Remote I/O Optimization and Evaluation for Tertiary Storage Systems through Storage Resource Broker

Xiaohui Shen, Wei-keng Liao and Alok Choudhary
Center for Parallel and Distributed Computing
Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208
{xhshen,choudhar}@ece.nwu.edu

Abstract

Large-scale parallel scientific applications are generating huge amounts of data that tertiary storage systems emerge as a popular place to hold them. SRB, a uniform interface to various storage systems including tertiary storage systems such as HPSS, UniTree etc., becomes an important and convenient way to access tertiary data across networks in a distributed environment. But SRB is not optimized for parallel data access: one SRB I/O call to storage systems must access a contiguous piece of data just like UNIX I/O. For many access patterns, this results in numerous small I/O calls which are very expensive.

In this paper, we present a run-time library (SRB-OL) for optimizing tertiary storage access on top of SRB low level I/O functions. SRB-OL extends various state-of-the-art I/O optimizations that could be found in secondary storage systems to a remote data access environment via SRB. We also present a novel optimization scheme: *superfile* that can deal with large amounts of small files efficiently. We also incorporate a *subfile* technique and other features in SRB such as container, migrate, stage and purge into our SRB-OL. How to use these optimizations is decided by a Meta-data Management System (MDMS) [7] that resides one level above SRB-OL. The user provides access pattern information/hints through user application to MDMS, and then MDMS uses these hints to choose an optimal I/O approach and passes the decision to SRB-OL. Finally, SRB-OL performs optimized SRB I/O calls to access data residing on tertiary storage systems.

To give a quantitative view of optimized SRB I/O functions, we propose a performance model based on significant I/O experiments. By using this performance model, we can prove that collective I/O, superfile etc have significant performance improvements. In addition, we present an *I/O Performance Predictor* that can estimate I/O cost before the user actually carries out her experiment. This provides the user a lot of benefits for running her application.

1 Introduction

More and more modern large-scale simulations employ data management techniques to effectively manage huge amounts of data that are generated by parallel machines. Our previous work [7] is a first step toward considering I/O optimizations as well in data management systems. We have designed a Meta-data Management System (MDMS) at Northwestern University that provides both ease-of- use and I/O optimizations for large-scale I/O intensive applications. The most significant feature of our MDMS is that it automatically provides best I/O optimizations such as collective I/O, prefetch and so on if the user provides access pattern and storage pattern information. Here the access pattern spans a wide spectrum that contains how data is partitioned among internal processors, information about whether the access type is read-only, write-only, or read/write, information about the size (in bytes) of average I/O requests, information about how frequently the generated data will be used and so on. For example, if the data is partitioned in a (Block, Block) way for a two dimensional array among processors and they are to be stored in a (Block, *) way, our MDMS will advise the application using collective I/O to achieve high performance.

Our MDMS was first built on top of MPI-IO to take advantage of I/O optimizations found in MPI-IO. But it is now typical for modern scientific simulations to adopt tertiary archival as permanent storage for holding huge amounts of datasets generated by parallel processors. These tertiary storage systems such as High Performance Storage System (HPSS) [3], Unitree [10] etc., usually have disks and tapes to form a storage hierarchy inside the systems which is one level below the magnetic disks. The characteristics of this storage hierarchy are that the data stored are large and the data access time is very slow: the access time of a typical I/O call for a tape-resident dataset would be more than two orders slower than from magnetic disks. In addition, many scientific applications are I/O intensive; this means I/O would dominate the overall performance for many applications. Therefore, aggressive I/O optimizations for tertiary data are critical for modern simulations. Unfortunately, MPI-IO is developed and optimized for secondary storage systems rather than tertiary storage systems, therefore, when our applications shift to tertiary storage systems, we have to turn to other interfaces for them.

We used HPSS at SDSC as the platform for development and evaluation. In principle one would use the native interface to develop runtime system, however the native interface to HPSS is normally reserved for system support people. We accessed HPSS using Storage Resource Broker (SRB) [1, 8] which is developed by San Diego Supercomputer Center (SDSC). SRB is a client-server middleware that provides a uniform interface for connecting to heterogeneous data resources over a network and accessing replicated datasets. Using SRB has some advantages.

- It provides a uniform interface for various storage systems: HPSS, UniTree, UNIX disks, databases and so on, therefore the user can easily experiment with different storage systems without changing the interface.

- It hides the details of storage systems and provides a small number of UNIX like I/O functions that are very easy to learn and use. HPSS, for example, consists of hundreds of internal API functions which would be overkill for a general scientific user. In addition, due to the complex nature of HPSS, its internal API is not available for typical application level users.

- It provides an easy way to access tertiary archival systems. The SRB allows users to access storage systems through networks over a long distance, so users can develop and run their applications locally and can still store/load their data on/from remote storage systems. Given the fact that tertiary storage systems are not so popular as parallel machines, this provides users a lot of convenience for tertiary data access.

We have employed SRB as a medium for tertiary data access. But, SRB also has some drawbacks: its I/O calls are not optimized for parallel data access. Like UNIX I/O, each SRB I/O call only accesses a contiguous piece of data. For noncontiguous data such as each processor accessing a subarray in a large array, which is very popular in parallel scientific applications, the user is forced to issue multiple I/O calls with each I/O call accessing a contiguous portion of data. In addition, SRB server may be far away from SRB client across networks, network failure and delay could seriously hurt the performance. Therefore, making less number of remote data accesses is crucial for performance. In this paper, we present a *SRB Optimization Library* (SRB-OL) that provides various state-of-the-art I/O optimizations on top of SRB or through SRB's features for accessing tertiary storage systems. Figure 1 shows the architecture of our library. The SRB-OL takes I/O optimization decisions from MDMS and performs corresponding optimizations through SRB. These optimizations include remote collective I/O, data sieving, subfile, superfile, container, migration, stage and purge. We also present a performance model for accessing HPSS through SRB and quantitatively demonstrate that our optimizations have significant performance improvements.

