High Performance I/O for Parallel Computers: Problems and Prospects

Juan Miguel de Rosario
Alok Choudhary

CRPC-TR93329
July 1993

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

High Performance I/O for Parallel Computers: Problems and Prospects*

(Submitted: IEEE Computer, July 1993)

Juan Miguel del Rosario[†]and Alok Choudhary[‡]

Northeast Parallel Architectures Center
111 College Place, RM 3-201
Syracuse University
Syracuse, NY 13244-4100

Abstract

Along with the increasing use of parallel computers has come an increased demand for I/O systems support. Data movement to temporary storage, archival storage, visualization systems, or across the network to other computing centers has become a necessity for high-performance computing. The research and development of I/O systems for this type of environment is at a very early stage of its evolution. Although research in I/O systems have been conducted in the past, only recently have there been efforts to form a comprehensive characterization the I/O problem encompassing various perspectives (e.g., I/O in parallel machines, distributed computing, mass storage, etc.).

The aim of this paper is to present an overview of the many issues related to high-performance I/O in parallel computing environments. We discuss I/O requirements for Grand Challenge applications, issues related to performance characterization, I/O architecture alternatives, operating and file systems, compiler and runtime support, checkpointing, network I/O and others. In most cases, we present the current state of understanding and research, discuss outstanding problems, and describe some alternative solutions.

*This work was sponsored in part by DARPA under contract # DABT63-91-C-0028 and NSF MIP-9110810. Alok Choudhary is also supported by the NSF Young Investigators Award. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred. We also acknowledge the support of Center for Research in Parallel Computing and Concurrent Supercomputing Consortium for providing access to the computing resources.

[†]CIS Dept., Syracuse University

[‡]ECE Dept., Syracuse University

1 Introduction

Over the past two decades, advances in semiconductor and integrated circuit technology have fueled the drive towards faster, ever more efficient computational machines. Today, the most powerful supercomputers are capable of performing computation at billions of floating point operations per second (Gigaflop rates). This represents a growth of two to three orders of magnitude over the past decade. Much of this computational capacity is being harnessed to undertake large-scale mathematical modeling and simulation of a variety of physical, chemical and biological phenomena in connection with a broad range of theoretical and practical endeavors. For example in science, unprecedented levels of clarity and detail are being attained in areas such as climate prediction and control, air and water pollution and quality management, lattice gauge theory, quantum chromodynamics, large-scale structure and galaxy formation, vision and cognition, etc. In engineering, computational techniques are being applied to the design and testing of anti-cancer agents, anti-AIDS drugs, aircraft wingfoils, modern combustion engines, oil reservoir simulation, etc. The increase in productivity that has been obtained in these areas by adopting such high performance computing techniques is fomenting a revolution towards even more powerful machines. It is expected that in the coming years computational limits for the largest supercomputers will break the Teraflops barrier.

1.1 I/O in Grand Challenge Applications

A large number of supercomputer applications, such as those previously mentioned, are members of a set of scientific and technical "Grand Challenges", an annual list initiated some ten years ago by a Nobel prize winning physicist, Kenneth Wilson [CC93]. Aside from being extremely complex and requiring significant amounts of processing time, these applications often deal with enormous quantities of data. Current near-term high performance applications involve from 1 GByte to 4 TBytes of data per run. Although the main memory regions of supercomputers are extremely large, applications that manipulate more data than can exist in memory, and therefore have very high I/O requirements, are appearing more frequently (see the sidebar on "I/O requirements in Grand Challenge applications" for more detail). For example, current archival sizes for a grand challenge group typically ranges from 500 MBytes to 500 GBytes of storage with a peak of 10 TBytes. It is anticipated that by the time Teraflops machines with their Terabytes of memory make their appearance, these I/O requirements will increase dramatically, in some cases over 100-fold (e.g., climate modeling) up to 10 PBytes per grand challenge group.

But space is not the only consideration. Supercomputers are commonly interfaced with a variety of peripheral devices - external disk storage systems, mass storage devices, visualization devices, video cameras, networks, other supercomputers, etc. - for pre or post-processing of data, or simply for additional working storage. In many cases, the speed with which data can be accessed by the supercomputer determines the rate at which it can complete the assigned job [1]. The need to access data via network connected remote devices introduces significant amounts of additional delay over access to the internal I/O subsystem. The rise in support for global computing paradigms amplifies the severity of this problem. Today, most high performance applications involve I/O rates of 1 MByte/s to 40 MBytes/s to secondary storage and 0.5 MBytes/s to 6 MBytes/s to archival store. Application developers indicate that in all probability about 1 GByte/s to secondary storage and 100 MBytes/s to archival store will be required in the near future [Wor93].

[1]Such jobs, for which I/O (not computation) is the bottleneck, are said to be "I/O bound".

BEGIN SIDEBAR: "I/O requirements in Grand Challenge applications"
Massively parallel computers, armed with mathematical models, are a key technological advancement which allow scientists to study, in intricate detail, complex physical, chemical and biological phenomena, as well as those of medicine and other sciences.They accelerate the investigative process way beyond the traditional trial and error approach. For instance, computational fluid dynamic simulations replace wind-tunnel testing and physical prototyping in the early stages of aircraft design; simulations of the growth of crystalline semiconductors are used by companies to determine how best to grow better and faster chips for future supercomputers. In addition, visualization techniques provide a more humanly palatable form of representation of the enormous amounts of experimental data that arise from simulation. However, along with more detailed or larger simulations comes an increase in the amount of data that must be dealt with, either as input to or as a product of the computation/simulation. This data must be collected, stored, viewed, or reprocessed. In order to gain an "orders of magnitude" understanding of the quantities involved, we provide the following examples.

Imaging of planetary data. The spacecraft Magellan has been collecting data from the surface of the planet Venus since Sept. 15, 1990. Using a radar to penetrate surrounding cloud cover and scan the surface for structural information, Magellan has transmitted to earth over 3 Terabytes of data. To produce a 3-dimensional rendering of the surface at 200 MBytes of data per frame would require over 13 GBytes/sec of I/O throughput at 50 frames per second. This far exceeds the I/O capacity of today's machines; it takes several days to render a portion of the Venus surface on a 512-node Intel Touchtone Delta [Gro92].

Climate Prediction. Research efforts in climate and global change, long-range weather prediction, land surface processes are crucial to our understanding of earth, ocean, and atmospheric systems. The most complex of these are the general circulation models (GCM) of atmosphere and ocean which must be capable of simulating geophysical fluid dynamics on appropriate scales. Current atmosphere/ocean models have the following requirements on an Intel Touchtone Delta. For a 100-year atmosphere run with 300KM^2 resolution and 0.2 simulated years/machine hour, the simulation takes 3 weeks run time and generates 1144 GBytes of data at 38 MBytes per simulation minute. For a 1000-year coupled atmosphere-ocean run with a 150KM^2 resolution, the atmospheric simulation takes about 30 weeks, while ocean simulation takes 27 weeks; the process produces 40 MBytes of data per simulation minute, or a total of 20 TBytes of data for the entire simulation [FHS91].

4-D Data Assimilation. By collecting new scientific data of the earth system, researchers hope to gain a better understanding of how meteorological and chemical processes interrelate. This knowledge will allow them to produce more accurate predictions of global environmental change. The NASA Data Assimilation Office (DAO) has adopted a technique called 4-D data assimilation as a method for incorporating, in the space-time domain, newly collected data with existing models. Currently, the algorithm for doing this operates from a 3 TByte database with single runs producing from 100 MBytes to GBytes of output. These quantities will increase by orders of magnitude when NASA's Earth Observing System - a project which employs satellites to transmit environmental observations down to earth - comes online. By the end of the decade, it is expected that up to 1 TByte per day of observational data will be collected for analysis.

Table 1 gives a summary of the I/O requirements for many Grand Challenge applications. This data is based upon information derived from presentations given by Grand Challenge scientists at the Pittsburgh Workshop on Grand Challenge Applications and Software Technology [Wor93]. By and large, it reflects I/O requirements for current computational scales; these figures are expected to increase significantly in the very near future.

References

[Gro92] Morris Grossman. "Modeling Reality". *IEEE Spectrum*, pages 56-60, Sept 1992.

[FHS91] Ian Foster, Mark Henderson, and Rick Stevens. "Workshop Introduction". In *Proceedings of the Workshop on Data Systems for Parallel Climate Models at the Argonne National Laboratory*, July 1991.

[Wor93] Workshop on Grand Challenge Applications and Software Technology. *Survey of Principal Investigators of Grand Challenge Applications*, Pittsburgh, May 1993.

