Load Balancing using Grid-based Peer-to-Peer Parallel I/O

Yijian Wang and David Kaeli
Department of Electrical and Computer Engineering
Northeastern University, Boston, MA 02115
yiwang,kaeli@ece.neu.edu

Abstract

In the area of Grid computing, there is a growing need to process large amounts of data. To support this trend, we need to develop efficient parallel storage systems that can provide for high performance for data-intensive applications. In order to overcome I/O bottlenecks and to increase I/O parallelism, data streams need to be parallelized at both the application level and the storage device level.

In this paper, we propose a novel Peer-to-Peer(P2P) storage architecture for MPI applications on Grid systems. We first present an analytic model of our P2P storage architecture. Next, we describe a profile-guided data allocation algorithm that can increase the degree of I/O parallelism present in the system, as well as to balance I/O in a heterogeneous system. We present results on an actual implementation. Our experimental results show that by partitioning data across all available storage devices and carefully tuning I/O workloads in the Grid system, our Peer-to-Peer scheme can deliver scalable high performance I/O that can address I/O-intensive workloads.

1 Introduction

In many high performance application domains there is a growing need to process very large datasets. Future high-energy physics applications are projected to generate tera-byte data every day, or a peta-byte of data a year [4]. Processing this amount data poses new challenges to the parallel computing community. Data-intensive applications demand large storage capacities and high I/O throughput. Grid computing is an emerging technology to connect geographically distributed computing resources (e.g., clusters, supercomputers) in order to provide high performance systems for users across the nation and the world. The development of the Globus [3] toolkit and networking infrastructure have made it possible for users to run their applications at a remote site, or to run their applications at multiple sites concurrently. However, for applications that access (i.e., read and write) large datasets, there lacks an effective storage mechanism to deliver high I/O bandwidth.

In order to achieve high I/O throughput on Grids, data streams need to be parallelized at both the application level (e.g., using I/O middleware) and storage level (e.g., applying data partitioning and allocation). I/O contention at any system level can cause severe performance bottlenecks.

In our past work [20], we studied how to provide scalable I/O performance using profile-guided partitioning on a single cluster. We found that partitioning files on local disks could provide significant application speedup in file-based parallel applications. In this work, we focus on improving parallel I/O performance on a Grid-level system.

1.1 Cluster I/O

Clusters are typically built from commodity off-the-shelf microprocessors, memories, disks, and networks. A large number of nodes are interconnected to deliver a high-performance, low-cost, computing platform. I/O storage is typically centralized on one or a few nodes in the cluster. I/O subsystems typically found on a cluster-based system can be complex:

1. *shared storage*, where program source codes, executables, scripts, etc. reside. These files and data should be accessed and available to all nodes within the cluster. They are usually stored on a centralized storage device that is managed by one or a few nodes. A parallel file system (i.e. NFS, PVFS, etc.) is used to provide concurrent access by multiple nodes. The centralized disk device on a cluster is often a high capacity RAID system that can support high throughput;

2. *local disk*. Each node in the system can run its own copy of operating system and uses local disk space to support virtual memory and to create temporary files. As a result, on a typical cluster, there exists a disk connected directly to each node.

Many proposed parallel systems the use MPICH-G2 employ a client server model [5, 19, 7], where MPI applications are executed at a remote site (e.g., a supercomputer or a cluster) and data is read from/written to a centralized file server. In this paper, we discuss the possibility of executing an MPI application at multiple sites concurrently by utilizing a Peer-to-Peer storage model.

1.2 Peer-to-Peer Model

The term *Peer-to-Peer* refers to a class of systems and applications that employ distributed resources (i.e., computing power, data and network bandwidth) to perform a critical function in a decentralized manner [1]. The critical functions can be distributed and include computing, data sharing, and communication services. Peer-to-Peer architectures have begun to receive more attention due to their ability to avoid dependencies on centralized points of control.

A Peer-to-Peer (P2P) storage network does not have the concept of *clients* and *servers*, but only equal *peer* nodes that function as both *clients* and *servers* with other nodes in the system. A P2P model differs from a client-server model where file requests are usually sent to and from a centralized server. By utilizing a P2P model, I/O traffic can be offloaded from the centralized file server and the backbone network.

1.3 Workload balancing

Today's parallel systems usually have heterogeneous storage devices, because:

- The Grid systems are naturally heterogeneous by connecting multiple distributed clusters that are managed by different groups.
- As we have discussed earlier, clusters typically have heterogeneous storage devices.
- The file system aging problem introduces heterogeneity to the system. Identical disks may have different bandwidth capabilities due to fragmentation, bad sectors, and different low level formats.