The remainder of the paper is organized as follows. In Section 2 we introduce HPSS, the tertiary storage system used in this paper. We also give an overview of SRB in this section. In Section 3 we present our performance model based on significant experiments, and we use this performance model for I/O analysis

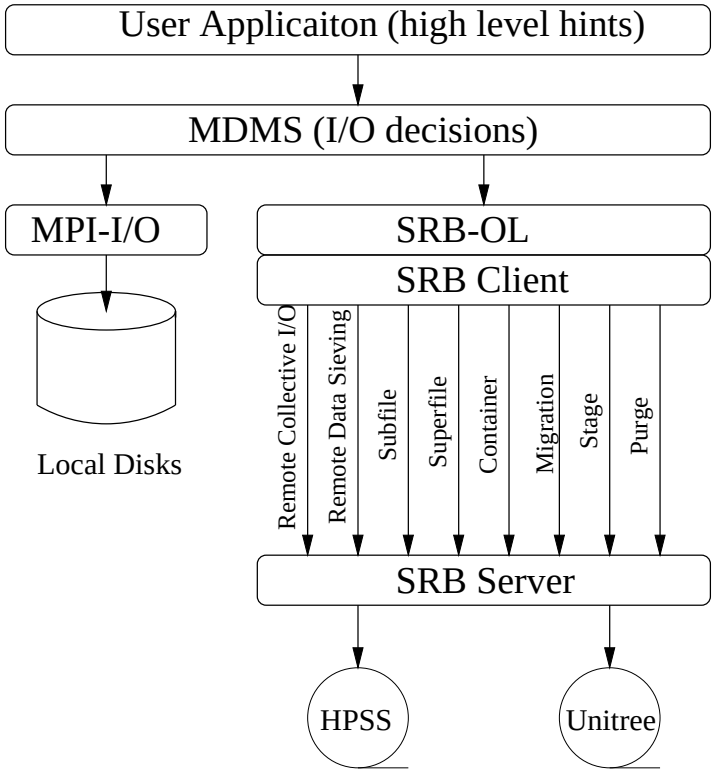

Figure 1: **System Architecture of I/O optimizations through SRB. The user provides high level hints to MDMS and MDMS makes I/O decision and passes the decision to either MPI-IO for local secondary storage system access or SRB-OL for accessing remote tertiary storage systems through SRB.**

in the subsequent sections. In Section 4, various I/O optimizations are presented and I/O performance is analyzed by our performance model. We present an *I/O Performance Predictor* in Section 5 which can provide an overview of I/O cost before performing an experiment, so that the user then can have better evaluation of her experiments. In Section 6 we present relationship between our previously developed MDMS and our current SRB-OL and we discuss how various optimizations in SRB-OL are used by MDMS. We conclude the paper in Section 7 and some future work is also presented.

2 Introduction to HPSS and SRB

2.1 HPSS

The High Performance Storage System (HPSS) [3, 4] is software that provides hierarchical storage management and services for very large storage environments. HPSS may be of interest in situations having present or future scalability requirements that are very demanding in terms of total storage capacity, file sizes, data rates, number of objects stored, and number of users. Most other storage management systems support simple storage hierarchies consisting of one kind of disk and one kind of tape. HPSS provides multiple hierarchies, which are particularly useful when inserting new storage technologies over time. As new disks, tapes, or optical media are added, new classes of services can be set up. HPSS files reside in particular classes of services which users select based on parameters such as file size and performance. A class of service is implemented by a storage hierarchy which in turn consists of multiple storage classes. Storage classes are used to logically group storage media to provide storage for HPSS files. A hierarchy may be as simple as a single tape, or it may consist of two or more levels of disk, disk array, optical disk,

Table 1: HPSS Class of Service (COS).

COS	File Size
tiny	0KB-8KB
small	8KB-2KB
medium	2MB-200MB
large	200MB-6TB
tape	6TB+

local tape, and remote tape. The user can even set up classes of services so that data from an older type of tape is subsequently migrated to a new type of tape. Such a procedure allows migration to new media over time without having to copy all the old media at once. Table 1 shows the class of service supported by HPSS and their corresponding file sizes.

We only use two levels of hierarchy in HPSS for simplicity, i.e. disks and tapes. The permanent copy of data resides on tapes, when the user sends a read request, data will be first copied to disks from tapes, which is called *stage* and then data will be read from disks to user's buffer. When the user writes data to HPSS, the data is written to disks first and then periodically, HPSS migrates data to tapes. The disk copy may also be freed from disks, which is called *purge*, according to HPSS's migration/purge policy. We can see that the disks here serve as a cache for tapes.