END SIDEBAR

Table 1: I/O requirements for Grand Challenge applications

A. Environmental and Earth Sciences:	
Application	I/O requirements
Environmental modeling	T: 10s of GBytes.
	S: 100s of MBytes - 1 GBytes per PE.
	A: Order of 1 TBytes.
Eulerian air-quality modeling	S: Current 1 GBytes/model, 100 GBytes/application; projected 1 TBytes/application.
	A: 10 TBytes at 100 model runs/application.
Earth system model	S: 108 MBytes/simulated day. 100 GBytes/decade-long simulation.
	B: 100 MBytes/s.
4-D data assimilation	S: 100 MBytes - 1 GBytes/run.
	A: 3 TBytes database. Expected to increase by orders of magnitude with the Earth Observing System (EOS) - 1 TBytes/day.
Ocean-Atmosphere Climate modeling	S: 100 GBytes/run (current).
	B: 100 MBytes/sec.
	A: 100s of TBytes.
Reservoir Modeling for porous media	B: Access to HiPPI speeds (50-100 MBytes/sec)
	A: 1 TBytes/year storage and retrieval
	M: Scientific database mgmt.
B. Computational Physics	
Solar Activity and Heliospheric Dynamics	B: 200 MBytes/sec
	A: Up to 500 GBytes
Convective turbulence in Astrophysics	S: 5-10 GBytes/run
	B: 10-100 MBytes/s
Particle algorithms in Cosmology and Astrophyics	S: 1-10 GBytes/file; 10-100 files/run.
	B: 20-200 MBytes/s
Radio synthesis imaging	S: 1-10 GBytes
	B: HiPPI bandwidths minimum.
	A: 1 TBytes
C. Computational Biology	
Molecular Computation	B: 30 MBytes/s
	S: 1-10 GBytes during execution
HP Computational Chemistry	B: 10-30 MBytes/s during execution 10-100 MBytes/s post-processing
	A: 100 GBytes
Computational structural Biology	B: 10-20 MBytes/s during computation
	S: Up to 500 MWords (2 GBytes) for a typical ensemble run
	A: 500 MWords
Computational quantum materials	S: 150 MBytes (time dependent code) 3 GBytes (Lanczos code).
	B: 40-100 MBytes/s
D. Computational Fluid and Plasma Dynamics	
Computational methods for coupled fields	B: 200 MBytes/s bandwidth
	A: 10 GBytes to 1 TBytes in 3 years.
Simulation of high-performance aircraft	T: 4 GBytes of data/4 hrs.
	B: 40 MBytes to 2 GBytes/s disk, 50-100 MBbytes/s disk to 3′ storage (comparable to HiPPI/Ultra)
Simulation of propulsion system	S: 50 MBytes
	B: 1 GBytes/s
	A: 50 MBytes to 1 GBytes
	M: High-speed object oriented database
Computational fluid and combustion dynamics	A: 1 TBytes
	B: 0.5 GBytes/s to disk, 45 MBytes/s to disk for visualization.

A: Archival Storage
T: Temporary Working Storage
S: Secondary Storage
B: I/O Bandwidth

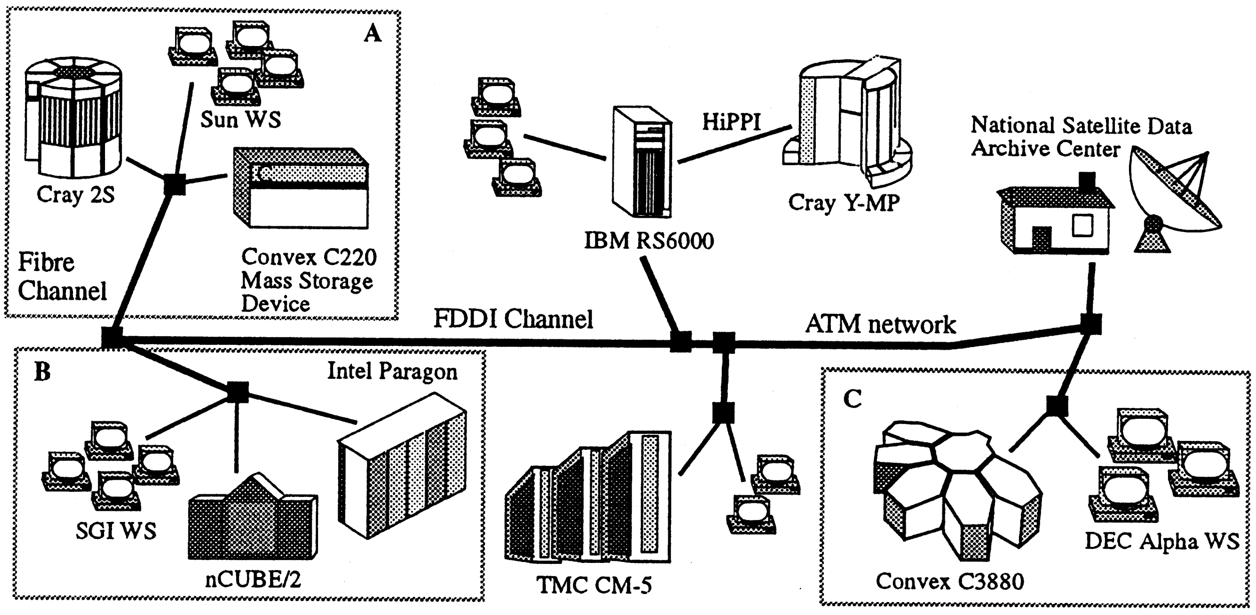

Figure 1: High Performance Distributed Computing Network

1.2 High Performance Distributed Computing

University, industrial and government research labs across the nation have long been interconnected either locally, or via wide area networks such as those that comprise the Internet. A vision for the future of high performance computing that is shared by many scientists today is that of a nation-wide heterogeneous distributed computing environment where information and data can be shared, processed, and stored in a seemless, globally oriented manner. The term *metacomputing*, originally used in the 1980's, refers to the basic concept of having several machines work cooperatively on a single problem. The recent popularity of this computing paradigm stems from the fact that the rate at which a supercomputer can execute a given application is a function of how closely the problem domain maps to the computer's architecture. Grand Challenge problems often consist of several tasks. Metacomputing will allow each task to be assigned to the machine that can execute it optimally.

In figure 1, we illustrate what a high performance computing infrastructure might look like. It consists of computational centers (academic, industrial or government research labs) composed of various combinations of vectorcomputers, massively parallel computers, multiprocessors, high resolution visualization systems, tens to hundreds of workstations, mass storage and archival systems, etc., connected together by network links of varying distances and capacities. As an example of a metacomputing application in this environment consider sites A, B, and C (enclosed within boxes in figure 1). Suppose that the application is a large-scale climate modeling program. Site A, a Cray-2S multiprocessor connected to a mass storage device via an HSX channel, may be performing data assimilation of archival data into a long-term climate prediction model. Simultaneously, it may be forwarding the data through Fibre channel connection to a massively parallel computer (iPSC, Paragon, nCUBE, or CM-5) some distance away at site B. The MPP may be using the data to execute a high resolution weather prediction model using a large internal disk array for additional temporary storage. Results from the MPP may be routed through an FDDI link to a network of workstations used for analysis by collaborating scientists. Or, as in site C, data may also be transported, via ATM for example, to a distant high performance visualization computer producing images resulting from 3-D ray tracing calculations.

The distributed nature of this computational paradigm places a high premium on the I/O capacities within and between processing centers. Such a distributed computing environment would only be feasible where the performance of the communications component of the infrastructure is in balance with its computational counterpart so that data can be transported as quickly as it is produced. This applies equally to the internal I/O subsystem of the supercomputer, and to external longer distance connections via network. Note that from the HPDC perspective, network communication can be viewed as an I/O problem.

1.3 Outline of the rest of this paper

In this paper, we discuss the nature of the I/O problem, important design issues and tradeoffs, and some recently proposed solutions. We restrict ourselves here to a detailed treatment of the I/O problem in relation to massively parallel machines; this includes their internal I/O subsystem as well as external interfaces. In addition, we briefly discuss some important issues in other areas such as networking, persistent store, etc. We begin in section 2 with a description of the nature of I/O in parallel machines. The succeeding section, section 3, deals with operating system and file system issues. Then, section 4 discusses runtime systems and compilers, followed by persistent object stores in section 5. In section 6, network technology is presented. Finally, section 7 presents a summary and a list of areas we did not cover in this paper.

2 The Nature of I/O in MPPs

In this section, we will discuss the issues related to the nature of the I/O problem in massively parallel machines (parallel I/O), and the architectural basis for these.

2.1 Characterization of the I/O Requirement

The parallel I/O problem can be viewed from a number of perspectives: languages, compilers, file and runtime systems, networking systems, operating systems services, storage systems, etc. This runs the gamut from hardware design, through system software, to user application.

Historically, three general types of I/O can be identified:

1. I/O to and from secondary storage (e.g., temporary storage, checkpointing, permanent storage, etc.)

2. I/O to a network interface (e.g., transfers over distributed file systems, or over high throughput connections such as HiPPI)

3. I/O to special devices such as video, graphic, or tape devices. Discussions on parallel I/O are often limited to secondary storage I/O (i.e., case 1).

At a lower level, parallel file organizations have been classified by Crockett [Cro89] into a number of categories on the basis of a global and internal view of the access pattern. For example, sequential files are accessed in a globally sequential manner which may include sequential, partitioned, interleaved, or self-scheduled sequential files. Global and partitioned direct access files are also defined. Existing parallel file systems such as the Intel's CFS and nCUBE's file system provide support for some subset of these file organizations.

As the use of parallel computers becomes more sophisticated, there is an important need to re-examine our understanding of the nature of the I/O requirement. In particular, the following concerns arise. Our current understanding of I/O requirements for scientific purposes stem primarily from past experience with supercomputing applications, or very basic (in terms of I/O) parallel applications. The nature of parallel I/O on distributed memory systems will vary greatly from those of supercomputers due to the difference in underlying hardware. The basic model for current parallel I/O systems includes an I/O subsystem architecture that is distributed in nature, consisting of independent I/O nodes each of which supports one or more secondary storage devices, while traditional supercomputers commonly have a large disk or disk arrays connected to the computational unit via a single bus line. I/O architectures are discussed further in the next section.

