces for operating systems. Once again, much of this work has focused on support for scientific applications. There has been a growing awareness of the operating-system overheads of moving data, especially in terms of repeated copying of data as it is moved among various system and user buffers. Another important issue is to separate data from the control information about the data ('metadata') to allow better control and coordination of I/O activities, e.g., via scheduling.

Architectural solutions to the I/O bottleneck have enjoyed tremendous acceptance, particularly in the use of RAID and parallelism within the I/O subsystem. Given the persistence of the I/O bottleneck despite these solutions, the scope of architecture investigations needs to be broadened. There has been some work on alternative I/O interconnection architectures as well as the possibilities offered by the growing capabilities of individual subsystem components like disk controllers.

There remains much exciting research in the field of I/O for parallel and distributed systems. A forward-looking survey of the problems and prospects for research in the area was completed in 1996 [GVW96], and may be a useful starting point for researchers entering the field.

In this short overview of I/O in parallel and distributed systems we have necessarily omitted mention of many interesting research projects and papers. For more information, including pointers to many research projects, software packages, events, and related papers from the scientific literature, visit the parallel-I/O web page `http://www.cs.dartmouth.edu/pario/`.

8 Terms

- DMA: ...

- RAID:
- I/O:
- HDTV:
- MPEG:

9 Cross-refs

Terms that may want to "See I/O in Parallel and Distributed Systems."

- RAID
- parallel I/O
- remote I/O
- distributed file system
- input/output

References

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[BC96] Rajesh Bordawekar and Alok Choudhary. Issues in compiling I/O intensive problems. In Jain et al. [JWB96], chapter 3, pages 69–96.

[dC94] Juan Miguel del Rosario and Alok Choudhary. High performance I/O for parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[DG92] David DeWitt and Jim Gray. Parallel database systems: The future of high-performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[Gib92] Garth A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. An ACM Distinguished Dissertation 1991. MIT Press, 1992.

[GVW96] Garth A. Gibson, Jeffrey Scott Vitter, and John Wilkes. Strategic directions in storage I/O issues in large-scale computing. *ACM Computing Surveys*, 28(4):779–793, December 1996.

[JWB96] Ravi Jain, John Werth, and James C. Browne, editors. *Input/Output in Parallel and Distributed Computer Systems*, volume 362 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, 1996.

[Kot96] David Kotz. Introduction to multiprocessor I/O architecture. In Jain et al. [JWB96], chapter 4, pages 97–123.

[NKP+96] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1996.

[SN96] Elizabeth Shriver and Mark Nodine. An introduction to parallel I/O models and algorithms. In Jain et al. [JWB96], chapter 2, pages 31–68.

[VS90] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Optimal disk I/O with parallel block transfer. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC '90)*, pages 159–169, May 1990.