2.2 SRB

The Storage Resource Broker (SRB) [1, 8] is a middleware that provides distributed clients with uniform access to diverse storage resources in a heterogeneous computing Environment. Figure 2 gives a simplified view of the SRB architecture. The model consists of three components: the meta-data catalog (MCAT) service, SRB servers and SRB clients, connected to each other via network. The MCAT stores meta-data associated with datasets, users and resources managed by the SRB. The MCAT server handles requests from the SRB servers. These requests include information queries as well as instructions for meta-data creation and update. Client applications are provided with a set of API for sending requests and receiving response to/from the SRB servers. The SRB server is responsible for carrying out tasks to satisfy the client requests. These tasks include interacting with the MCAT service, and performing I/O on behalf of the clients. A client uses the same common API to access every storage systems managed by the SRB. SRB automatically selects HPSS class of service based on file size (Table 1).

SRB has two suites of APIs, one is called high level API and one is low level API. The difference is that the high level API provides abstraction of file name, file location etc. and the user only need to provide the object name to access her data. While in the low level API, like UNIX I/O, the user needs to provide file name, location, offset etc. explicitly. We chose SRB low level API as I/O functions to access HPSS in order to be able to specify which I/O functions to use so that optimizations can be performed and controlled. We call these low level functions SRB I/O calls/functions henceforth.

3 Performance Model

Figure 4 shows the architecture of our experimental environment for the performance model. To access the data that resides on HPSS tapes, the data is first staged to disks from tapes and then it moves upward through networks and arrives at user's memory buffer. The I/O time or cost (T) is defined as the time spent on moving data from HPSS (either on its tapes or disks) to user's memory buffer. We only consider read

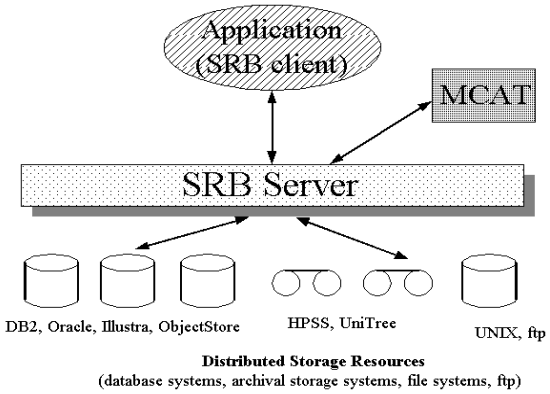

Figure 2: **SRB client-server architecture.**

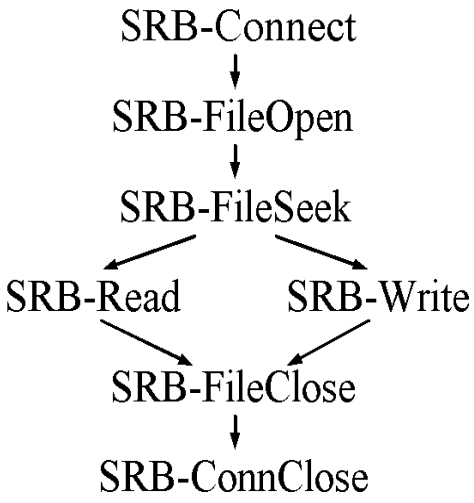

Figure 3: **SRB low level API.**

time for our performance model in this paper. A typical SRB low level I/O flow, reading data from or writing to HPSS, is shown in Figure 3.

SRB-Connect establishes the communication connection between SRB client and server across the network, $SRB - FileOpen$ opens a file in HPSS, $SRB - FileSeek$ finds the location in the file for read or write, and $SRB - Read$ and $SRB - Write$ will transfer data between SRB client which connects to the user's memory buffer and SRB server which connects to HPSS. Finally $SRB - FileClose$ closes the opened file and $SRB - ConnClose$ closes the current connection.

Therefore, the total I/O time T equals the sum of time spent on connection, open and read etc.

$$T = T_{conn} + T_{open} + T_{seek} + T_{read} + T_{fileclose} + T_{connclose} \quad (1)$$

Unlike UNIX I/O that works on random access devices, there is one more issue with SRB and HPSS: when does the stage take place? HPSS allows it to take place either during the file open or the first read to the file. According to the SDSC's configuration, for most classes of services in HPSS, files are staged completely during open. Therefore, T_{open} can be broken down to $T_{pureopen}$ and T_{stage}, where $T_{pureopen}$ is the time for

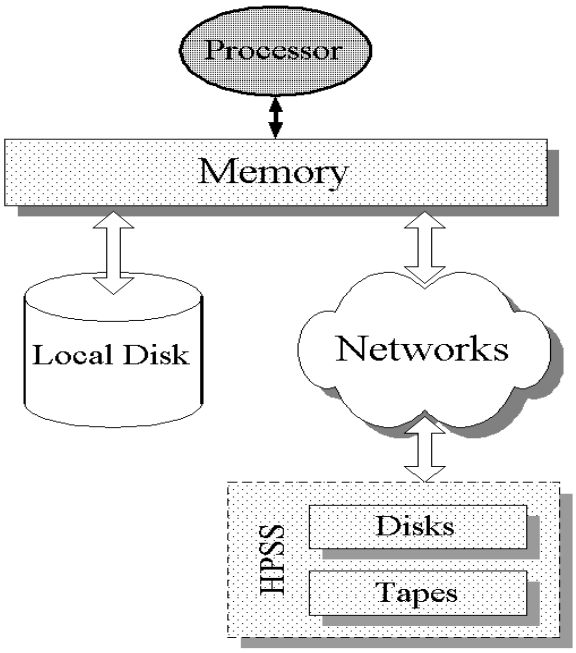

Figure 4: **System Data Flow. The I/O time is defined as the time spent on moving data from HPSS (disks or tapes) to the user's memory buffer (read).**

a normal file open and T_{stage} represents the time for staging data from tapes to disks. We rewrite the above equation as follows for one read I/O call.