Another concern is that simply evaluating I/O bound parallel applications may not necessarily provide a good representation of the I/O requirement since the likelihood is great that the code had been designed to optimize its performance on existing inadequate I/O systems. Further, because of the rudimentary nature of current system software support, the application interface is very tightly coupled to the particular machine architecture. As the architecture of I/O subsystem architectures change, what we learn from these applications may no longer be valid. Finally, future fine-grained machines will change the I/O requirements, possibly making requirements more akin to those of transaction processing. Thus, we need to gain a more refined understanding of the I/O requirement for parallel computing. In designing approaches to explore this area, we need to be aware of parameter dependencies and possible design bias in any benchmark application used.

A key issue therefore is I/O requirements characterization. Here, understanding application I/O behavior is paramount. Application I/O patterns determine: the general I/O performance requirements, the necessary secondary and tertiary data storage capacities and bandwidths, and the acceptable system configurations and software features. Thus, before the I/O problem can be properly addressed, quantitative assessments are needed for the following:

1. spatial and temporal data access patterns
2. current hardware and system software response to application I/O demands
3. current performance bottlenecks

A takeoff point for this research is the instrumentation of a number of platforms (vector and parallel) for the purpose of collecting I/O trace data from large application codes. With the results obtained from observing I/O patterns of several large applications, scientists hope to create templates, evaluation tools, benchmarks, and abstract models of I/O behavior with which the following questions can be examined: Are there commonly observed temporal patterns of I/O requests; If so, what are they? What patterns are associated with each given data distribution? How do we expect I/O behavior to scale with increasing problem size? What is the behavior of I/O in physically remote, heterogeneous environments?

This acts as an entry point into a development cycle in which I/O characterization is followed by re-evaluation and modification of existing designs, and back. It is hoped that this cycle will create an evolutionary pathway for the design of scalable I/O systems.

2.2 Architectural Model for Distributed Memory I/O Subsystems

The single overriding design objective in distributed memory design systems is scalability. This property extends to the I/O subsystem and its communication channels. In considering the architectural model for distributed memory I/O subsystems, we assume a fixed number of processors

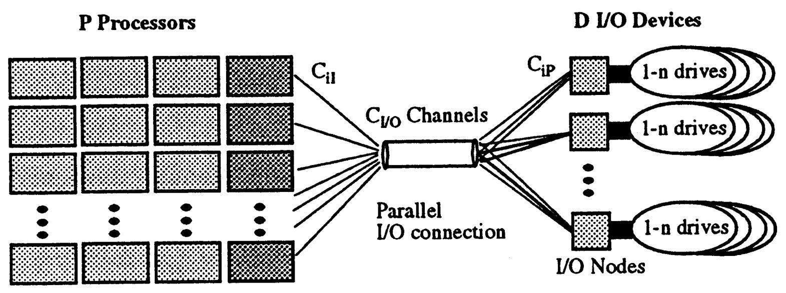

Figure 2: Parallel I/O subsystem architecture

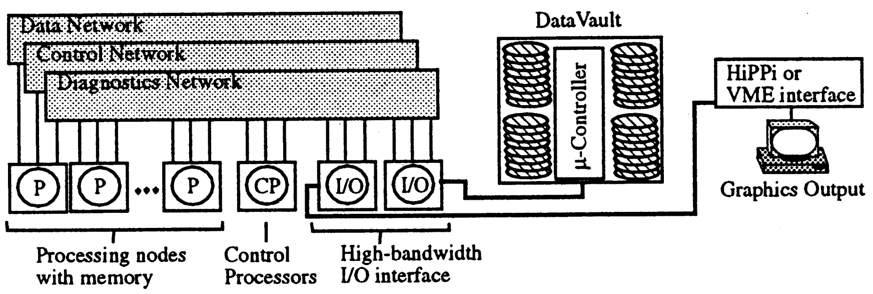

Figure 3: Thinking Machines CM-5 architecture

within the computational array, and a fixed number of disks. Each disk is associated with an I/O node through a connected controller. Each I/O node acts as part of a distributed file server and operates as an intermediary between a set of disks and the computational array. Figure 2 illustrates the system configuration where the darker processors within the computational array represent processors with a direct connection to an I/O device. A specific example of this configuration can be found in the Intel Touchtone Delta architecture which is a 16 x 32 two-dimensional array of compute nodes with 16 I/O nodes on each side of the array; each I/O node has two disks associated with it.

Although following the same basic model, the Thinking Machines CM-5 has a slightly different I/O device architecture. The CM-5, shown in figure 3, generally has fewer numbers of I/O processors with a high bandwidth channel connected to either a standard interface or to the CM-5 secondary storage device. The secondary storage device, called the DataVault, is a RAID level 3 device [Thi91].

The Intel Paragon, on the other hand, has a mesh topology where there are no special I/O channels. Separate I/O nodes are used but can be placed anywhere within the mesh network. Each I/O node has its own set of disks attached to it as in our model; the set is maintained as a RAID level 3 device [Int91].

An I/O node, with its accompanying set of disks, can be viewed as a separate functional unit within the parallel I/O system. The organization between the disks and its I/O node can take on any of the large number of existing host-to-disk architectures. These range from the more conventional host/disk head-of-string type of format to any of the RAID level architectures [PGK88]. An example of the I/O sub-unit comprised of an I/O node and its associated disks is illustrated in figure 4.

The performance of each I/O device is measured by its ability to handle the workload from the computational array. Included among the factors that constrain the I/O device's performance are

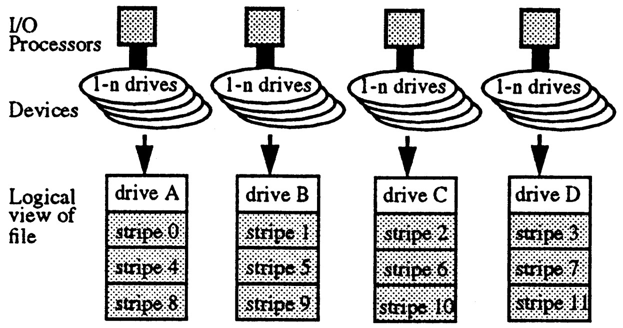

Figure 4: Parallel I/O device architecture

the canonical set of costs associated with disk devices. Seek time, rotational latency, and transfer time all contribute towards the overall disk service time at the I/O node, and thus to the service time of the entire I/O subsystem. As a first approximation, the workload applied on the set of disks in the parallel computer may be determined by the type of data distribution selected at the application level. For instance, cyclic distributions tend to require smaller transfers from disk, while block distributions tend to require the movement of large blocks of contiguous data. This workload is transmitted to the disks in the form of logical requests from the computational array to the I/O nodes. However, caching and/or prefetching within the computational array and/or on the I/O nodes contribute towards redefining the workload that is actually applied (i.e., visible) to the set of disks. For example, an I/O node may group together many small requests into a single large data request to its set of disks, resulting in better performance.

Parallel files are distributed among the set of disks in the I/O subsystem by declustering the data across the disk array (a technique known as striping), as shown in figure 4. Load balance issues arise from the degree of correspondence between the application defined data decomposition and the data storage mapping defined by the stripe size. A discussion of the relation between stripe size and load balance is provided in section 3.

Another major concern in parallel I/O architectures involves the data transfer bandwidth to and from the I/O devices. This bandwidth is limited by the size and number of communication channels between the computational array and the I/O devices. This number varies with the ratio of processors to I/O devices, and the underlying interconnection topology. In general, we can view the interconnection model as illustrated in figure 2. We assume that the computational array has P processors and some interconnection topology such as a grid or hypercube. Also, there are a fixed number, D, of I/O devices (here composed of one I/O node and one disk each). The number of I/O channels, $C_{I/O}$, between the computational array and the I/O devices is given by

$$C_{I/O} = \sum_{i=1}^{D} D_i \times C_{iP} = \sum_{i=1}^{P} P_i \times C_{iI}$$

where $D_i \in$ D the set of I/O devices, and C_{iP} represents the number of connections from each D_i to the computational array; further, $P_i \in$ P the set of computational processors, and C_{iI} represents the number of connections (1 or 0) from a P_i to an I/O device. The greater is $C_{I/O}$, the greater the data transfer capacity of the I/O subsystem.

