Profile-Guided File Partitioning on Beowulf Clusters

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

Abstract

On cluster-based systems, data is typically stored on a centralized resource, and each node has a local disk used for the operating system and swap space. Although I/O middlewares (e.g., MPI-IO) and high performance I/O subsystems (e.g., RAID) can generate parallel I/O streams, disk contention and network latency still dominate I/O performance. To address this performance barrier, I/O access needs to be partitioned to use local disks in order to distribute I/O workload and reduce network contention.

In this paper, we present a profile-guided I/O partitioning scheme that utilizes both local and centralized I/O storage within a cluster to improve I/O parallelism and scalability. We target our work at scientific applications that are dominated by file I/O. We first characterize file access patterns by studying I/O profiles. We propose and evaluate a new file partitioning algorithm to optimize data layout in cluster I/O subsystems. Experimental results show that application execution time can be reduced by 27.8%-87.0% using our profile-guided approach.
keywords: I/O partitioning, profiling and MPI-IO.

1 Introduction

On Beowulf cluster systems, nodes 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 to one or a few nodes in the cluster.

Many research studies have focused on the performance of different components in clusters [5, 16, 25]. Little has been done to exploit the availability of local disks since most clusters utilize a centralized, typically networked, storage device. While clusters can be effectively exploited to obtain massive speedups in compute-bound workload, they are much less effective in applications that include a large amount of file I/O. To reduce I/O bottlenecks, as well as to achieve scalable throughput, I/O should be carefully partitioned across centralized and local disk devices.

To effectively tune parallel workload throughput, it is important to characterize I/O access patterns. Studies of parallel I/O workloads have shown that file access patterns often consist of accesses to a large number of small, noncontiguous data chunks [19]. I/O subsystems are designed to exploit large chunk sizes and contiguous data chunks. Irregular access patterns create bottlenecks in I/O intensive applications.

I/O middlewares (e.g., MPI-IO[11]) support runtime parallelization of access to shared files. Parallel disk arrays can also be used accelerate I/O performance by striping data across multiple disks[14]. Some parallelizing compilers try to address poor I/O performance by improving data locality, while also reducing the number of disk accesses. Also, a number of parallel filesystems

allow for multiple processes to access shared data efficiently [1, 3, 17]. In order to effectively exploit I/O parallelism, I/O streams need to be parallelized at multiple layers (i.e., process level, file level, disk level). Contention at any layer can dramatically impact performance.

1.1 I/O Middlewares

MPI-IO is a middleware standard used for describing parallel I/O operations within an MPI-2 parallel application [13]. ROMIO, an implementation of MPI-IO developed at Argonne National Laboratory [22], uses two key optimizations:

1. *collective I/O* [21], and
2. *data sieving* [20].

Collective I/O merges access requests from multiple MPI processes and performs disk accesses *collectively*. Instead of accessing multiple noncontiguous data pieces individually, data sieving accesses a single contiguous data chunk, including the requested datasets, as well as unneeded data holes. Data sieving works very well as long as the holes remain small.

While MPI-IO realizes parallel I/O at the application level, it is still possible that multiple processes will attempt to access datasets that reside on the same disk device. Since disks can only process a single access at a time, I/O operations will be serialized at the disk level. While I/O operations have been parallelized, they will be performed serially and an application will not reap the potential performance benefits from using MPI-IO. Since working at an MPI-IO layer of abstraction prevents us from knowing more about the underlying disk devices, we cannot guarantee that I/O will be serviced in parallel.

To address disk contention problems, Kotz designed a *disk-directed collective* scheme [7]. The proposed technique reorganizes read and write requests to match the physical layout of data on disk. Seamons et al. proposed *server-based collective I/O* [18], that can transform collective operations into a series of requests for whole file blocks at each I/O node.

To further improve the performance of parallel I/O streams, accesses should also be parallelized at the underlying disk level. Disk arrays have proven to be very effectively in increasing I/O performance, using a technique call *striping*. Disk arrays also provide for increased reliability by adding parity or ECC. RAID-x, as described by Hwang et al. [5], utilizes *orthogonal striping* across multiple RAID groups. While a RAID device can improve access to a single chunk of data, bottlenecks occur in I/O intensive parallel applications when multiple processes access the same disk array. This will cause I/O to serialize.

1.2 Current I/O Bottlenecks

In a cluster-based environment, MPI-IO is a commonly used middleware to parallelize I/O at the application layer. RAID devices can improve the throughput of a single I/O stream, and cluster-based I/O throughput can be improved significantly with the combination of these two schemes. However, given the limitations of MPI-IO and the underlying parallel disk arrays, the performance of the I/O subsystem is still limited and there remains a significant amount of parallelism to be exploited.