In a heterogeneous storage system, parallel I/O performance can drop dramatically due to workload imbalance. Load balancing is a commonly used solution to improve performance on heterogeneous parallel systems. Load balancing attempts to tune I/O on the heterogeneous disk devices to fully utilize the available disk bandwidth.

The rest of this paper is organized as follows. Section 2 will discuss some related work. Section 3 will present our profile-guided file partitioning and load balancing approach. Section 4 will provide our experimental environment and results that illustrate the power of our algorithm. In section 5, we will summarize this paper and talk about some future directions.

2 Related Work

MPI-IO [11], which is included as part of the MPI-2 [12] standard, has been widely used to provide parallel access to shared data files. Techniques such as data sieving and collective I/O [16] have been successful in improving performance. Many parallel applications produce accesses to small, potentially non-contiguous, data chunks. Collective I/O merges multiple I/O accesses generated by multiple processes into a single I/O function call. MPI-IO provides for collective communications so that multiple MPI processes can access a single data file simultaneously.

There have been a number of projects that have developed scheduling techniques on Grids. In [2], Thain et al. describe a system that allows jobs and data to meet by binding execution and storage sites together into I/O communities which then participate in the wide-area system. They demonstrate their implementation of I/O communities by improving the performance of a key high-energy physics simulation on an international distributed system. In [6], Abawajy addresses the problem of effective management of parallel I/O in cluster computing systems by using two new I/O scheduling algorithms. In [9], Ranganathan and Foster describe a scheduling framework that addresses large scale, data-intensive, problems. Within the framework, data movement operations may be either tightly bound to job scheduling decisions, or alternatively, performed by a decoupled, asynchronous process on the basis of observed data access patterns and system load. In [15], Oldfield and Kotz describe the *Armada* framework for building I/O access paths for data-intensive grid applications. Armada allows grid applications to efficiently access data sets distributed across a computational grid. *Armada* allows the application programmer and the dataset provider to design and deploy a flexible network of application-specific and dataset-specific functionality across the Grid.

In order to integrate the standard MPI-IO interfaces with Grid systems, a number of technologies have been

proposed to provide the capabilities of remote parallel file access for Grid MPICH-G2 applications. In [5], Foster et al. describe a remote I/O library called RIO to address issues of portability of adopting MPI-IO interfaces. The paper defines a RIO device and a RIO server within the ADIO abstract I/O device architecture. By providing traditional I/O optimizations such as enabling asynchronous operations, and through implementation techniques such as buffering and message forwarding to offload communication overhead, RIO can improve turnaround time relative to staging [5]. In [19], Baer and Wyckoff present an implementation of the MPI-IO interface using the Globus GridFTP client API to access shared remote data sets. The authors compare the performance of GridFTP to that of NFS on the same network using several parallel benchmarks. Their results show that GridFTP can be a workable transport for parallel I/O, particularly for distributed read-only access to shared datasets. In [7], a remote file system (RFS) is presented to improve remote I/O performance by using active buffering with threads (ABT), which hides I/O costs by aggressively buffering the output data using threads. The SDSC Storage Resource Broker (SRB) [18] is a client-server middleware that provides a uniform interface for connecting to heterogeneous data resources over a network and accessing replicated data sets.

However, when managing I/O in a client-server model where the data is written to/read from files that are located on a remote site, parallel access to remote datasets can be extremely expensive and will hurt the overall performance of the user applications. To overcome this performance barrier and to reduce inter-cluster communications, we propose a novel storage architecture that partitions data across local storage devices to provide a Peer-to-Peer storage system for MPI-IO applications.

3 Peer-to-Peer Storage System Modeling

Our goal is to find an improved data layout across the storage devices within a Grid system to improve I/O parallelism and to balance I/O workload as well. MPI-IO applications usually access files in a large number of data chunks. We want to find a tuned assignment of M data chunks that can be mapped to N disks/partitions with the objective of limiting network contention.

Let $X_{i,j}$ be the output data layout matrix, where $X_{i,j} = 0$ denotes the ith data chunk is NOT assigned to the jth partition, and $X_{i,j} = 1$ represents the ith data chunk IS assigned to the jth partition. S_i is the size of the ith data chunk; B_j is the disk bandwidth of the jth partition/disk; BN is the network bandwidth; $A_{i,j}$ is the number of times processor j accesses data chunk i (this value can be extracted from profiles); $C1_{i,j}$ is the estimated time due to disk I/O and network I/O when data chunk i is assigned to partition j (for simplicity, we model the network as a queuing system):

$C1_{i,j} = \frac{S_i}{B_j} + \frac{S_i}{BN} * \sum_{k=0,k\neq j}^{N-1} A_{i,k}$

$C2_{i,j}$ is the total network I/O time for processor j to access data chunk i, which is not assigned to partition j:

$C2_{i,j} = \frac{S_i}{BN} * A_{i,j}$

The aggregated parallel I/O bandwidth is computed by:

$$Max_{j=0,\ldots,N} \sum_{i=0}^{M-1} (X_{i,j} * C1_{i,j} + (1 - X_{i,j}) * C2_{i,j})$$

We can cast this problem into an Integer Linear Programming problem:

To minimize

$$Max_{j=0,\ldots,N} \sum_{i=0}^{M-1} (X_{i,j} * C1_{i,j} + (1 - X_{i,j}) * C2_{i,j})$$

,

where $C1_{i,j} = \frac{S_i}{B_j} + \frac{S_i}{BN} * \sum_{k=0,k\neq j}^{N-1} A_{i,k}$,

$C2_{i,j} = \frac{S_i}{BN} * A_{i,j}$,

$0 \leq i \leq M-1, 0 \leq j \leq N$,

$X_{i,j} \epsilon 0, 1, \sum_{j=0}^{N} X_{i,j} = 1$

Data partitioning is a well known NP-hard problem. In the next section, we will present greedy heuristics to produce an efficient solution to this problem.

4 Profile-Guided Peer-to-Peer Partitioning

In this section, we describe our profile-guided methodology and our partitioning algorithm for a P2P model.

4.1 Access Patterns Detection

To guide our partitioning algorithm, we need to collect the following profile information:

- process ID (*PID*),
- file handler (*fh*),
- address of each data chunk accessed (*Adr*),
- chunk size (*size*),
- access type (*read/write*), and
- timestamp.

We have developed a library function to capture this information for every file operation. The library function is called on every file I/O and produces a profile record. We then use this profile to characterize the file access patterns and guide I/O workload partitioning across local disks and centralized disk devices.

4.2 P2P Data Partitioning

Optimal I/O partitioning is an NP-hard problem. We have developed a greedy algorithm to partition the I/O workload that produces an improved data layout within the Grid storage system. For every contiguous data chunk accessed, we identify which process accesses each chunk most frequently. We can then simply assign it to the local disk of the associated node. For data sets that are heavily referenced by multiple processes, we use a hybrid storage strategy, described as follows: If the chunk is read-only by multiple processes, we can replicate the chunk in multiple file partitions. If the chunk is written by multiple partitions, we can create a shared file partition. In the case where a chunk is first written by a single process and later read by multiple processes, we let the writing process broadcast the updated data to those processes that are going to read it later. After we complete the assignment of chunks to partitions, we then reorder data chunks in a partition based on the earliest timestamp recorded for each chunk(chunks may be accessed many times, we only consider the time of the first access though). Figure 1 provides pseudo-code of our greedy partitioning algorithm. The complexity of this algorithm is $O(M^2 * N)$, where M is the number of chunks and N is the number of processes.

4.3 Load Balancing

One assumption we made in the previous section is that the system provides a homogeneous storage environment. Assuming a homogeneous cluster, data can be partitioned based on a profile of the access frequencies of each process. In this section, we focus on load balancing on a heterogeneous P2P system by taking into account differences in disk bandwidth.

4.3.1 Coarse Grained Load Balancing

Assume there are N disks $D_i, 0 \leq i \leq N-1$. and there are M data chunks $C_j, 0 \leq j \leq M-1$. For every contiguous data chunk accessed, we identify which process accesses each chunk most frequently. We then generate matrix T, where $T_{i,j}$ denotes the cost of assigning data chunk C_j to disk D_i. $T_{i,j}$ is calculated as:

$$T_{i,j} = \begin{cases} \frac{S_i}{B_j} & \text{if } S_i \text{ is local-to-Peer } P_j \\ \frac{S_i}{B_j} + \frac{S_i}{NB} & \text{if } S_i \text{ is not local-to-Peer } P_j \end{cases}$$