Another approach being considered to alleviate the limited data transfer capacity of the I/O subsystem is the use of tightly coupled storage devices. This approach anticipates a continuation of the trend towards miniaturization of disk devices from 3.5 inches to 1 inch in the next five years

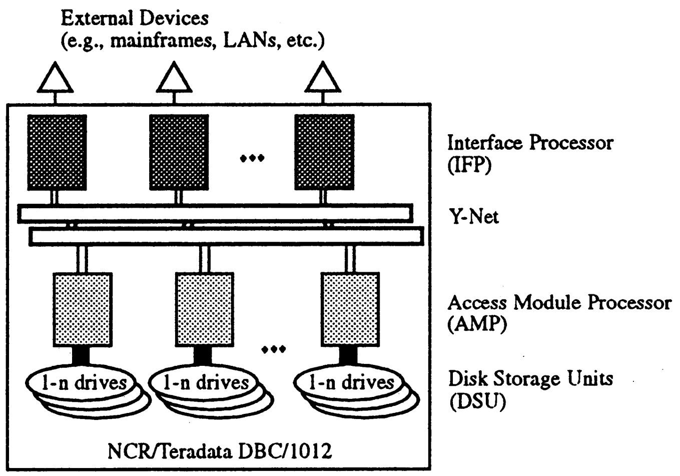

Figure 5: Architecture of the NCR/TERADATA

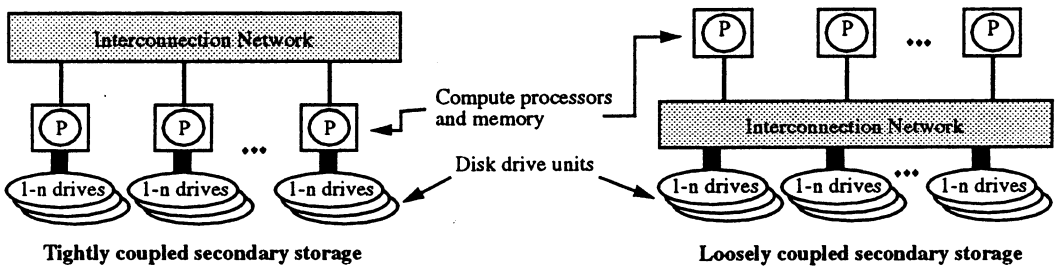

Figure 6: Tightly vs. Loosely coupled compute and I/O systems

and is based on the scheme used by parallel database systems such as Teradata [Pag92]; this is illustrated in figure 5. In an extension of this model to general purpose parallel machines, the disk units would be an integral part of the computational array. As seen in figure 6, each disk unit is closely coupled to a processing unit resulting in very low transfer times between the processor and its local disks. With this scheme, functionality that is currently available in existing parallel file systems for distributing data blocks across a variable number of disks will remain unchanged. However, since the disks are part of the computational array, the degree of connectivity of each disk with the set of processors will increase giving each a higher communication bandwidth. The problem of embedding I/O nodes in parallel computers is discussed by Reddy and Banerjee [RBA88]. In their study, they propose an algorithm for embedding the I/O processors within hypercube and binary n-cube multiprocessor topologies. They present theoretical results to show that such a method results in fewer overall links, higher I/O adjacency, and a higher degree of tolerance to I/O failures (for the same number of I/O nodes) than existing designs. Some practical implications of their method is also discussed.

Before such models can be completely adopted, however, the following questions remain to be addressed: What are the effects on compute performance of extra I/O processing required of the processors within the computational array? What limits are imposed by additional memory requirements for I/O buffering, etc.? What are the effects of additional contention in the interconnection network arising from increased I/O traffic within the computational array? How will latency reduction issues (arising from reduced possibilities for overlapping I/O with computation) be addressed? How are external devices (e.g., tape silos, networks, etc.) to be connected?

The use of tightly coupled storage systems in persistent object storage implementations will be discussed in section 5.

3 Operating System & File System

Providing the necessary support for parallel I/O at the lower levels of system software requires an investigation of existing algorithms for file management in distributed memory parallel environments. Layered upon the previously discussed architectural model, requests for service are initiated by sending messages to the I/O nodes. Concurrency is achieved due to the declustering (striping) of data across the I/O devices. User level libraries within the computational nodes provide the necessary file data management support. Benchmark studies conducted on existing file systems allow us to identify deficiencies that must be addressed in our attempts to construct better parallel file systems. We consider a number of them below.

Communication Latency. Communication latency to the I/O nodes contributes greatly to the poor performance of existing systems. High latency dominates the overall transfer time of a sequence of requests moderate size. This means that requests must be made for large quantities of data if they are to be efficiently serviced; this is incurred at the expense of using an access mapping with a more natural correspondence to the computational decomposition. In object oriented systems such as the Intel Paragon OSF/1 operating system, another concern is that the object structures impose additional processing overhead thereby increasing communication latency.

One scheme that has been proposed to alleviate latency problems within the computational array takes a diametrically opposite view. In this system, called Active Messages [vECSK92], requests are broken down into many smaller ones. The communication protocols are simplified and the software latency (no. of instructions) in transferring a request from the application layer to the interconnection network is greatly reduced. Additional benefits arise from the overlap between computation and I/O made possible by the asynchronous nature of the message passing protocol.

Such approaches have yet to be investigated in relation to I/O systems.

Data Decomposition. In constructing parallel file systems, we are concerned with providing support for user applications. A common programming paradigm in scientific computing involves a decomposition of the problem domain. This decomposition gets translated into a mapping of the data domain over the computational array (see the sidebar on "Data Mapping in Parallel I/O). In conducting I/O, the application must be able to preserve some correspondence between this mapping and the mapping of data over the storage devices (e.g., disks). This must be accomplished by the file system in an efficient manner.

Current parallel file systems have little support for data decomposition or control over stripe size. Recent tests have been conducted in which I/O access patterns for several data decompositions have been applied to existing parallel file systems. Benchmark results show that these file systems are extremely sensitive to I/O access patterns, and that performance varies greatly as a function of the data decomposition [BCdR93]. Also, since data is declustered across I/O devices, and by inference over the I/O nodes, the load balance aspect of data access becomes critical. If the data decomposition selected by the application is incompatible with the stripe size, it is possible to overload a particular I/O node with requests, creating a severe bottleneck in the file system. For example, suppose an application has been programmed to distribute data in a Row-Block fashion where a 2K byte block row is assigned to each of 4 processors. Suppose the data is then striped across some number of disks with a Row-Block distribution, and the stripe size is set such that each disk will have an 8K byte block. Then, at the first access phase, all four processors will simultaneously access data from the first disk. On the second phase, each processor will access its second block from the second disk, and so on for succeeding access phases.

To further illustrate load balance effects, the graph in figure 7 shows, for a variety of data decompositions, the time it takes to complete a read operation as a function of stripe size. In this figure, stripe size, which ranged from 64 bytes to 1 MByte in the actual experimental results, is shown as a normalized value obtained by dividing each stripe size by 1 MB. The read times associated with each curve are also normalized by dividing over the largest read time taken for that curve. The data was collected from an nCUBE/2 [dRBC93].

Hence, it is necessary for the file system to be capable, not just of accommodating a variety of data distributions, but of efficiently managing various permutations of data decomposition versus decluster mappings. An approach for dealing with this problem at the runtime system level has been proposed. This will be discussed in the section on runtime systems below.

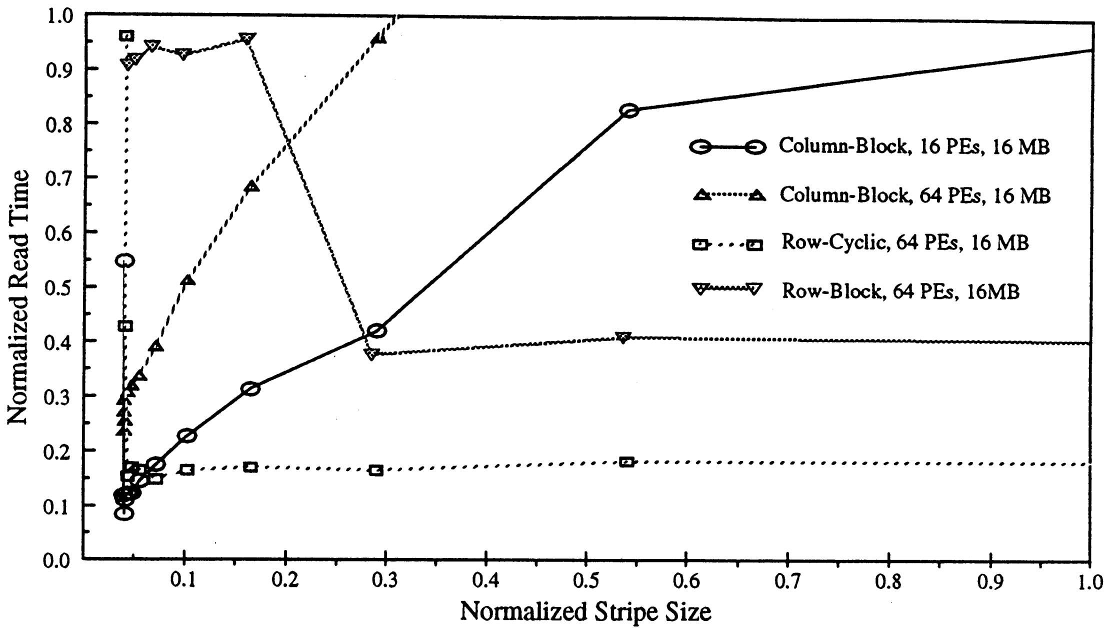

Figure 7: Read time as a function of stripe size

BEGIN SIDEBAR: "Data Mapping in Parallel I/O"

In programming a parallel computer, data decomposition is often used as a method of obtaining some degree parallelism that is usually easy to manage and typically closely matches the problem domain. The decomposed data structure is then distributed across the computational nodes of the parallel computer and processed concurrently.