The bottlenecks that occur due to access to shared, centralized, storage devices are due to:

- Network latency - centralized disk devices are usually connected to one or multiple nodes via a shared network. Network contention and network latency become part of the time needed to access data on the shared disk.

- Disk contention - disk contention occurs when multiple nodes need to access the same disk concurrently; concurrent access is very common in many I/O intensive applications.
- Disk access latency - to expose more parallelism at that the disk level, data can be stored on interleaved disks. Even using interleaved schemes, disk access speed is on the order of tens of microseconds, while processor speeds continue to climb into the range of hundreds of picoseconds. Disk latency poses a significant barrier to obtaining better performance without the potential for hiding the overhead by overlapping accesses to many disks.

1.3 Our Approach to Parallel I/O

The ultimate goal in this work is to increase I/O parallelism by creating multiple, independent I/O streams. In order to effectively exploit additional I/O parallelism and reduce I/O traffic to centralized disk devices, I/O workload must be partitioned and distributed across multiple disks. In this paper, we present a profile-guided file partitioning algorithm that achieves scalable I/O speedup by moving the high I/O demands from the centralized device to local disks.

To effectively partition the I/O workload, we first need to characterize file access patterns by profiling requests generated by a number of parallel applications. We have developed a greedy heuristic to partition our file I/O based on file access patterns. One key observation is that we have found that in most applications, the generated partition is insensitive to changes in data values. We have also found that while input dataset sizes can change, that the resulting impact on the original partitioning will be small, requiring only minor modifications to the partition. We discuss issues related to profile sensitivity in Section 4.

The rest of this paper is organized as follows. Section 2 will present our partitioning algorithm. Section 3 will describe the target Beowulf cluster used in this paper. Section 4 will present our workloads and the experimental results, and Section 5 will provide a summary of the paper.

2 File Partitioning

In a cluster-based system, access to datasets stored on a set of centralized storage devices can experience delay due to disk contention and network latency. To achieve high performance I/O, it is critical to create parallel streams of access, utilizing many disks to provide data to compute nodes. A shared disk device can be accessed by all nodes within the cluster via a network connection; local disks can only service I/O requests from the nodes they are attached to.

When using MPI, processes are assigned to individual nodes in the cluster. The partitioning of an application is typically based on data access patterns (e.g., slicing or sub-griding matrices, or spawning off iterations of a loop). The result is that we will have a number of processes, each assigned to a node in the cluster. At this point, little attention has been paid to whether file I/O is localized to a single processor, or if it is spread across all processes. To better understand this issue, we attempt to characterize file access patterns. We classify data chunks into two classes:

1. *local data* and
2. *shared data*

Data chunks that are referenced by only one node are called local data and should be stored on the corresponding local disk. Shared data are those chunks that are accessed by multiple nodes. Several storage strategies can be used to manage shared data chunks:

- **Data duplication** - duplicate the shared chunk on each disk;

- **Data broadcast** - store the chunk on one disk and broadcast updates to other disks who need it;
- **Data centralization** - put the shared chunk on centralized, shared, devices;
- **Combinations of the above policies.**

Selection of the best policy from the above options is heavily dependent on the nature of the parallel application. Therefore, it is important to carefully characterize parallel access patterns before following a particular strategy.

2.1 Access Pattern Characterization

To identify file access patterns, several approaches have been proposed. Memik et al. designed a compiler-based scheme to capture access patterns [10]. Madhyastha and Reed describe how to use *learning algorithms* to predict access patterns during execution [9]. In this paper, we propose a profile-guided approach to characterize access patterns. We have already reported on an early version of this approach [24]. We have developed a greedy algorithm to partition a file that results in improved spatial locality.

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

- process ID,
- file handler,
- address of each data chunk accessed,
- chunk 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.

2.2 Greedy 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 cluster storage system. For every contiguous data chunk accessed, we classify the chunk as either a local or shared (as defined above). If the chunk is local, we can simply assign it to the local disk of the associated node. For datasets that are referenced by multiple processes, we use a hybrid storage strategy. We have developed a graph-based partitioning strategy that attempts to automate the partitioning process. Graph construction works as follows.