$$T = T_{conn} + T_{pureopen} + T_{stage} + T_{seek} + T_{read} + T_{fileclose} + T_{connclose} \quad (2)$$

To set up our performance model, we need to establish time for each component in the above equation quantitatively. T_{stage} is a factor that can not be completely measured at present. It depends on several factors: available disk cache, available tape drive(s), robot latency, file size, and location on tapes. It is fairly unpredictable. According to SDSC's HPSS, the tapes require a minimum of 20 to 40 seconds to be ready and then move the data to disks. We once observed that the T_{stage} could be more than 70 seconds for a small file on HPSS tapes. We can not provide more exact timings for T_{stage} as of this time due to both unpredictable nature of T_{stage} and our limited privileges on HPSS. But in any cases, T_{stage} is very expensive.

To measure other components in the above equation, we performed the following experiments. After just writing a file, we immediately read it. Therefore, the measured T_{open} should be $T_{pureopen}$ because the data has not been migrated and purged to tapes. We also time T_{conn} and T_{read}. To measure T_{seek}, we did another experiment. We measured the seek time for different locations in a large file.

We did these experiments on a local IBM SP2 at Center for Parallel and Distributed Computing (CPDC henceforth) of Northwestern University. The HPSS is located at San Diego Supercomputer Center (SDSC). Figure 5 show the results of these experiments. We made the following observations:

T_{conn} is time for SRB to establish connection between its client and server, so we observed a fixed overhead. We measured that the average connection time is about 0.69 seconds. (Figure 5).

$T_{pureopen}$ is time for opening a file in HPSS. It is also a constant overhead independent of file size. We measured that it is about 5.11 seconds.

We find T_{seek} is quite stable, averaging 0.20 seconds independent of locations in the file. This is because file seek takes place on random-access devices: HPSS disks (file has already been on disks after open).

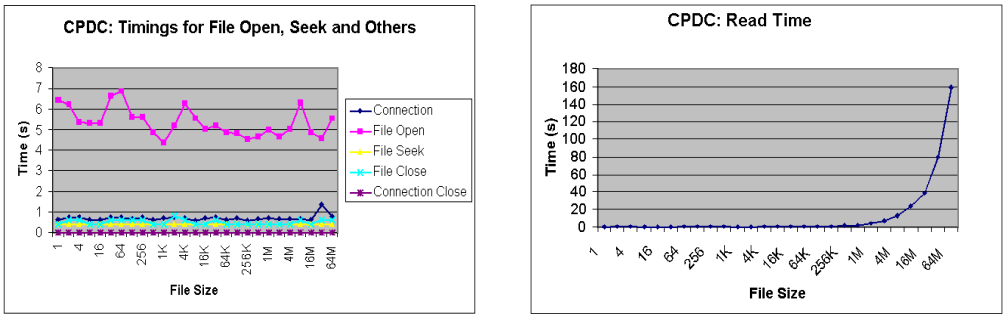

Figure 5: **Timings for SRB file Open, seek, close (left) and read (right)**

Table 2: Average Values of File Open, Seek and Others.

Location	T_{conn}	$T_{pureopen}$	T_{seek}	$T_{fileclose}$	$T_{connclose}$
CPDC	0.71	5.36	0.42	0.52	0.00004

$T_{fileclose}$ and $T_{connclose}$ are also fixed costs independent of file size. We observed that $T_{fileclose}$ is about 0.31 seconds. $T_{connclose}$ is very small that can be ignored (less than 0.001s). Table 2 shows the average values of these timings.

T_{read} is a little more complicated than the other factors that we discussed above. Conceptually, it can be further broken down into SRB server and client overheads ($T_{SRBoverhead}$) and data transmission time across networks $T_{transmission}$ etc. We find that when the data size is small (less than 8KB for CPDC), T_{read} is almost a constant: averaging 0.56s. This is in part due to the underlying fixed packet size of Internet Protocol and in part due to the fixed overheads such as $T_{SRBoverhead}$ that dominate the overall time. As data size goes up, we can see that T_{read} increases almost linearly for large data size. This means the data transmission time for large data dominates the overall read time. We observed that T_{read} is about 2.51×data size (in MB) ($T_{read}(s) = 2.51s$ where s is the data size (in MB) and $s > 16MB$). Note that s is not limited to 64MB as show in Figure 5. For example, when $s = 1GB$, $T_{read}(s) = 2.51 \times 1024(MB) = 2570seconds$, and our experiment shows that the read time is $2482seconds$.

In general, for the small data size (less than 8KB), the cost of a single SRB I/O call is:

$$\begin{aligned} T &= T_{conn} + T_{pureopen} + T_{stage} + T_{seek} + T_{read} + T_{fileclose} + T_{connclose} \\ &= T_{stage} + C \end{aligned} \quad (3)$$

Where C is a constant. C can be easily calculated as 7.57s (less than 8KB). For large data sizes (s>16M), we have

$$\begin{aligned} T(s) &= T_{conn} + T_{pureopen} + T_{stage} + T_{seek} + T_{read}(s) + T_{fileclose} + T_{connclose} \\ &= T_{stage} + Ks + L \end{aligned} \quad (4)$$

Where $T_{read}(s) = Ks$ and s is data size (in MB). K and L are constants: 2.51 and 7.01. Table 3 shows K, L and C values. Note that Equations 3 and 4 also apply to data size exceeding 64MB, although we only show data size less than 64MB in Figures 5.