When I/O has to be performed, each compute node has to possess some knowledge of the location of the data that belongs to its portion of the distributed data structure. In other words, a mapping function has to be established from the data structure element to the relevant disk block. In order to establish such a mapping from the processor array to the distributed file, we note that two sub-mappings need to be considered [DM91]. The first mapping, M1, involves the (more familiar) mapping of data over the set of processing elements. The organization of the file data over the set of disks represents the second mapping, M2. For any system, this mapping is determined by the system configuration setup. For parallel I/O to take place efficiently, both these mappings must be resolved into a data transfer strategy as in figure 8. The current parallel file systems on the nCUBE-2 resolves these mappings internally into a single data transfer mapping which is used to compute proper source and destination addresses during file data access. The Intel Touchtone Delta file system (i.e., CFS) maintains only the M2 mapping, giving the user the responsibility for managing the M1 map. This type of access is called *direct access*. Problems arise from this approach in cases where the first and second mappings resolve into a data transfer mapping (representing an access strategy) that result in poor I/O performance. It turns out that such problematic mapping pairs are quite common [dRBC93].

As a simple example, consider a ozone-depletion program which reads satellite information from a file that is distributed across a set of disks. Assume that the satellite information is stored as position associated data on a rectangular grid region covering the geographical area of interest. Further, assume that the data is distributed across the disk array according to grid column. Thus the M1 mapping is a column mapping from logical data grid positions to the physical disks. Suppose that the program requires that the data be read according to rows of the grid region. This

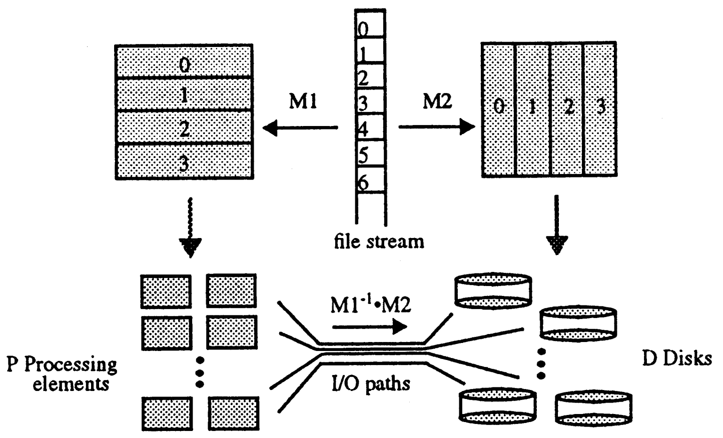

Figure 8: Parallel I/O mapping functions

represents the M2 mapping, which is a row mapping from the data to the computational array. On existing I/O systems, these mappings will be resolved into an access strategy which requires that each computational element read numerous small sections of data from the disks in order to construct its allocated row in local memory. Thus, the direct access strategy can result in enormous costs.

References

[DM91] Erik P. DeBenedictis and Peter Madams. "nCUBE's parallel I/O with Unix capability". In *The Sixth Annual Distributed-Memory Computer Conference*, pages 270-277, 1991.

[dRBC93] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. "Improved parallel I/O via a two-phase run-time access strategy", In *The 1993 IPPS Workshop on Input/Output in Parallel Computer Systems*, pages 56-70, 1993.

END SIDEBAR

Interface. Assessing a file system's ability to support data distribution must be made on the basis of some consensus. This consensus must establish an importance hierarchy for classes of data decomposition patterns on the basis of parameters such as extent of use. In conducting these evaluations, attention is naturally drawn to an important aspect of parallel file system design - the interface. Until now, no standard interface has been agreed upon for parallel file systems. Although this is a critical requirement from a software engineering standpoint, vendors are reluctant to accept standards for fear of losing the ability to provide user access to special architecture dependent features of their product that may increase its marketability. However, a standard interface clearly needs to be agreed upon since this would serve as a foundation for the evolutionary development of the I/O system.

The key questions in interface design have to do with the amount of information and control made available to the user. Traditional interfaces have only provided simple access mechanisms. Parallel programming paradigms require that some mechanism be provided which will allow annotations regarding access patterns to be communicated to the file system. Further, it might be possible that explicit control of lower level configuration parameters (e.g., block placement, striping, prefetching, etc.) need to also be made available.

These capabilities provide the user with the ability to explicitly adjust for good performance. Further, the system's ability to utilize additional access information will allow it to perform optimizations instead of relying on general purpose algorithms.

Table 2: Capabilities of existing commercial parallel file systems

File System	S	Q	I	C	P	D
Intel CFS	limited	limited	yes	no	n/a	no
Paragon PFS	yes	yes	limited	yes	limited	no
nCUBE (old)	yes	limited	yes	no	n/a	no
nCUBE (new)	yes	limited	yes	no	n/a	limited
KSR-1	no	?	limited	no	limited	n/a
MasPar	no	yes	no	no	no	n/a
TMC DataVault	no	yes	no	no	no	n/a
TMC SDA	no	yes	no	no	no	n/a
IBM Vesta	yes	yes	yes	no	n/a	yes

S	:	control over data declustering or stripe factor
Q	:	ability to query the current configuration
I	:	ability to access disk blocks independently
C	:	ability to turn caching on or off
P	:	ability to turn parity on or off
D	:	ability to recognize and support (optimize for) data distribution

BEGIN SIDEBAR: File System Functionality and Interface Requirements

File systems for massively parallel machines come in many flavors, varying both in functionality as well as interface. As parallel programming matures, it becomes increasingly important for the community to develop and adopt a set of standards which will allow for greater application portability.

Cormen and Kotz have recently surveyed existing commercial parallel file systems, evaluating on the basis of a proposed set of required capabilities [CK93]. Their results are illustrated in table 2 which has been extended here to include support for application level specification of data decomposition, D, as an additional criterion (considered n/a for SIMD or shared memory machines).

Their formulation for the set of necessary capabilities is founded upon what might be required to perform a collection of commonly used algorithms such as sorting, permutations, matrix transpose, FFT, matrix multiplication, and LU decomposition; further support is taken from results of previous empirical studies. The table shows that existing file systems have still quite limited functionality.

Other researchers propose that extensions to the Unix file system will be adequate to support the wide variety of parallel and distrbuted computing requirements [DJ93]. This approach has the advantage of emphasizing an evolutionary developmental path. Figure 9 illustrates a sample code fragment for the current nCUBE file system which provides some support for data distribution mappings by adopting the Unix extension approach.

References

[CK93] Thomas H. Cormen and David Kotz. "Integrating Theory and Practice in Parallel File Systems", In *The Proceedings of the 1993 DAGS/PC Symposium*, pages 64-74, Hanover, NH, June 1993.

```
/* EXAMPLE :  Sample C program using BLOCK mapping */
main()
{
    int array[1024];                  /* each node's data is a 1024 element array of ints */
    ullong parms[2];                  /* data structure to contain mapping parameters */
    struct mapinfoioctl map;          /* data structure for mapping function information */
    int fd;                           /* striped file descriptor */

    fd = open ("/dir1/tmp/foo", O_RDWRIO_CREAT, 0666); /* open striped file */

    map.fn_type = BLOCK;  /* select a block mapping */
    map.parmcnt = 3;      /* number of parameters for this mapping */
                          /* allocate 32 rows of 128 bytes (32 ints) each, per processor */
    parms[0] = 128;       /*     width of one block (row bytes / blocks per row) */
    parms[1] = 32;        /*     height of block (32 rows) */
    parms[2] = 2;         /*     blocks per row */
    map.parm = parms;     /* store map parameters */

    ioctl (fd, IOCTL_SETSTRIPEMAP, &map);  /* inform I/O subsystem of selected mapping */
    status = write (fd, array, 4096);      /* write local data to striped file */
    close(fd);                             /* close the striped file */
}
```

Figure 9: Sample code for data mapping interface on the nCUBE

[DJ93] Erik P. DeBenedictis and Stephen C. Johnson. "Extending Unix for Scalable Computing", *IEEE Computer*, Nov 1993.

END SIDEBAR

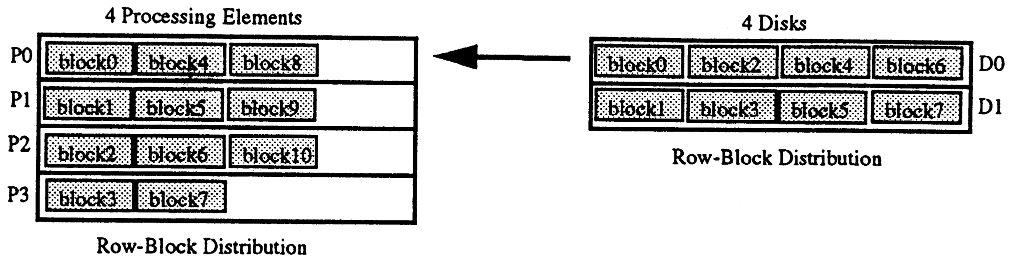

Figure 10: Block distribution for striped file

Prefetching and Caching. Prefetching and caching of data within the I/O subsystem involves an extension of the solution applied to primary memory. The idea is to exploit locality of access by temporarily saving (caching) blocks which contain recently used data in the expectation that nearby data will also soon be accessed. Prefetching is a predictive extension to caching based upon longer distance distributions of locality. That is, blocks next to those which contain requested data are retrieved and cached. The critical issue here is the nature of locality. In the traditional view, locality is established by a sequential view of data access. However, in a parallel subsystem this view may no longer hold. From the point of view of an I/O server node, requests may arrive such that prefetching every other block instead of every block results in improved performance. For instance, consider an application which has selected an 8K row-block distribution over 4 processor nodes, and suppose that the file is striped in a row-block order of 4K stripe blocks across 2 disks. Assume further that each disk will prefetch an extra block (i.e., it reads 8K bytes - 2 blocks - at a time). With this configuration, processor P0's first 4K (file block 0) is on disk D0, its second 4K (file block 4) is on disk D1, its third 4K block (file block 8) is on disk D0, etc. A partial view of this configuration is illustrated in figure 10.