In our graph, there are two kinds of nodes: *process nodes* and *data nodes*. We create a process node for every MPI process and a data node for each unique contiguous data chunk. Process nodes and data nodes are connected with an undirected weighted edge to denote that a particular process accesses that data chunk. We keep track of the number of incident edges on each data node (we will refer to this as the data node weight). Since we classify a data node as local if the particular

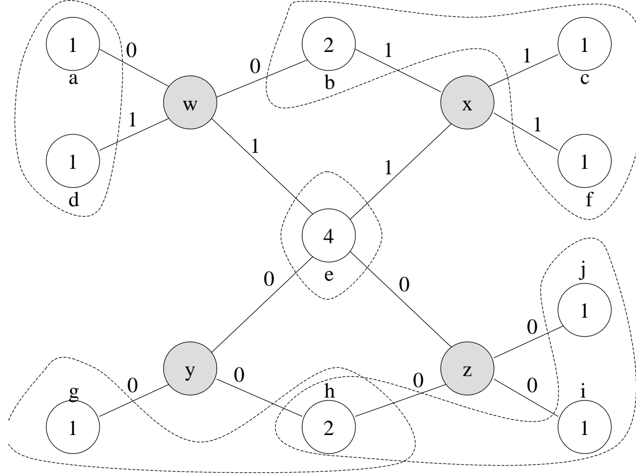

Figure 1: A file access profile, generating 5 partitions.

chunk is accessed by a single process, a local data node will have a weight of 1, and a shared data node will have a node weight greater than 1. In our current implementation, the weight of an edge is assigned a 0 or 1, if it is a read access or write access, respectively. If a chunk is both read and written, the edge value will be 1. No two data nodes share an edge and no two process nodes share an edge. Table 1 summarizes our partitioning heuristics. We then traverse the graph, generating a partition for each process node, easily identifying the data nodes that are connected to it, and use the graph to guide data layout based on rules provided in Table 1.

Data node weight	Edge weight	Data layout
1	any	Place on Local Disk
> 1	all $0's$	Data Duplication
> 1	only a single 1	Data Broadcast
> 1	other edge values	Data Centralization

Table 1: Partitioning Algorithm

After partitioning is complete, for each partition, we sort the data chunks based on the earliest timestamp. The complexity of the entire partitioning algorithm is $O(n^2 * p)$, where n is the number of unique, contiguous, data chunks accessed by the program and p is the number of MPI processes.

Figure 1 is an example of using our graph-based partitioning algorithm. In this example, there are 4 process nodes ($w - z$) and 10 data nodes ($a - h$). Based on our algorithm, five partitions are generated. Data chunks a, c, d, f, g, i and j are stored on the local disk of their corresponding process; chunk b, which is written to by process x and read by process w, is partitioned using the broadcast strategy; node e is written by multiple processes and so should be assigned to the centralized shared storage partition; node h is read-shared by 2 processes and so should be copied to both local disks associated with process nodes y and z.

Compared with a centralized storage system, this partitioned storage system effectively distributes the I/O workload across multiple local disks, reduces contention for the shared disk, and improves I/O parallelism significantly.

```
Foreach I/O process
  Create a Partition;
Foreach contiguous data chunk
  Sort the number of accesses based on Process ID;
     If the chunk is accessed by only one Process ID
       Assign the chunk to the associated partition;
     If the chunk is read (but never written) by multiple processes
       Replicate the chunk in all partitions where read;
     If the chunk is written by one partition, but later read by
     multiple partitions
       Assign the chunk to all partitions where read
       and broadcast the updates to all partitions on writes;
     Else
       Assign the chunk to a shared partition;
Foreach Partition
  Sort chunks based on the earliest timestamp for each chunk;
```

Figure 2: Pseudocode for our partitioning heuristics.

In future work we will consider local nodes based on the ratio of edge weight values (i.e., data nodes connected to multiple process nodes, though when one edge weight dominates, the node will be classified as a local chunk). We will then maintain edge weights as a tuple, recording the read/write accesses on a per process basis. We plan to apply more classical graph partitioning algorithms such as the Kernighan and Lin algorithm [6] to produce a better partition.

After we complete file partitioning, we then need to define the ordering of chunks in each file partition. We utilize an ordering heuristic that places chunks in the file based on their earliest time of access (a chunk may be accessed many times by a process, but we only consider the timestamp of the first access).

Figure 2 provides pseudocode of our greedy partitioning algorithm. The complexity of this algorithm is $O(n^2 * p)$, where n is the number of chunks accessed by the program and p is the number of partitions.

3 Experimental Cluster

We have used MPI and MPI-2 to parallelize our applications. We ran each application on our Beowulf Cluster, generating profile data. We then applied our I/O partitioning scheme and reran the application.

The Beowulf Cluster used in this work has 32 nodes; each node has a local 8.4 GB IDE disk and there are shared SCSI RAID devices directly attached to four of the nodes. In the performance numbers provided, for configurations of 4 and 8 nodes, we use a RAID device hosted on an additional node outside the configuration. For configurations with more than 8 nodes, the RAID device is hosted by one of the nodes contained in the configuration (i.e., a single node serves as both a compute node and an I/O node for the RAIDed SCSI traffic). All I/O is directed to a SCSI RAID device. Our current system uses NFS to manage the RAID device, though in future work we will study a parallel file system such as the Parallel Virtual File System [15].