From Equation 3 and Equation 4 we can easily calculate the I/O time for a single SRB I/O call for reading small and large data files given the value of T_{stage}. For the middle sized data file, we can consult Figure 5 for exact values.

One usage of these equations is that before we really carry out the experiments, we can calculate in advance the estimated time that will be spent on I/O. So the user can better evaluate her experiment.

Table 3: Average values of K, L and C.

Location	K	L	C
CPDC	2.51	7.01	7.57

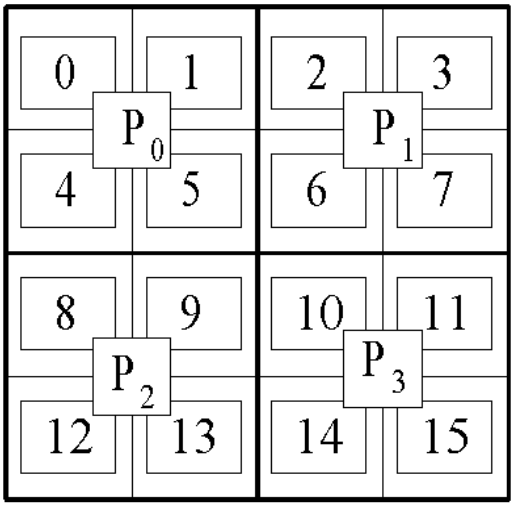

Figure 6: **A two dimensional array and data partition among 4 processors.**

4 SRB I/O Optimizations and Experiments

In this section, we present various I/O optimization techniques employed in our SRB-OL (Figure 1). These optimizations are classified into two categories: one is generic which includes remote collective I/O, remote data sieving, subfile and superfile. These optimizations are independent of storage types, suitable for all remote data accesses; one is storage specific which includes container, stage, migration and purge. The experiments are run on a 16 node IBM SP2 at CPDC of Northwestern University. Each node of the SP2 is RS/6000 Model 390 processor. The HPSS is located at SDSC. We also use the performance model in the previous section to analyze the results that we obtained.

4.1 Remote Collective I/O

Collective I/O [9, 2, 5] is a technique to glean individual I/O requests of each processor into a global picture and then decide how the combined I/O requests should be best performed. The two-phase I/O [9, 2] is one of significant implementation techniques of collective I/O. In the two phase scheme, the data is accessed based on the conforming distribution from I/O system in the first phase, in phase two, data is redistributed at run-time among processors. Consider a 4×4 two dimensional array that is stored on HPSS. Suppose the data is partitioned in (Block, Block) among 4 processors (Figure 6), so each processor will access a subarray of data. For example, processor 0 will access 0, 1, 4, 5 and processor 1 will access 2, 3, 6, 7. In the naive I/O approach, each processor has to issue 2 SRB I/O calls to HPSS. For instance, in the first I/O call processor 0 reads 0 and 1, and in the second call, it reads 4 and 5. This is because the data is not contiguous for 0, 1 and 4, 5. In general, suppose the array is $N\times N$, the application is run on 4 processors, each processor would issue $N/2$ SRB reads each with $N/2$ data items. According to Equation 2, we have

$$T = T_{conn} + T_{pureopen} + T_{stage} + N/2(T_{seek} + T_{read}) + T_{fileclose} + T_{connclose} \quad (5)$$

We can even calculate the total I/O time without performing the experiment. Suppose $N = 1024$ and the data element is character. So by referring to table 2, we have

$$\begin{aligned} T &= 0.71 + 5.36 + T_{stage} + 512(0.42 + 0.56) + 0.52 + 0.0003 \\ &= T_{stage} + 508.35 \end{aligned} \quad (6)$$

Where $T_{read}(512) = 0.56$. As T_{stage} is quite hard to measure and may vary significantly, we do not consider it at this time, i.e. we guarantee there is a disk copy of data on HPSS. (An aggressive prestage optimization can also eliminate this item.) So $T_{stage} = 0$ and $T = 508.35$ seconds. This calculation is commensurate with our experiment: 506.74 seconds (Figure 7). In general, if the user issues n I/O calls, $T = O(T_{stage} + n)$ for small data size and $T = O(T_{stage} + ns)$ for large data size (s is data size in MB). In any cases, number of I/O calls is a critical factor for performance. Therefore, decreasing the number of I/O calls is the target of collective I/O approach.

In two phase implementation, each processor will access one contiguous section in one I/O call during the first phase. For example, processor 0 would read 0, 1, 2 and 3, processor 1 reads 4, 5, 6 and 7 etc. In a remote data access environment, even the underlying storage does not provide data striping for parallel data access, remote collective I/O can still benefit from parallel data transfer which is the dominate factor of performance. In the second phase, data is redistributed among the processors themselves. The redistribution of data introduces extra internal communication cost, but compared to expensive remote I/O access, the internal communication cost can be ignored. Again, we can calculate the results first when collective I/O is performed:

$$\begin{aligned} T &= T_{conn} + T_{pureopen} + T_{stage} + T_{seek} + T_{read} + T_{fileclose} + T_{connclose} + T_{commcost} \\ &= 0.71 + 5.36 + T_{stage} + (0.42 + 4.5) + 0.52 + 0.0003 + 0.15 \\ &= T_{stage} + 11.66 \end{aligned} \quad (7)$$