When the parallel file is opened, disk D0 will prefetch its first two blocks (file blocks 0 and 2). On the first read request by processor P0, disk D0 will send its first block (file block 0) and prefetch another block (file block 4), keeping file block 2 in its cache. The next read request that disk D0 will get should come from processor P2, which should be a request for the previously prefetched file block 2. It seems from this that traditional locality is preserved. However, suppose that the disk system could exploit knowledge of the data decomposition on the computational array. In this case, the disk system could prefetch every other block to improve performance. The read sequence would then be as follows. When the parallel file is opened, disk D0 will prefetch its first and third block (file blocks 0 and 4). Then on the first read request by processor P0, disk D0 will send both blocks and prefetch the next two blocks (file blocks 2 and 6). Thus, requests can now be satisfied two blocks at a time instead of one. The number of messages required to read the entire file is reduced by half, and the cost of latency of each request is also reduced by half.

Concerns exist that the cost of maintaining the coherence and consistency conditions necessary to sustain prefetching and caching approaches will outweigh its benefits. However, further research in this area is warranted on the basis of the success this approach has had in other areas of hierarchical storage.

Additional considerations include the possibility of providing some measure of redundancy in order to increase the availability of data and as a method of improving load balance. Research in this area has been conducted for shared memory machines [Dav90], but none so far for distributed memory machines. In conjunction with this, the need for providing fault tolerance or recovery also requires addressing. In either case, algorithms to handle consistency and coherence problems need to be formulated.

Checkpointing. Because of the immense computing power made available by parallel machines, it is common for users to share portions of the machine for their production runs. A problem arises when the system must be reset due to a crash in one of the programs sharing the machine. In this case, programmers must provide regular state saving routines which will allow them to start from some point just prior to the interruption. Concurrent checkpointing will allow long running jobs to automatically save state at regular intervals so that they may be restarted after interruptions without unduly retarding their progress. The presence of a checkpointing facility will ease debugging problems which appear far into production code. It will provide for fault tolerance of hardware as well as software errors, network malfunction, and system interruptions. This capability is essential for grand challenge applications which typically require large amounts of resource and time.

In order to provide concurrent checkpointing, message synchronization techniques need to be developed that are capable of providing a consistent view of the global machine state. Parallel I/O support must be available that can be effectively used in the state saving process. This support is absolutely essential for developing real-time checkpoint and restart capability. To see this, consider the following example. Suppose we have a program that we wish to checkpoint. We assume that our only concern is to checkpoint data that is in memory (i.e., state of disk files assumed to be unnecessary). Further, we take as our goal a 100 sec. duration to complete the checkpoint process. In this situation, for each GByte that needs to be checkpointed, we need a $\frac{10^9 Bytes}{10^2} = 10^7 Bytes/sec = 10 MBytes/sec$ I/O bandwidth. For a Terabyte of data, this translates to a 10 GBytes/sec I/O bandwidth; such a bandwidth that would require 100 100 MByte/sec HiPPI channels to attain.

4 Runtime System and Compilers

Compiler and runtime system support for parallel I/O will maximize the system's ability to exploit user supplied or source level information to optimize I/O performance. The additional information will allow for the formulation of improved I/O access schedules, which will result in a more effective communication latency hiding strategy.

Compilers. Various compiler optimizations may be used to enhance I/O performance in parallel programs. Recognition and parallelization of I/O operations are key ingredients here. Compiler techniques need to be developed to allow I/O operations to be parallelized for a variety of file types and data formats. Mechanisms that permit user expression of I/O data structures would facilitate programming as well as functional interpretation of instructions by the compiler. Analysis methods similar to those for automatic decomposition should be investigated to enable the compiler to reschedule operations, overlapping I/O with computation. Further, compile time information on the access pattern used by the application may be supplied to the runtime system to assist in the generation of efficient access and checkpointing schedules. In short, it is the responsibility of the compiler to:

1. determine the memory and disk data distributions
2. generate I/O statements which are optimized
3. generate optimized communication instructions.

The types of optimizations the compiler must perform include:

1. overlap of I/O and computation by pre-fetching and caching

2. move I/O and communication out of loop structures whenever possible
3. match communication and I/O
4. block computation and I/O operations.

With such a system, a programmer could define an array as being of "out-of-core" type, for example, and select some data decomposition. The compiler would then recognize the array as an I/O structure and analyze the application's usage pattern (e.g., assignment statements) for elements of the structure. Then, with this information and on the basis of the data decomposition, the compiler could reorganize I/O operations to produce a efficient I/O schedule.

Runtime. Runtime libraries afford a level of insulation from operating system and file system software that make them attractive as a development environment. Providing parallel I/O support at this level allows for a greater chance of portability [Ken92]. Further, by incorporating a comprehensive interface to compilers, additional compile time information can be harnessed in the formulation of a data movement strategy for the application.

Although language extensions provide us with a means of representing data distribution information in a way that closely matches the underlying computation, provisions for language support which will allow similar specifications to be made with I/O expressions have not been sufficiently addressed. As a result, it is difficult and sometimes impossible to perform such a parallel I/O mapping in a manner that results in optimal performance. At the runtime level, information provided by the compiler or runtime routines can be employed to dynamically deal with array partitions and data mappings. Additionally, algorithms can be devised that use runtime information of access patterns and data distributions to prefetch and cache data from the I/O system more effectively. Techniques may also use system parameters such as the number of I/O and compute nodes, the stripe size, I/O and computational data distribution mappings, etc. to determine the most advantageous patterns of I/O request that each processor can make. Current work on runtime systems make use of composite mapping techniques to improve parallel I/O performance (see the sidebar on "Composite Mapping Strategies" for more details).

Table 3: Direct vs. Two-phase access (16 PEs, 5K*5K Array, time in msec)

Distribution	Best Read Time	Redist. Time	Total Read Time	Direct Read Time	Speedup
	5K*5K	5K*5K	5K*5K	5K*5K	5K*5K
Column Block	3357	-	**3357**	**3357**	**1**
Column Cyclic	3357	1805	**5162**	**9890**	**1.92**
Row Block	3357	673	**4030**	**69939**	**17.36**
Row Cyclic	3357	2603	**5960**	*	> 604.03

BEGIN SIDEBAR: "Composite Mapping Strategies"

Previous experimental results show that file system performance can vary greatly as a function of the selected data distribution. Further, performance degradation by a factor of 20 or more was commonly observed; the bandwidth for any given parallel I/O configuration is highly dependent upon the file size; stripe size dependent factors (e.g., load-balance, request size) cause widely divergent access times for most distributions; and lastly, that parallel I/O for certain common data decomposition patterns can not be supported by some existing parallel file systems [dRBC93].

Based upon these observations, alternative schemes for conducting parallel I/O have been devised. One particular approach, the two-phase access strategy, employs combinations of mappings improve average performance and guarantee greater consistency over a broader spectrum of data distributions. The basic idea behind the two-phase access strategy involves a division of the parallel I/O task into two separate phases. In the first phase, the parallel data access is performed using a data distribution which conforms with the distribution of data over the disks. That is, (from the sidebar on data mappings above) we introduce an intermediate mapping $M2'$, and access data with $M2' = M1$. Subsequently, in phase two, we redistribute the data at runtime to match the application's desired data distribution (i.e., from $M2'$ to $M2$). By employing the two-phase redistribution strategy, the costs inherent in many of the I/O configurations are avoided. Selecting a single, "good" configuration effectively reduces the bottleneck activity - I/O to the parallel device. Further, the redistribution phase improves performance because it can exploit the higher bandwidths made available by the higher degree of connectivity present within the interconnection network of the computational array. This strategy effectively decouples the user selected data distribution mapping from the file mapping to the disks (i.e., the declustering mapping); this results in performance that is much less dependent on user selected mappings. In tables 3 and 4 we present a comparison of access rates between the direct access and two-phase access strategies obtained from runs on an Intel Touchtone Delta [dRBC93]. The second column, in each table, labeled "Best Read Time", is the time it takes to read the desired data into the computational array using a distribution that conforms to the distribution of data on the disks. This is phase one. the "Redist. Time" (i.e., redistribution time), is the time it takes to redistribute the data into a distribution that conforms with the application selected decomposition; this is phase two. The "Total Read Time" is the sum of the first and second columns and represents the total time to read the desired data into the computational array via two-phase access. The fifth column, labeled "Direct Read Time", represents the time it takes to read the desired data according to the selected data distribution using direct access. The last column, "Speedup", shows the improvement gained from two-phase access over the traditional direct access method.