Figure 3 shows a picture of our 32-node Beowulf Cluster where we performed this work. Table 2

Number of nodes	32 - (27 standard nodes, 4 RAID device host nodes and 1 SMP node)
Processor Type	Intel Pentium II 350 (standard nodes and RAID nodes) 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 3
Network NIC	10/100 Ethernet Cisco Catalyst 2924 Switch Intel 82558 10/100Mb

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

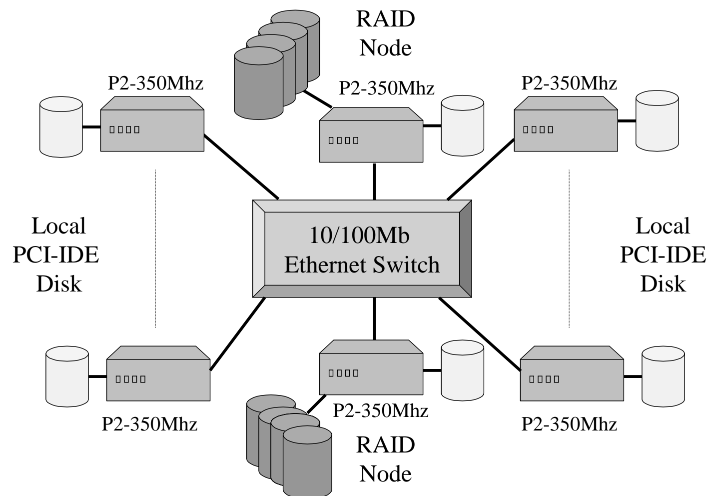

Figure 3: The 32-node Joulian Beowulf Cluster. Complete information on the cluster can be found at *joulian.hpcl.neu.edu*.

Disk/Operation	128	512	1K	2K	4K	32K	64K	128K	512K	1 MB
IDE read	7.1	11.9	13.0	10.1	9.8	7.6	8.5	8.2	8.1	9.0
SCSI read non-local	0.4	1.1	2.1	3.9	5.6	7.0	7.5	10.7	9.9	8.9
SCSI read local	9.8	13.8	14.6	16.7	16.4	19.5	16.1	17.9	17.1	17.9
IDE write	2.6	3.2	3.7	4.6	4.5	4.1	4.1	4.4	4.9	4.1
SCSI write non-local	0.2	0.6	0.8	1.7	2.8	2.1	2.8	2.8	3.3	3.3
SCSI write local	4.6	8.2	9.9	12.5	11.6	12.9	11.2	10.0	11.3	12.1

Table 3: Raw bandwidth rates in MBs per second for our Beowulf Cluster. IDE disks are locally connected; bandwidth rates are provided for both non-locally connected SCSI disks and locally connected SCSI disks.

provides additional details about the hardware. In the results presented, all runs used standard nodes and one RAID-connected node. The SMP node is not used in this study.

Table 3 provides raw bandwidth rates for a local access to an IDE disk, a non-local access to the SCSI disk, and a local access to a SCSI disk. Read/write rates are provided for different chunk sizes. Non-local SCSI access assumes that the I/O must communicate across the 100Mb switched ethernet network to transfer data, as well as to read or write to the SCSI disk.

4 Experiments

4.1 Parallel I/O Workloads

Our target applications are parallelized scientific and multimedia applications, and parallel I/O benchmarks that require intensive disk accesses.

Name	Source	Language	Parallel I/O implementation
NPB/BT	NASA Parallel Benchmark 2.4	Fortran	Unix,MPI-IO
SPECseis	SPEChpc96	Fortran, C	Unix
mpi-tile-io	Parallel I/O Benchmarking Consortium	C	MPI-IO
perf	MPICH	C	MPI-IO
Mandelbrot	Synthetic	C	MPI-IO
Jacobi	Univ. of Georgia	C	MPI-IO
FFT	MPI-SIM	C	MPI-IO

Table 4: Parallel I/O workloads used in this work

We report on the speedup obtained from profile-guided partitioning for seven applications/benchmarks:

- The NPB2.4/BT benchmark 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 dataset (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). Three parallel I/O schemes are studied with this benchmark: parallel Unix I/O, MPI I/O (source is provided in the benchmark source for these two implementations) and partitioned I/O (our own implementation).