B_i is the disk bandwidth and NB is the network bandwidth. For instance, if data chunk C_1 is only accessed by process P_1 and the chunk is located on disk D_1, then $T_{1,1}$ will be S_1/B_1, and likewise, for any $T_{1,j}, j \neq 1$ will be $S_1/B_1 + S_1/NB$. Our algorithm will start from one data chunk C_i and assign it to the disk with the lowest cost, and then update the matrix T by adding the lowest cost to other unassigned data chunks in the same column (this means that if another chunk is assigned to the same disk later, it needs to wait for the service time of the previous chunk). In Figure 2, we show a simple example to illustrate our algorithm. There are four peer systems and each system hosts a disk with different bandwidth (e.g., X/8, X/2, X, X). The four peers access four data chunks of the same extent (i.e., size) simultaneously. The network bandwidth is X/2 (this number is not small if you consider inter-cluster communication bandwidth). Based on this access pattern, network latency and disk bandwidth, the cost matrix is initialized as shown in Figure 2(a). In a homogeneous storage system, we can easily partition the data chunks and assign them to local disks based on the spatial access pattern. However, for a heterogeneous disk system, this will result in a workload imbalance and waste the available bandwidth of the faster disks. As we can see in this example, if our homogeneous partitioning algorithm is applied, the overall execution time will be 8, which is determined by the slowest disk in the system. Our load balancing algorithm starts from any data chunk (chunk C_0, for simplicity), and assigns it to the disk with lowest cost, which is D_2 in this example. The cost matrix is then updated and the same algorithm continues until all data chunks are assigned. The overall execution time is 4, which is half of the time for the homogeneous case.

4.3.2 Fine-Grained Load Balancing

In Figure 2, we can see that Disk D_0 is not assigned any data based on our coarse-grained balancing algorithm. In order to achieve a more balanced workload, data chunks on heavily-loaded nodes can be split and migrated to peers that are less heavily loaded. As an example, if disk D_i(bandwidth B_i) is assigned a workload with the size of S_i, and D_j(bandwidth B_j) is assigned a workload with the size of S_j (assume $\frac{S_j}{B_j} > \frac{S_i}{B_i}$), then we can split the chunk S_j and assign a portion of it to D_i in order to balance workload. In other words, our goal is to:

```
For each I/O process
    Create a Partition;
For each contiguous data chunk
    Total up the number of read accesses on a Process-ID basis;
    Total up the number of write accesses on a Process-ID basis;
    If the chunk is accessed by only one Process ID
       Assign the chunk to the associated partition;
    If the chunk is written by one process and later read by multiple processes
       Assign the chunk to all partitions where read
       and broadcast the update to all partitions when writes;
    Else Assign the chunk to a shared partition;
For each Partition
    Sort chunks based on the earliest timestamp for each chunk;
```

Figure 1. Pseudocode for our partitioning heuristics.

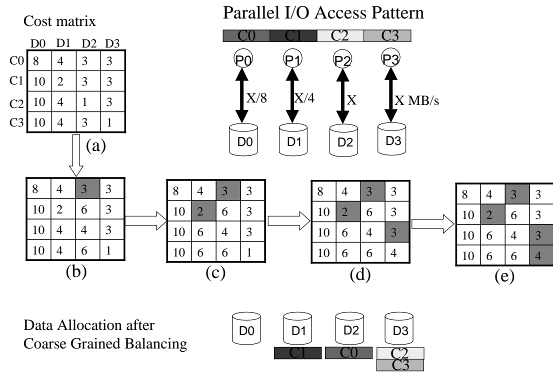

Figure 2. An example of our load balancing algorithm for parallel data accesses.

$$minimize|\frac{s_j - s}{B_j} - (\frac{S_i + s}{B_i} + \frac{s}{NB})| \qquad (1)$$

where s is the size of dataset that we want to move from D_j to D_i, and NB is the network bandwidth.

We can derive s from equation 1:

$$s \approx \frac{B_i S_j - B_j S_i}{B_i + B_j + \frac{B_i B_j}{NB}} \qquad (2)$$

5 Experiments

Next, we present our experimental environment and results on four parallel I/O workloads.

5.1 Experimental Environment

We work on a Globus-enabled Grid system. This system includes two clusters, *Joulian* [8] and *Keys* [10]. The joulian cluster has 32 nodes (only 16 are used here-in). Each node has a Pentium 2 processor, 256MB memory and are connected with a 100 Mbit/s Fast Ethernet switch (the head node has two Pentium 2 processors). Each node also has a local 8.4GB IDE disk and there is a shared SCSI RAID device directly attached to the head node.

The Keys cluster has 9 nodes, with Pentium 4 processors, 1 GB memory and 1Gbit/s Ethernet switch. Each node has a 40GB IDE disk. The disk drive hosted by the head node is enabled as a shared device by the NFS system. All I/O is directed to this shared disk. More details of these two clusters can be found in Table 1 and 2.

Figure 3 shows a picture of our Grid system where we implemented our system. The storage devices in dark color represent the shared spaces enabled by NFS on each cluster.

Table 3 provides the raw latency and bandwidth rates for intra-joulian, intra-keys, inter-head nodes and inter-subnodes. Inter-subnodes communication assumes that communications across the local Fast Ethernet switch on joulian, inter-cluster communications over the Internet and the local Gbps Ethernet on Keys. We have also measured the sustained disk bandwidth for the three various devices across the system (write/read in MB/s): Joulian-IDE (3.5/3.8), Joulian-RAID (13.0/17.7) and Keys IDE (6.4/9.8).