Where $T_{read} = 4.5$ according to Figure 5. It also commensurates with our experiment 13.12 seconds ($T_{stage} = 0$). We can see that collective I/O significantly reduced the I/O times. In general, given number of I/O calls (n) and each with size s, we have

$$\begin{aligned} T_{noncol} - T_{collective} &= n(T_{seek} + T_{read}(s)) - (T_{seek} + T_{read}(ns) + T_{commcost}) \\ &= (n-1)T_{seek} + (nT_{read}(s) - T_{read}(ns)) - T_{commcost} \end{aligned} \quad (8)$$

In general, nT_{read}(s) is significantly larger than $T_{read}(ns)$ according to Figure 5. When the data size is large (s>16MB), $T_{read}(ns) = Kns$ and $nT_{read}(s) = nKs = Kns$ according to Equation 4, so $nT_{read}(s) = T_{read}(ns)$. Therefore, collective I/O can save at least $(n-1)T_{seek}$ which is O(n) ($T_{commcost}$ can be ignored compared to $(n-1)T_{seek}$).

Figure 7 shows performance numbers of collective I/O compared to the naive approach.

4.2 Data Sieving

If the user requests several small, noncontiguous accesses, data sieving [9] is helpful to reduce the I/O latency. The basic idea is to make fewer large contiguous I/O requests and later extract what the user's actual request in main memory. Data sieving reads more than needed, but the benefit of reducing number of I/O calls outweighs the cost of reading unwanted data. Consider Figure 6 again, if processor 0 reads items

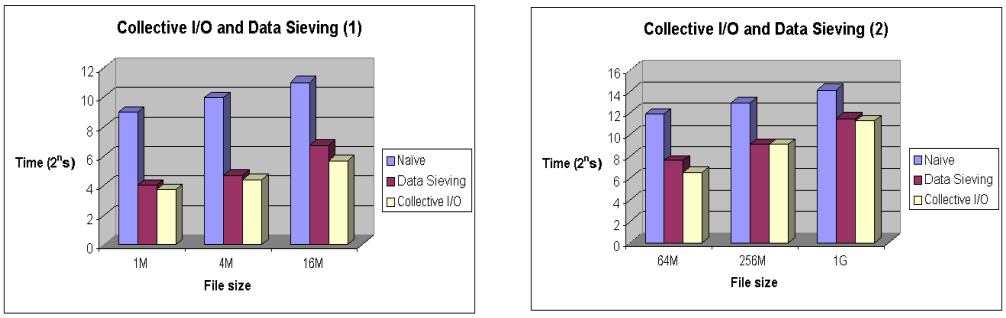

Figure 7: **I/O time for Collective I/O and Data Sieving for small (left) and large (right) data sizes. The data is stored in a two dimensional array on HPSS disk. It is partitioned in (Block, Block) among 4 processors. Note that the Y-axis is in log scale.**

0, 1, 4, 5, the data sieving optimization would read 0, 1, 2, 3, 4 and 5 in one I/O call, and then discard 2, 3 in memory. As more data are read in data sieving and each processor has to establish its own I/O connection, data sieving is slightly worse than collective I/O in performance, but compared to the naive approach, we observed significant performance improvements (Figure 7).

4.3 Subfile

In many tape-based storage systems, the access granularity is a whole file [10]. Consequently, even if the program tries to access only a section of the tape-resident file, the entire file must be transferred from the tape to the upper level storage media (e.g., magnetic disk). This can result in poor I/O performance for many access patterns. Subfile [6] is a technique that transparently breaks up a large data file into smaller subfiles in storage. When the user requests a small portion of data, one or several subfiles can be served. This technique can significantly reduce number of I/O calls, therefore improves the performance. Details of subfiling techniques can be found at [6].

4.4 SRB Container

Due to the relative high overhead of creating/opening files in archival storage systems such as HPSS, they are not suitable for storing large amount of small files typically found in digital library systems. The SRB container concept [8] was specifically created to handle this type limitation. Through the use of containers, many small files can be aggregated before storage in the archival storage system. The implementation is that SRB maintains a 'permanent' (e.g., HPSS) and a 'temporary' or 'cache' (e.g., UNIX disk file system) resource for a container. Internally, all I/O are done only to the 'cache' copy of the container. Figure 8 shows some performance numbers for container. In this experiment, we assume there are 10 small files, they are read sequentially by one processor. The total I/O time is measured. Since we assume $T_{stage} = 0$, so at first glance, one may think container should have roughly the same performance number as the naive approach. But note that the container is working with SRB *high level* API, so its I/O performance does not belong to our performance model (Section 3). So we observed that for smaller data sizes (1K×10, 16K×10 and 100K×10), container has better performance. For larger data sizes (1M×10, 2M×10 and 4M×10), its performance is worse. But in the future when the T_{stage} is taken into account, the naive approach would have 10 T_{stage}s and container has only one, so we should observe significant performance improvements for all data sizes.