- The SPECseis96.1.2 benchmark 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 dataset (1.6 gigabytes) and during phase 2 it reads the dataset back. Each process writes 96KB chunks, and each process then reads back 2KB chunks. Three parallel I/O schemes are studied with this benchmark: parallel Unix I/O (provided in the benchmark source), MPI I/O (our own implementation) and partitioned I/O (our own implementation).
- The MPI-Tile-I/O benchmark is a synthetic benchmark that is part of the Parallel I/O Benchmarking Consortium benchmark suite. The code is written in C. The application implements tile access on a two-dimensional dataset, with overlapped data between sequential tiles. The size of the tiles and the overlap can be user defined. Each process writes 32KB, with 2KB of overlap between consecutive chunks. Two parallel I/O schemes are studied with this benchmark: MPI I/O (provided in the benchmark source) and partitioned I/O (our own implementation).
- The Perf benchmark is a parallel I/O test program provided with the MPICH standard distribution [12]. 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. The chunk size is user-defined. There is no file overlap between chunks. Two parallel I/O schemes are studied with this benchmark: MPI I/O (provided in the benchmark source) and partitioned I/O (our own implementation).
- 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. Two parallel I/O schemes are studied with this benchmark: MPI-IO (provided in the benchmark source) and partitioned I/O (our own implementation).
- Jacobi is a file-based out-of-core jacobi application from University of Georgia [23]. In this program, the initial out-of-core data is stored in a file. The code is written in C. The result of the jacobi iteration is written to a second file. The size of each file is 64 MB. The size of every contiguous data chunk is a function of the number of processes. Two parallel I/O schemes are studied with this benchmark: MPI-IO and partitioned I/O.
- FFT is another file-based out-of-core application from MPI-SIM [2]. It performs Fast Fourier Transform on a disk-resident array of 1M complex numbers. The size of each dataset depends on the number of processes. The input file size is 8M bytes, two temporary files and the final result file are generately dynamically with the size of 8M bytes. The code is written in C. We compare the performance of MPI-IO and partitioned I/O for cluster sizes of 4, 8 and 16 nodes. Since the the number of nodes had to be a power of 2, we were not able to configure a system with 32 nodes.

We evaluate a range of different implementations of these applications. We utilize the MPICH 1.2.1-16 for MPI and MPI I/O. We use the mpicc and mpif77 to compile the C and Fortran codes, respectively.

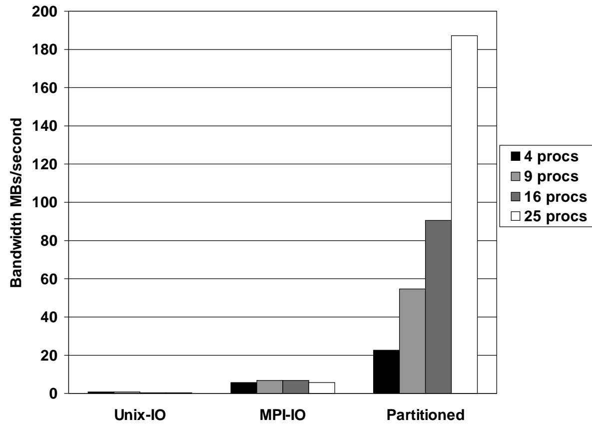

Figure 4: NPB read bandwidth in MBs/second.

4.2 Experimental Results

Figures 4 and 5 present average read/write bandwidth numbers for the NPB/BT benchmark. In these results we measure the end-to-end latency of each access and then aggregate these latencies to obtain an average bandwidth measurement. The NPB/BT benchmark requires the number of processes to be a square number. We compared parallel Unix I/O and MPI I/O with our partitioned I/O scheme. The Unix file operations are generally slow and MPI I/O achieves significant speedup compared with Unix I/O. But MPI I/O can not scale on a parallel system because performance is still bound by the underlying disk subsystem. This fact is evident in all of our results. Comparing partitioned I/O with MPI I/O for NPB/BT, we achieve a speedup factor of up to 32.8X on reads and 17.8X on writes, while also scaling throughput as the size of our system increases.

Figures 6 and 7 show the read and write bandwidth for the SPECseis96.1.2 benchmark. Again, we measured the performance of Unix I/O, MPI collective I/O, as well as our partitioned I/O scheme. Compared with MPI I/O, the performance of partitioned I/O improved by a factor as much as 31.7X on reads and 3.7X on writes. This workload also scales well when using partitioned I/O as we increase the number of nodes.

Figure 8 shows the read and write throughput for the MPI-Tile-IO benchmark. The left two bars in each graph show the read bandwidth and the right two bars show the write bandwidth. The speedup of partitioned I/O over MPI I/O is 15.7X and 5.1X for reads and writes, respectively. The reduction in read bandwidth is due to the amount of overlap between data chunks.