5.2 Parallel I/O Workloads

Our target applications are parallel scientific applications that require intensive disk accesses. We have studied four MPI-IO workloads:

Number of nodes	32 - (1 head nodes, 31 subnodes (including 1 SMP node))
Processor Type	Intel Pentium II 350 (2 processors on head node) Intel Pentium II Xeon 450 (SMP nodes)
Memory	256MB SDRAM, PC100, ECC, (standard nodes and RAID nodes) 2GB (SMP node)
Disk adapters IDE SCSI	 Onboard Intel PCI (PIIX4) dual ultra DMA/33 UltraWide SCSI
RAID device	Morstor TF200 with 6-9GB Seagate SCSI disks, 7200rpm, QLogic 64-bit PCI-Fibre Channel Adapter
RAID level	5
RAID capacity	36GB usable, one hot spare
IDE disk	IBM UltraATA, 8.4GB, 5400rpm
File system	NFS
Network	100Mbits/s Fast Ethernet

Table 1. Hardware specifics of the Joulian Cluster used in this work.

	Intra-Joulian	Intra-Keys	Inter-Headnodes	Inter-Subnodes
Ave. latency	0.3ms	0.1ms	2.5ms	3.6ms
Ave. bandwidth	10.28MB/s	37.0MB/s	0.7MB/s	0.6MB/s

Table 3. Raw latency and bandwidth rates for our grid system across two clusters.

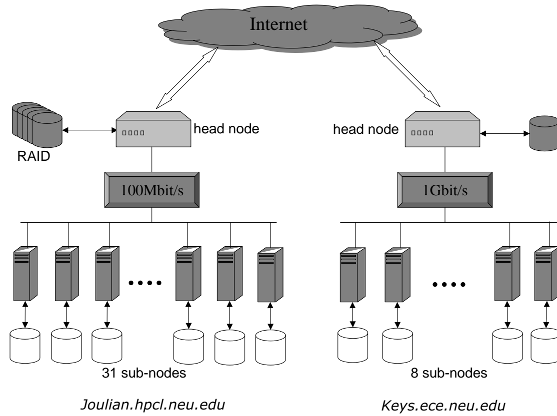

Figure 3. A Grid System used in this work

Number of nodes	9
Processor Type	Intel Pentium 4 1G
Memory	1GB SDRAM
Shared disk	IBM Deskstar, 40GB, 7200rpm
IDE disk	IBM Deskstar, 40GB, 7200rpm
File system	NFS
Network	1Gbits/s Ethernet

Table 2. Hardware specifics of the Keys Cluster used in this work.

- The NPB2.4/BT benchmark [14] is part of the NAS Parallel Benchmark (NPB) suite version 2.4. The suite consists of 8 programs designed to evaluate the performance of parallel supercomputers. The code is written in Fortran. The benchmarks are taken from computational fluid dynamics problems. The application that we are using is the Block-Tridiagonal (BT), that is file bound. The application is provided with different input problem sizes (A-D); we are using size B, that dynamically generates a data set (1.5 gigabytes) and then reads it back. Each process periodically writes sequentially, and this chunk is later read. Chunk sizes are a function of the number of processes. This parallel application needs to run on a number of processes that is a square (i.e., 4, 9, 16, 25).

- The SPECseis96.1.2 benchmark [17] is one of three applications in the SPEChpc96 benchmark suite. The code is written in both Fortran and C. The application is performing seismic data processing. The code consists of four phases. We only study the first two phases of this benchmark. During phase 1, the program dynamically generates a data set (1.6 gigabytes) and during phase 2 it reads the data set back. Each process writes 96KB chunks, and each process then reads back 2KB chunks.

- The Perf benchmark is a parallel I/O test program provided with the MPICH standard distribution [13]. The code is written in C. Every process writes a 1 MB chunk at a location determined by

its rank, and then reads it back later. We study the synchronized disk performance of this benchmark.

- Mandelbrot is an image processing application that generates a Mandelbrot image file. In this benchmark, a Mandelbrot image data file (256MB) is generated by multiple processes and then read back for visualization. The code is written in C. The code is computationally intensive, I/O intensive, and visualization intensive. The size of each contiguous file access depends on the number of processes.

We utilize the MPICH-G2 1.2.5 (Globus 2.2.4) for MPI computation. If the number of MPI processes is an even number, we have each site host half processes; if the number of processes is a square (i.e the BT benchmark), then we simply assume that Keys has $\lceil n/2 \rceil$ and Joulian has $\lfloor n/2 \rfloor$ processes.