4.5 Superfile

While container concept can improve the performance for accessing numerous small files, we notice that to access objects in container, the user still has to issue as exact number of I/O calls as before, although the objects can be guaranteed on disks most of the time. In an environment like ours, the application may be thousands miles away from the storage systems, the number of I/O calls will still dominate the I/O performance. Therefore, we propose a novel technique: *Superfile* for large amounts of small files. The basic idea is that when these small files are created, we transparently concatenate these small files into one large *superfile.* So later on, when the user reads these small files, the first read call brings the large *superfile* in main memory, so the subsequent reads to other files can be copied directly from main memory. Two database tables are created in our MDMS to help keep the filename, offset etc of each file in the *superfile.* Note that the database connection cost is very small. Before experiments, we calculate some results first. Suppose there are 10 small files, each is 1K. The naive approach would give:

$$\begin{aligned} T_{naive} &= T_{conn} + n(T_{pureopen} + T_{seek} + T_{read} + T_{fileclose}) + T_{connclose} \\ &= 0.71 + 10(5.36 + 0.42 + 0.56 + 0.52) + 0.0003 \\ &= 69.31 \end{aligned} \tag{9}$$

Using superfile, we have

$$\begin{aligned} T_{superfile} &= T_{conn} + T_{pureopen} + T_{seek} + T_{read} + T_{fileclose} + T_{connclose} + T_{db} + T_{memcpy} \\ &= 0.71 + 5.36 + 0.42 + 0.63 + 0.52 + 0.0003 + 0.5 + T_{memcpy} \\ &= 8.14 + T_{memcpy} \end{aligned} \tag{10}$$

Where $T_{read} = 0.63$ according to Figure 5 and in both equations $T_{stage} = 0$. We measured that the T_{db} is about 0.5s, T_{memcpy} is a small overhead. Our experiments show that the results of naive and superfile are 65.80s and 8.01s respectively which are consistent with our performance analysis.

In general, T_{naive} - $T_{superfile}$ is given by

$$\begin{aligned} T_{naive} - T_{superfile} &= (n-1)(T_{pureopen} + T_{stage} + T_{fileclose}) + (nT_{read}(s) - T_{read}(ns)) \\ &\quad -(T_{db} + T_{memcpy}) \end{aligned} \tag{11}$$

Ignore the last item and since s is small, so $nT_{read}(s)$ is larger than $T_{read}(ns)$. Therefore, $T_{naive} - T_{superfile} = O(n)$.

Figure 8 shows that *superfile* improved the performance near 10 times for smaller data sizes (1K×10, 16K×10 and 100K×10) and when the data sizes are larger (1M×10, 2M×10 and 4M×10), its performance improvement is still obvious. When T_{stage} is considered in the future, the performance improvements should be more obvious since in superfile there is only one T_{stage} compared to 10 T_{stage}s for the naive approach.

4.6 Migration, Stage and Purge

SRB provides some tape related functions such as *migration,* which advises HPSS move the data from disks to tapes; *stage* which brings the data from tapes to disks for prospective read by the user and *purge* which removes the data from HPSS disks. Our MDMS can aggressively utilize these functions for optimization. For example, if the newly generated data is not used in the near future, a *migration* hint can be issued to inform HPSS to migrate/purge the data as soon as possible, thus save the space of disk cache. Another example is that when the user finds that she is going to access a data file in the future, our MDMS can send a *stage* request through SRB to prestage data from tapes to disks. Unfortunately, these functions are set to privileged functions for system security and reliability and we are not able to experiment with them at present. But we incorporate them into our SRB-OL for possible future usage.

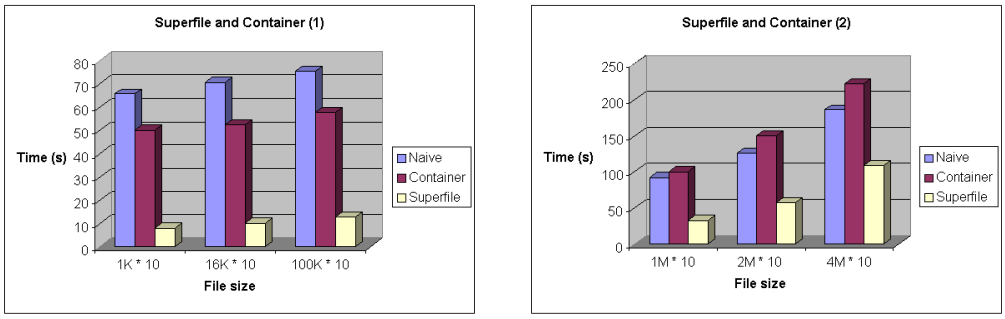

Figure 8: **Timings for Superfile and Container. 10 files are read sequentially by one processor and the total I/O time is measured. It is obvious that superfile improved the performance by a factor of 10 for smaller data sizes (left) and for larger data sizes (right), the performance improvement is also obvious. Note that container and naive approaches use different SRB APIs, so their performance numbers are not comparable when $T_{stage} = 0$ at present. In the future, when T_{stage} is taken into account, the performance of container would exceed that of naive approach for all data sizes since the naive approach suffers multiple T_{stage}s. In any cases, superfile approach is always the best.**

5 I/O Performance Predictor

As we can see in section 4, we can calculate in advance the I/O costs before we actually perform the experiments and our estimates are very close to the actual experimental results. Knowing how much I/O costs for the user's application before experiments (especially I/O intensive applications) has some advantages: the user can choose suitable parameters to obtain a time that she can be satisfied. These parameters include number of iterations, array sizes, frequency of I/O dumps and so on. The user can also compare how much I/O time can be saved by I/O optimizations such as collective I/O, superfile and so on.