Figure 9 shows the read and write throughput for the Perf benchmark. The left two bars in each graph show the read bandwidth and the right two bars show the write bandwidth. The speedup of partitioned I/O over MPI I/O is 24.6X and 7.7X for reads and writes, respectively.

Figure 10 shows the read and write throughput for the Mandelbrot workload. The figures show both read and write performance for both MPI I/O and partitioned I/O. The speedup for reads is 37.8X and for writes is 20.7X.

Figure 11 shows the read and write throughput for the Jacobi workload. The figures show both read and write performance for both MPI I/O and partitioned I/O. The speedup for reads is 16.0X and for writes is 11.1X.

Figure 12 shows the read and write throughput for the FFT workload. The figures show both

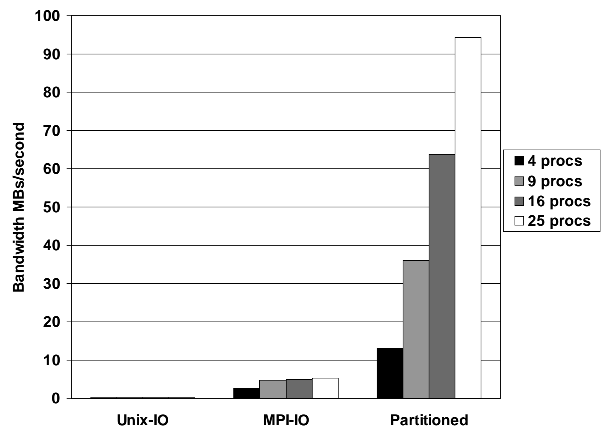

Figure 5: NPB write bandwidth in MBs/second.

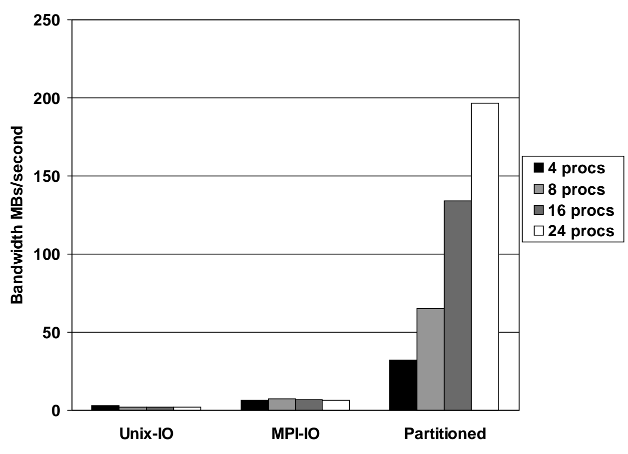

Figure 6: SPEChpc read bandwidth in MBs/second.

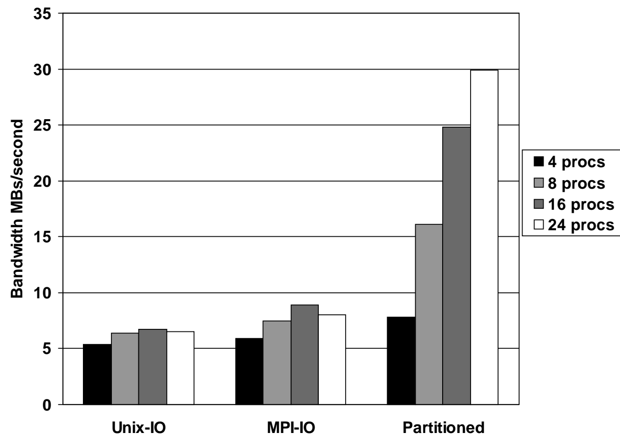

Figure 7: SPEChpc write bandwidth in MBs/second.

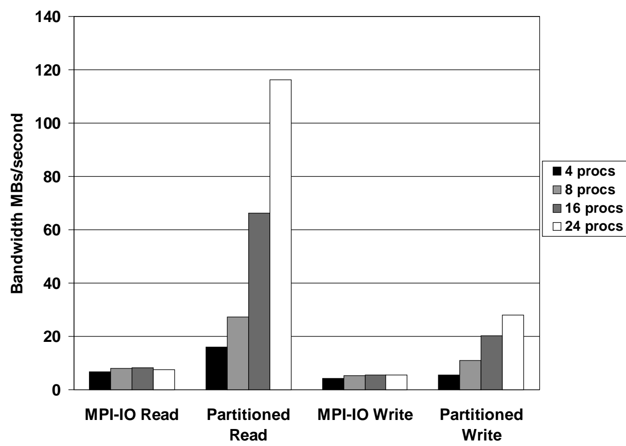

Figure 8: MPI-Tile-IO read and write bandwidth in MBs/second.