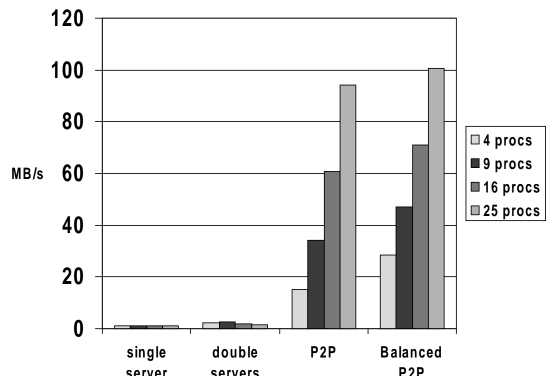

Figure 4. BT read performance in MBs/second.

5.3 Experimental Results

We implemented four parallel I/O models: (a) client-server model with one NFS server (single server); (b) client-server model with two NFS servers (double servers); (c) P2P model before balancing; (d) P2P model after fine grained balancing.

In (a), all data is stored on a shared storage server (i.e., on the RAID device on joulian). In the source code, data are explicitly staged from the file server node to compute nodes. In (b), we use the simplified partitioning algorithm presented in Section 4 that partitions data sets across the two shared spaces on the two clusters; In (c), we partition the data across all the nodes by applying our partitioning algorithm in; and finally we apply and evaluate our load balancing algorithm in (d).

Figures 4 and 5 present read/write bandwidth data for BT benchmark. We measure the end-to-end latency of each access. Client-server file operations are generally slow and P2P achieves significant speedup in comparison. The double server model has produced a significant speedup compared to the single server model. However, both client-server models cannot scale due to bottlenecks in parallel file system. This fact is evident in all of our results. If we compare P2P I/O (before balancing) with single server configuration of client-server I/O, we achieve a speedup of 105.7X on reads and 94.9X on writes. We also achieve significant speedup with our load balancing algorithm: after fine-grained balancing, the bandwidth of P2P model has been improved by up to 81% and 74% for reads and writes respectively.

Figure 6 and 7 show the I/O bandwidth measured with the SPEChpc/seis benchmark. Again, we measure the performance of single server I/O, double servers I/O, and P2P I/O. Because of the poor data locality of this application (each peer needs to read from all other peers' disk and this leads to all-to-all intercluster communications), we did not get any noticeable speedup with our partitioning algorithm. However, we are able to get a significant improvement on writes. Compared with single server I/O, the write performance of P2P I/O (before balancing) is improved by a factor up to 56.1X, while the read performance remains roughly the same.

Figure 8 and 9 report the read/write bandwidth of the Perf benchmark. The speedup of unbalanced P2P I/O over single server I/O is up to 60.9X and 56.1X for reads and writes, respectively. Again, our load balancing scheme is able to balance the P2P model with a factor up to 78% and 84% for reads and writes respectively.

Figures 10 and 11 present I/O performance of the Mandelbrot application. Again, we got similar results with a speedup factor up to 89.2X for reads and 87.1X for writes using unbalanced P2P architecture. With our load balancing algorithm, we can get even better parallel I/O performance: compared with unbalanced workloads, balanced P2P model can achieve a speedup factor of up to 31% and 61% for reads and writes respectively.

For the above experiments, the P2P model can significantly offload the centralized server and the backbone network compared with the traditional centralized model. We have also shown that our load balancing algorithm can help balance I/O loads in the heterogeneous storage environment and generates even better performance.

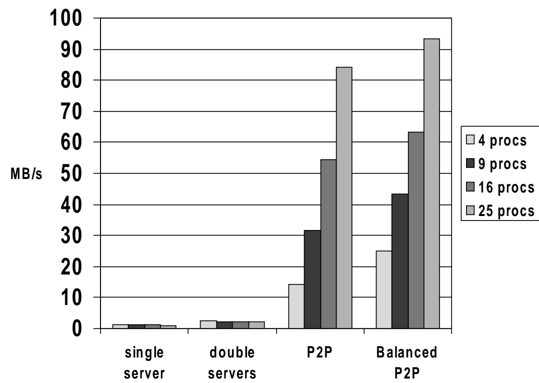

Figure 5. BT write performance in MBs/second.