We designed and implemented a general *I/O Performance Predictor* that can give a prediction of I/O cost based on our performance model. The user provides such information to the predictor: number of iterations, array sizes, number of datasets, frequency of I/O dumps, access patterns and so on [1]. The predictor takes these parameters and use the figure 5 which are stored in database to calculate the I/O cost based on the access pattern. Both naive and optimized costs are calculated and compared.

6 MDMS and SRB-OL

In section 4 we presented the SRB-OL which includes various state-of-the-art I/O optimization schemes on top of SRB low level I/O functions. The decision whether or not to use these optimizations and what kind of optimization should be used is the task of MDMS (Figure 1). The MDMS takes user specified information/hints from user application which is one level above MDMS and selects a suitable I/O optimization approach transparently [7]. These hints (access pattern) include how data is partitioned among processors, how large the data will be generated, how frequently the data will be used, what datasets are associated together and so on. These user level hints are high level in a sense that it is commensurate with human customs and behaviors. For example, the user could express (Block, Block) when she is going to partition a two dimensional array into subarrays among processors. When MDMS obtains this hint (Block, Block), it searches the database to see how data is stored on storage systems. If the data is stored in (Block, *) which does not conform the access pattern (Block, Block), then MDMS would suggest a collective I/O be performed. In addition, the high level hint (Block, Block) will be converted by MDMS to a data structure

[1] In [7], the user has already specified these information to MDMS when she starts a new application. So our performance predictor can just search database to obtain these information.

understandable by lower level I/O functions to SRB-OL or MPI-IO. Finally, SRB-OL or MPI-IO uses this data structure and I/O decision to perform optimized I/O. We can see that this user level hints (from user application), system level optimization decision maker (by MDMS) and low level I/O optimization library (SRB-OL) provide a good example of layered design and implementation in software engineering. This allows design of low level library (such as design and implementation of our SRB-OL) independent of other layers and make the higher level layers (such as MDMS) portable.

7 Conclusion and Future Work

In this paper, we presented our *SRB Optimization Library* (SRB-OL) which is built on top of SDSC's Storage Resource Broker (SRB) to optimize I/O access to various archival storage systems such as HPSS, UniTree etc. Like UNIX I/O, SRB I/O only reads/writes a contiguous piece of data, but many user access patterns are non-contiguous, this results in large number of I/O calls which are very expensive. Our SRB-OL employs numerous state-of-the-art I/O optimization approaches that address this problem. We designed and implemented remote collective I/O, data sieving, superfile on top of SRB and we also incorporated subfile, container, migration, stage and purge into our library. These optimization schemes provide our Meta-data Management Systems (MDMS) with ample choices of I/O optimizations.

We proposed a performance model based on our experiments for SRB I/O accessing HPSS. By using this performance model, we can prove quantitatively that collective I/O has significantly better performance than does the naive I/O approach which includes numerous I/O calls when the data partition does not conform the layout on storage.

We also presented the design of an *I/O Performance Predictor* that can give the user a concept how much I/O cost for her application, so she can make best use of her application by choosing appropriate parameters.

Our performance model and all our experiments have not considered the tape stage time T_{stage} yet. This is because T_{stage} may vary significantly and our current privilege is not allowed to perform *purge*. Therefore, we can not guarantee that there is no disk copy on HPSS when we run the experiment[2]. In the future, we would find a way to solve this problem and re-do all the experiments. We believe we can obtain much more performance improvements for most of experiments. For example, in superfile optimization, when T_{stage} is taken onto account, the naive approach would have 10 stages for 10 small files, while in superfile, only one stage occurs. As stage is very expensive, superfile would have much more performance improvements.

Acknowledgments

This research was supported by Department of Energy under the Accelerated Strategic Computing Initiative (ASCI) Academic Strategic Alliance Program (ASAP) Level 2, under subcontract No W-7405-ENG-48 from Lawrence Livermore National Laboratories. We would like to thank Regan Moore and Mike Wan of SDSC for helping us with the usage of SRB. We thank Mike Gleicher and Tom Sherwin of SDSC for answering our HPSS questions.

References

[1] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The sdsc storage resource broker. In *Proc. CASCON'98 Conference, Dec 1998, Toronto, Canada*, 1998.

[2] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. Passion: parallel and scalable software for input-output. In *NPAC Technical Report SCCS-636*, 1994.

[2] According to SDSC, the lifetime of disk cache copy could be as long as 10 days, this makes us difficult to judge when the disk copy is purged.

[3] R. A. Coyne, H. Hulen, and R. Watson. The high performance storage system. In *Proc. Supercomputing 93, Portland, OR*, 1993.

[4] Hpss worldwide web site. In *http://www.sdsc.edu/hpss/*.

[5] D. Kotz. Disk-directed i/o for mimd multiprocessors. In *Proc. the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, 1994.

[6] G. Memik, M. Kandemir, A. Choudhary, and V. E. Taylor. April: A run-time library for tape resident data. In *the 17th IEEE Symposium on Mass Storage Systems*, 2000.

[7] X. Shen, W. Liao, A. Choudhary, G. Memik, M. Kandemir, S. More, G. Thiruvathukal, and A. Singh. A novel application development environment for large-scale scientific computations. In *International Conference on Supercomputing, May 8-11, 2000, Santa Fe, New Mexico*, 2000.

[8] Srb version 1.1.4 manual. In *http://www.npaci.edu/DICE/SRB/OldReleases/SRB1-1-4/SRB1-1-4.htm*.

[9] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective i/o in romio. In *Proc. the 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[10] *UniTree User Guide, Release 2.0*. UniTree Software, Inc., 1998.