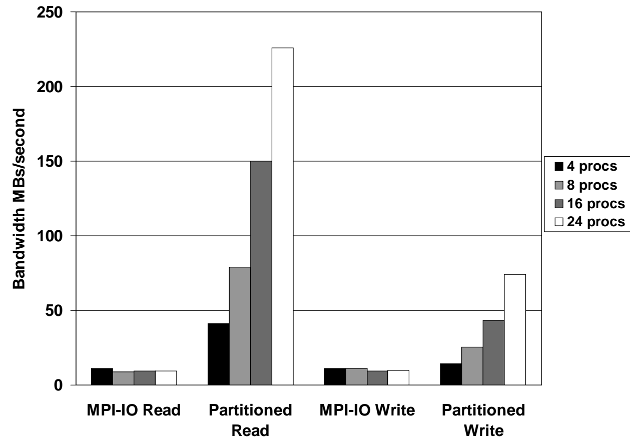

Figure 9: Perf read and write bandwidth in MBs/second.

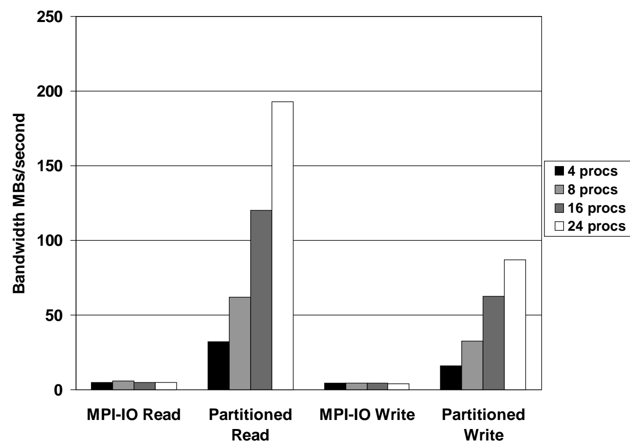

Figure 10: Mandelbrot read and write bandwidth in MBs/second.

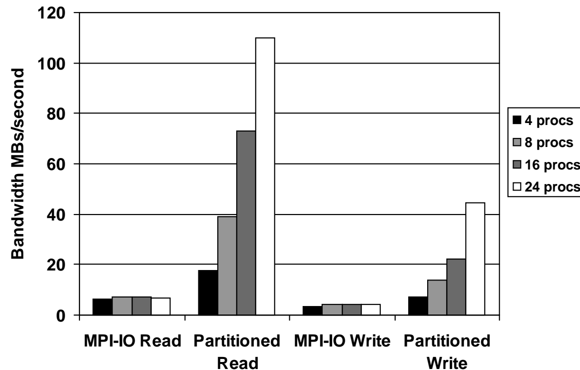

Figure 11: Jacobi read and write bandwidth in MBs/second.

read and write performance for both MPI I/O and partitioned I/O. The speedup for reads is 7.8X and for writes is 4.6X.

Figure 13 shows the overall execution time for our seven applications as run on 24 nodes (25 for NPB-BT, and 16 for FFT). We capture the runtimes for MPI I/O and partitioned I/O. Partitioned I/O achieved a significant reduction for all the benchmarks. The NPB/BT and SPEChpc are both computation-intensive and I/O-intensive applications. The MPI-Tile-IO and Perf benchmarks only issue parallel I/O's. The Mandelbrot program is both I/O and compute intensive, while also performing a significant amount of visualization. The Jacobi and FFT programs are both file intensive and out-of-core applications. The reduction in runtime of these seven workloads is 47.2%, 31.6%, 82.6%, 81.2%, 27.8%, 87.0% and 71.7% for NPB-BT, SPEChpc, MPI-Tile, Perf, Mandelbrot, Jacobi and FFT, respectively.

4.3 Discussion

For a profile-guided approach to be accepted, it needs to be easy to use and resilient to changes in the input training set. As far as providing a user interface, we are presently building our profiling and instrumentation into a compiler pass, using the SUIF-2 infrastructure [4, 8].

We have also evaluated the sensitivity to the profile we generate when we either change the data values stored in files, or when we change the size of the data files. We have found that changes in data values have little to no effect on the profile data, and so there is no need to re-profile an application when the input data set values are changed.

We did experience some measurable differences when we changed the size of the file-based datasets. Through further analysis, we found we could detect two different trends. When increasing the size of the input dataset, either:

- 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 SPEChpc96 and Mandelbrot),
- the size of each chunk increased, while the number of I/O's remained the same (e.g., in NPB2.4/BT, MPI-Tile-IO, Perf, Jacobi and FFT).