Another composite mapping strategy can be applied to "out-of-core" type applications. The idea is to extend the two-phase access with an additional mapping function which will define the relationship between portions of the out-of-core file. This is illustrated in figure 11. From the

Table 4: Direct access vs. Two-phase access (64 Processors 10K*10K Array, time in msec)

Distribution	Best Read Time	Redist. Time	Total Read Time	Direct Read Time	Speedup
	10K*10K	10K*10K	10K*10K	10K*10K	10K*10K
Column Block	11395	-	**11395**	**11395**	**1**
Column Cyclic	11395	2478	**13873**	**63400**	**4.57**
Row Block	11395	1028	**11623**	**78767**	**6.78**
Row Cyclic	11395	3092	**14487**	*	> 248.50

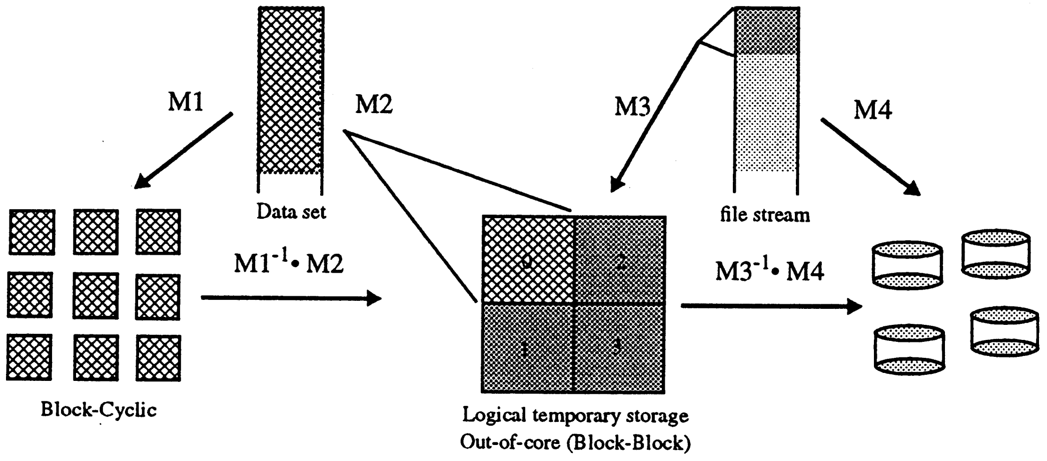

Figure 11: Composite mapping for parallel I/O

figure, we see that two mapping compositions are formed. The first composite mapping, $M3^{-1} \bullet M4$, defines how the file stream maps from the disk array to a logical temporary storage, in this case decomposed into a block-block distribution. The second composite mapping, $M1^{-1} \bullet M2$, defines which portion of the logical temporary storage is to be accessed by the computational array and processed in the current iteration of the out-of-core algorithm; in this example, the first block is accessed. With this structural mapping, the application is now able to access any portion of the parallel file during each iteration on the basis of the data decomposition it specifies for the logical temporary storage data structure. In addition, the composite mapping strategy enables this type of access in a transparent manner similar to that of the two-phase access strategy.

Figure 12 shows what type of statements may be required to perform an out-of-core computation given a runtime system that supports a composite mapping strategy for parallel I/O. The sample code is based upon HPF Fortran [Ken92] and a hypothetical runtime support system. Initially, distribution mappings are declared by using *temp1* and *temp2*. Then, *temp2* is declared as a special out_of_core type with Block_Block distribution; the entry for Logical_Map determines in what order the blocks of *temp2* will be accessed. *temp1* is then mapped to a real array A in which per-iteration data will be stored for computation. When a pread is performed, data will be read from the parallel file according to the composite distribution of *temp1* and *temp2*. That is, for the first read, for example, data from the parallel file which maps to the first block of *temp2* (Block 0 in figure 12) will be what is read into A. But, this data will be read into A according to the distribution defined by *temp1* (i.e., Row_Block). On subsequent preads, succeeding blocks of *temps2* will be read into A. Note that *temp2* is not a data-structure and has no memory requirement; it is a merely a logical entity representing a data decomposition mapping.

References

[dRBC93] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. "Improved parallel I/O via a two-phase run-time access strategy", In *The 1993 IPPS Workshop on Input/Output in Parallel Computer Systems*, pages 56-70, 1993.

[Ken92] Ken Kennedy (chair). "High Performance Fortran Language Specification, version 0.3", *CRPC technical report, High Performance Fortran Forum*, Rice University, 1992.

END SIDEBAR

```
C Sample code for out-of-core computation.
        PROGRAM Sample
        real, array(1024,1024) :: A
        decomposition temp1(1024,1024)
        decomposition temp2(4096,4096)
        distribute temp1(Row_Block)
        distribute temp2(Block_Block, Out_of_Core, Logical_Map)
        align A(i,j) with temp(i,j)


        call popen(3,'grid.dat', old)          ! open the file for access.
        do                                     ! iteration loop
            call pread(3, A, temp1, temp2, eof) ! read data from file into A.
            if (eof) go to 20
C start computation
        .
        .
        call pwrite(3, A, temp1, temp2)        ! write data back to file from A.
        end do
        20  call pclose(3)                     ! close the file
        stop
        end
```

Figure 12: Sample code for out-of-core computation

5 Persistent Object Store

Aside from attempting to devise alternative data access techniques within the framework of existing conventions and standards (e.g., stream oriented files, block oriented devices), additional approaches are being investigated. One such approach is the development of persistent object storage systems.

Persistent object storage systems have long been in use in conjunction with object oriented databases. In these systems, all data are treated as distinct objects, each object belonging to a class which define attributes such as type information, permissible access methods for that object, etc. In persistent object models, object attribute information - called metadata - are stored along with the object data.

Most parallel file systems today provide little support for specifying the desired distribution of data across the disk array. The programmer is left with the responsibility (and the burden) of managing distribution mappings in the application by incorporating appropriate loop indices around data access operations. Applications created in this manner are inflexible, tied closely to the declustering scheme and I/O hardware architecture of the development machine.

Object oriented representation can potentially allow users to deal with data distribution at a higher semantic level, specified merely as a type parameter to an access method associated with the desired object. The addition of persistent object store functionality extends this paradigm to the storage system. The approach taken here is to combine mechanisms that have proven to be successful in parallel database systems with object oriented database and programming language technology. The target architecture for such systems is the tightly coupled I/O configuration discussed in section 2.2. A scalable persistent object store would consist of a file system providing management of strongly typed persistent objects in which:

1. files are collections of typed objects
2. a variety of mechanisms are provided for aggregating objects (e.g., bags, sets, arrays, etc.)
3. metadata are themselves persistent objects.

The structure imposed by typing requirements allow object components to be easily identified and distributed across the set of I/O devices at the point of creation. At some later time, data can be reorganized (i.e., alternative component subdivisions can be utilized) according to access pattern constraints, without modification of application code. For example, as illustrated in figure 13, data (possibly very large aggregates) may be distributed among the disks in agreement with the desired data access pattern for computation. Access to a matrix element (an object) by processor P1 initiates a request that is sent to processor P4 whose disk contains the desired object. At some later point in time, if the matrix elements (objects) are cached among the processors as shown, the data can simply be remapped to the disks accordingly.

In persistent object storage systems, metadata is preserved along with the data to be stored so that object type information is always available. This extra information (which would have been "lost" in a Unix file system) can be used by the parallel I/O subsystem in improving the data access strategy it devises. Explicit file access would no longer be necessary. Access to memory or file objects can be made indistinguishable. Further, I/O operations within applications would no longer be architecture dependent.

6 Networking Technology

When data has to be transferred from one computing environment to some external device or some other remotely connected supercomputer (e.g., for pre or post-processing, visualization, etc.), then

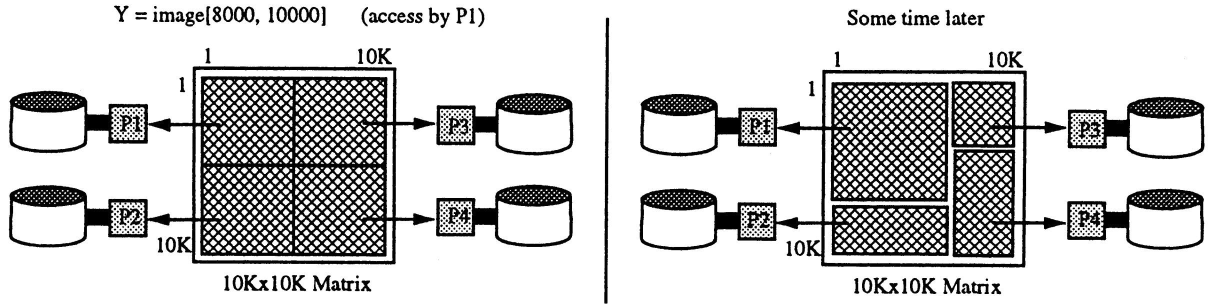

Figure 13: Distribution of persistent objects across disks

the capacity of the network becomes a critical consideration. Just like I/O devices, network capacity has lagged behind memory and processor technology. This is illustrated in figure 14 for the case of local area networks; wide area network technology has had an even flatter growth pattern over the last decade [Cat92].

In the past few years, a number of technologies have been developed to improve network interface and capacity. The High Performance Parallel Interface (HiPPI) standard for instance, includes a mapping to IEEE 802.2 for supporting common network protocols like TCP/IP. Interfaces to alternative layers are also under development. For example, the Intelligent Peripheral Interface (IPI-3) provides command sets for disk and tape, and will allow support for striped disks and tape devices directly connected to HiPPI channels or LANs.