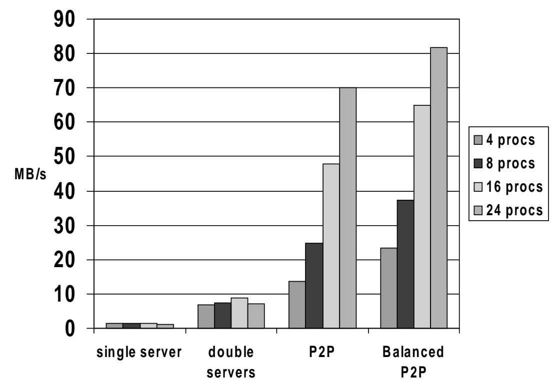

Figure 7. SPEChpc/seis write performance in MBs/second.

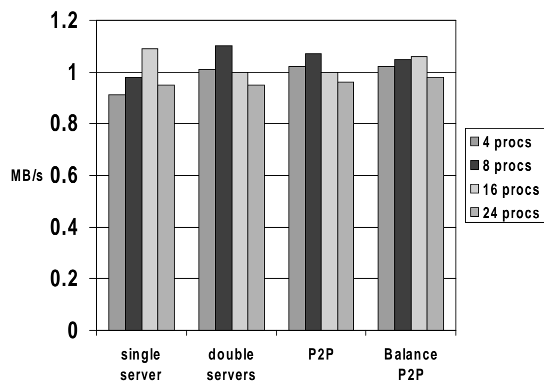

Figure 6. SPEChpc/seis read performance in MBs/second.

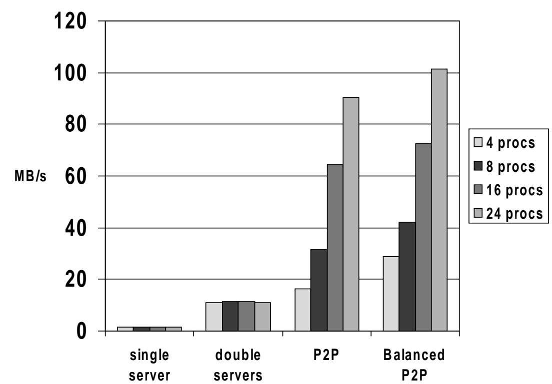

Figure 8. Perf read performance in MBs/second.

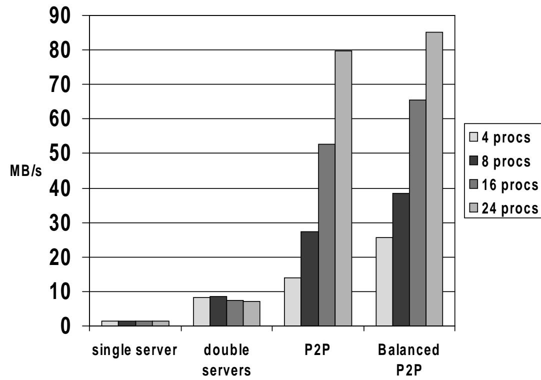

Figure 9. Perf write performance in MBs/second.

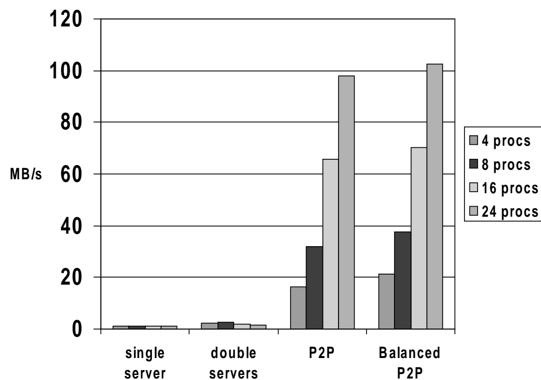

Figure 10. Mandelbrot read performance in MBs/second.

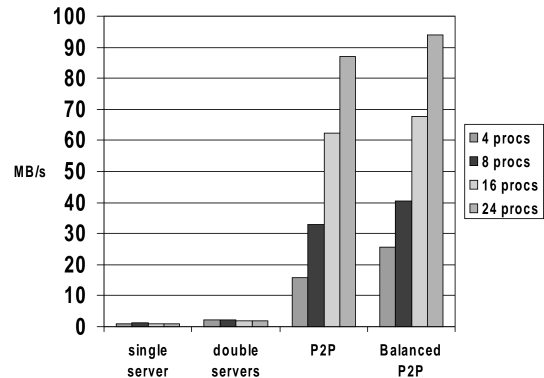

Figure 11. Mandelbrot write performance in MBs/second

5.4 Discussion

In this paper, we have presented a profile-guided Peer-to-Peer storage approach to achieve scalable speedup. We have been able to create multiple I/O channels, reduce disk seek times and decrease I/O latency. Our results demonstrate that our scheme yields significant performance improvements for the parallel applications we have studied.

For our profile-guided approach to be adopted, this technique needs to be insensitive with the changes of input data set. We have evaluated the sensitivity to various changes in terms of the stored data values, the number of processes and the data sizes. We have found that changes in data values have very little effect on the profiling data, and so there is no need to re-profile an application when the input data set values are changed. When we change the number of processes, the chunk sizes will change predictably. For instance, in the BT benchmark, we have made the following observations:

1. file access patterns remain the same when data values change; and

2. data chunk sizes vary with the number of processes.

Through further analysis, we found that we could detect two different trends. When increasing the size of the input datasets, either:

1. the number of I/O increases, though the pattern of I/O's remained the same, and the chunk size remains the same (e.g., these patterns were observed in SPEChpc/seis), or

2. the size of each chunk size increases, while the number of I/O's remains the same (e.g., BT, Perf and Mandelbrot).

In summary, for the three applications we have studied, we find that the I/O access patterns of parallel scientific applications are highly predictable and insensitive to input datasets. Based on this observation, we can successfully improve the I/O performance by applying our profile-guided approach.

6 Conclusion

To achieve scalable I/O performance of parallel applications on Grids, it is important to parallelize I/O streams at both the application level and storage level. In this paper, we present a Peer-to-Peer architecture to parallelize I/O accesses guided by profiles. We then use the profiles to guide how to partition data set across disks to achieve high I/O throughput and to balance I/O workloads as well. For the applications we have studied, P2P I/O delivers significant high performance I/O compared with traditional client-server models (for both read-intensive and write-intensive applications), while also scaling throughput as the size of our system increases. We are currently working on our data partitioning tool and we will make it publically available shortly.

For the future work, we are going to implement a P2P parallel I/O middleware for MPI applications on Grid systems. Since the parallel I/O access patterns for this category of applications are highly predictable, we are also looking at the possibility of building a dynamic mechanism that can bypass the profiling phase and partition and balance data files during runtime.

7 Acknowledgements

This work was supported by CenSSIS, the Center for Subsurface Sensing and Imaging Systems, under the Engineering Research Centers Program of the NSF (Award Number EEC-9986821), by the Institute of Complex Scientific Software at Northeastern University, and by the NSF Major Research Instrumentation Program (Award Number MRI-9871022). We would also like to thank Juemin Zhang for his effort in setting up the grid environment used in this work.

References

[1] D. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins and Z. Xu. Peer-to-Peer Computing. Technical report, HP Laboratories, 2002.

[2] D. Thain, J. Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau and M. Livny. Gathering at the Well: Creating Communities for Grid I/O. In *Prceedings of the SuperComputing Conference*, 2001.

[3] The Globus Toolkit. http://www.globus.org.

[4] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[5] I. Foster, D. Koha and R. Krishnaiyer. Remote I/O: Fast Access to Distant Storage. In *Proceedings of the Workshop on I/O in Parallel and Distributed Systems*, 1997.

[6] J. H. Abawajy. Performance Analysis of Parallel I/O Scheduling Approaches on Cluster Computing Systems. In *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*, 2003.

[7] J. Lee, X. Ma, R. Ross, R. Thakur and M. Winslett. RFS: Efficient and Flexible Remote File Access for MPI-IO. In *Proceedings of the International Conference on Cluster Computing*, 2004.

[8] The Joulian Cluster at Northeastern University. http://joulian.hpcl.neu.edu.

[9] K. Ranganathan and I. Foster. Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications. In *Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing*, 2002.

[10] The Keys Cluster at Northeastern University. http://keys.ece.neu.edu.

[11] ROMIO: A High-Performance, Portable MPI-IO Implementation. http://www-unix.mcs.anl.gov/romio.

[12] Message Passing Interface Forum. http://www.mpi-forum.org/.

[13] MPICH - A Portable Implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich.

[14] NAS Parallel Benchmark Suite. http://www.nas.nasa.gov/Software/NPB.

[15] R. Oldfield and D. Kotz. The Armada Framework For Parallel I/O on Computational Grids. In *Proceedings of the 1st USENIX File and Storage Technologies*, 2002.

[16] R. Thakur, W. Gropp and E. Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the 7th Symposium on Frontiers of Massively Parallel Computation*, 1999.

[17] SPEChpc Benchamrk Suite. http://www.specbench.org/hpg/.

[18] The SDSC Storage Resource Broker. http://www.sdsc.edu/srb/.

[19] T. Baer and P. Wyckoff. A Parallel I/O Mechanisim for Distributed Systems. In *Proceedings of the International Conference on Cluster Computing*, 2004.

[20] Y. Wang and D. Kaeli. Profile-guided I/O Partitioning. In *Proceedings of the 17th ACM International Conference on Supercomputing*, 2003.