Another technology being developed by IBM in collaboration with Lawrence Livermore National Laboratory, is called Fibre Channel. Aside from the use of an optical (as opposed to copper) medium, Fibre Channel encapsulates a more extensive set of services than does HiPPI. Unlike HiPPI, it targets up to 4K switch connections for distances of up to several kilometers. Further, Fibre Channel supports multiple connection types (e.g., datagram, virtual circuit, etc.) over a variety of physical layers (e.g, coax cables, fiber, lasers, LEDs, etc.) at multiple data rates.

A recent standard, the Scalable Coherent Interface (SCI), makes possible the development of local area networks with speeds of up to 8 Gb/s and is about 10 times as fast as Futurebus+. SCI provides bus services (read, write, lock, etc.) by using sending packets over a large number of point-to-point links.

A summary of existing technologies is presented in table 5.

Over the years, LANs and WANs have developed along independent, sometimes divergent, lines. With the growing support for distributed computing paradigms, heterogeneous networks will come into more extensive use. This should act as a driving force to integrate LAN and WAN technologies. The Asynchronous Transfer Mode (ATM) technology offers the possibility of making this possible. Implemented as the underlying support layer for B-ISBN, it provides a common framework for public-wide area networks as it does for local area networks. ATM facilitates this integration by providing the following [LMT93]:

1. Upper layer services which are all supported by a single transfer mechanism.
2. The ability to interconnect a wide range of bandwidths and localities (i.e., distances). Point-to-point links improves scalability over shared media networks.
3. Separation of basic transfer mechanisms from control mechanisms (i.e., separate in-band and out-of-band mechanisms) allows for increased efficiency.

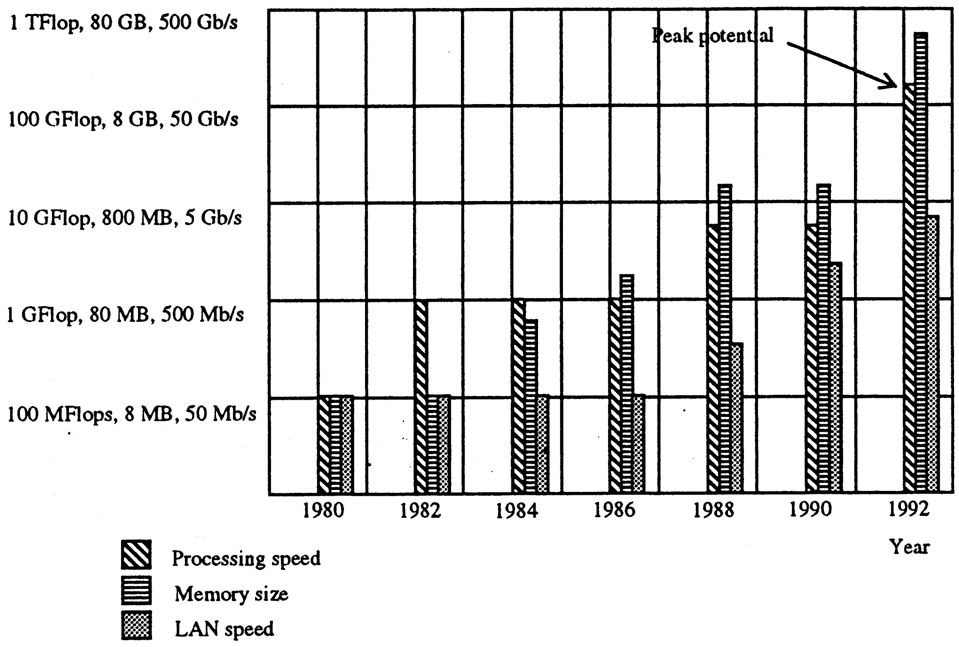

Figure 14: Growth in local area network capacity [Cat92]

Table 5: Network Technology

Type	Bandwidth	Distance	Technology
Fibre Channel	100-1000 Mb/s	LAN, WAN	Fiber optics technology
HiPPI	800 Mb/s or 1.6 Gb/s	$\leq$ 25 m	Copper cables (32 or 64 lines)
Serial-HiPPI	800 Mb/s or 1.6 Gb/s	$\leq$ 10 KM	Fiber optics channel
SCI	8 Gb/s	LAN	Copper cables
Sonet/ATM	55-4.8 Gb/s	LAN, WAN	Fiber optics technology
N-ISDN	64 kb/s, 1.5 Mb/s	WAN	Copper cables
B-ISDN	$\leq$ 622 Mb/s	WAN	Copper cables

LAN - up to several meters
WAN - up to several kilometers

ATM is a mode of transfer where the information is encapsulated in fix sized "cells". It is "asynchronous", according to the CCITT definition [CCI90], in the sense that "the recurrence of cells containing information from an individual user is not necessarily periodic." Virtual channels are employed to allow simultaneous connectivity to large numbers of peers. In some implementations, point-to-point links between switches allows the ATM network to increase in bandwidth as hosts are added; thus, they are scalable. Further, data transfers between any number of hosts can occur in parallel [BCS93].

As technologies such as ATM are developed and mature, our ability to share information over long distances will be improved. This will play a significant role in determining the future of high performance distributed computing.

7 Summary

In this paper we have presented some issues in high-performance I/O for massively parallel computers. We discussed some of the questions that remain to be answered and provide some examples of alternative solutions.

To summarize, the recurrent themes in the parallel I/O problem are the existence of great variety in access patterns, and the sensitivity of current I/O systems to these access patterns. An increase in the variability of access patterns is also expected to occur, and single resource management approaches will likely not suffice. Providing the I/O infrastructure that will support these requirements will require research in: operating systems (parallel file systems, runtime systems, drivers), language interfaces to high-performance storage systems, high-speed networking, graphics and visualization systems, and new hardware technology for I/O and storage systems. As a first step to this research, I/O access patterns have to be quantitatively characterized by instrumenting multiple platforms and collecting trace data for large application codes. The knowledge gained from this step has to be integrated into an evolutionary development cycle for I/O systems as a whole.

The area of parallel I/O is a vast one, with aspects related to many areas of computing in general. In the limited space of this paper we were able to give, at best, only a brief overview of the issues and concerns related to this problem. The following is a list of relevant areas that have not been discussed here.

1. Multi-media requirements
   - real-time, high bandwidth video, audio, imaging, etc. place different demands on the I/O system
   - the synchronization problem that arises from using different media and data types
2. Database systems
   - access patterns differ from scientific computing
   - I/O requirement is for throughput rather than bandwidth
   - cache management techniques crucial
3. Parallel data transfer
   - e.g., how do we do parallel ftp over parallel channels?
   - support for range of data decompositions at network endpoints

- asynchronous transfers

4. Distributed file system (over HPDC network)

5. Archival storage

   - high-speed tape
   - optical devices

6. Visualization

Acknowledgment: This paper has been influenced by the following people; many of the ideas that have been presented here have come from their lectures or from the many informative discussions the authors have had with them: Brian Bershad, Rajesh Bordawekar, Andrew Chien, Tom Corman, David DeWitt, Denise Ecklund, Geoffrey Fox, Bill Gropp, Ken Kennedy, David Kotz, Kai Li, Paul Messina, Regan Moore, David Payne, Terry Pratt, A.L.N. Reddy, Dan Reed, Joel Saltz, and Rick Stevens.

References

[BCdR93] Rajesh Bordawekar, Alok Choudhary, and Juan Miguel del Rosario. An Experimental Performance Evaluation of the Touchtone Delta Concurrent File System. In *Proceedings of the International Conference on Supercomputing '93*, Tokyo, Japan, July 1993.

[BCS93] Edoardo Biagioni, Eric Cooper, and Robert Sansom. Designing a Practical ATM LAN. *IEEE Network*, pages 32–39, March 1993.

[Cat92] Charles E. Catlett. Balancing Resources. *IEEE Spectrum*, pages 48–55, Sept 1992.

[CC93] High Performance Computing and Communications. Grand Challenges 1993 Report. *A Report by the Committee on Physical, Mathematical, and Engineering Sciences Federal Coordinating Council for Science, Engineering and Technology*, 1993.

[CCI90] CCITT. *Draft Recommendation I.113, Section 2.2, Geneva*, May 1990.

[Cro89] Thomas W. Crockett. File concepts for parallel I/O. In *Proceedings of Supercomputing '89*, pages 574–579, 1989.

[Dav90] David Kotz and Carla S. Ellis. Prefetching in File Systems for MIMD Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, pages 218–230, April 1990.

[Int91] Paragon XP/S Product Overview. Intel Corporation, 1991.

[LMT93] Ian M. Leslie, Derek R. McAuley, and David L. Tennenhouse. ATM Everywhere? *IEEE Network*, pages 40–46, March 1993.

[Pag92] Jon Page. A Study of a Parallel Database Machine and its Performance. In *Advanced Database Systems: 10th British National Conference on Databases, BNCOD 10; Aberdeen, Scotland*, pages 115–137. Springer Verlag, July 1992.

[PGK88] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

[RBA88] A.L.Narashima Reddy, P. Banerjee, and Santosh G. Abraham. I/O Embedding in Hypercubes. In *Proceedings of the 1988 International Conference on Parallel Processing*, 1988.

[Thi91] Thinking Machines, inc. *The Connection Machine CM-5 technical summary*, Oct 1991.

[vECSK92] Thorsten von Eicken, David E. Culler, Seth Copen Schauser, and Erik Klaus. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992. ACM Press.