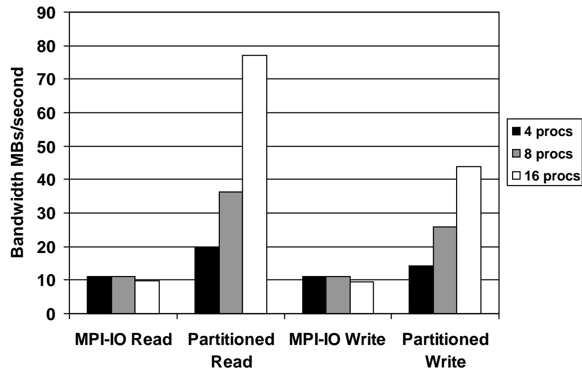

Figure 12: FFT read and write bandwidth in MBs/second.

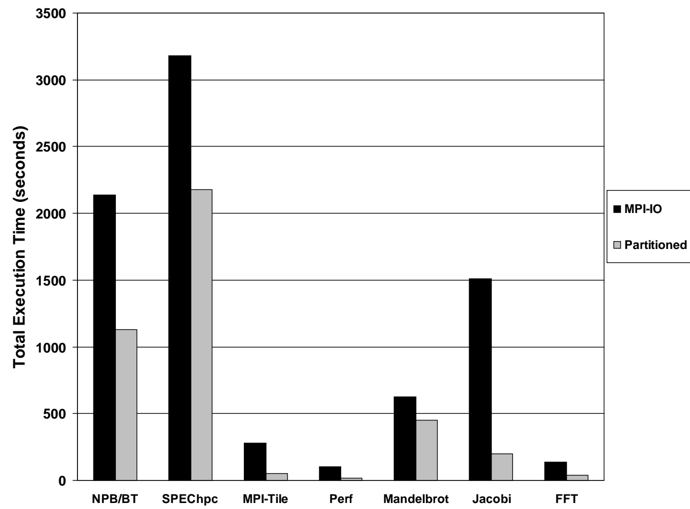

Figure 13: Performance of the entire applications comparing MPI I/O and partitioned I/O.

To better understand which pattern is followed, we can either inspect the application source, or we can re-profile for a short sample. We will quickly see which pattern is followed, and adjust our partitioning for the larger dataset accordingly.

Since most of the applications we have studied access disk storage in fairly uniform patterns, our profiling and partitioning are insensitive to the number of processes assumed during either pass. While this would not be true for less regular accesses, most high performance I/O-dominated applications will exhibit uniform patterns.

5 Acknowledgements

This work was funded by the Institute for Complex Scientific Software (ICSS), the Center for Subsurface Sensing and Imaging Systems (CenSSIS) under the Engineering Research Centers Program of the NSF (Award Number EEC-9986821), and by an NSF Major Research Instrumentation Grant (Award Number MRI-9871022).

6 Summary

In this paper, we have presented a profiled-guided I/O partitioning scheme that utilizes both local and centralized I/O storage within a cluster to improve I/O parallelism and scalability. We have described our profiling technique and described a graph-based analysis that is used to guide partitioning. Data chunks assigned to a partition are then ordered in the file, based on the time of first access. Experimental results were presented that showed that overall execution time of these applications can be reduced by 27.8%-87.0%. We feel that these are very attractive speedups.

In future work we will investigate more aggressive partitioning algorithms that utilize more precise edge weights. We will track the number of accesses per process for each chunk. We can then explore more sophisticated heuristics for partitioning, replication, and broadcasting based on these weights. We also plan to investigate improved partition layouts that take a more global view of temporal and spatial locality of reference in a disk partition.

References

[1] Ching A., Choudhary A. Liao W., Ross R. and Gropp W. Noncontiguous I/O Through PVFS. In *Proceedings of the IEEE International Conference on Cluster Computing*, September 2002.

[2] MPISIM Software. URL: http://pcl.cs.ucla.edu/projects/mpisim/software/.

[3] Garg S. and Mache J. Performance Evaluation of Parallel File Systems for PC Clusters and ASCI Red. In *Proceedings of the IEEE International Conference on Cluster Computing*, October 2001.

[4] Hall M. and Anderson J. and Amarasinghe S. and Murphy B. and Liao S. and Bugnio E. and Lam M. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Transactions on Computers*, 29(12), 1996.

[5] Hwang K., Jin H. and Ho R. RAID-x: A New Distributed Disk Array for I/O-Centric Cluster Computing. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing*, August 2001.

[6] Kernighan B.W. and Lin S. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 29(2):291–307, 1970.

[7] Kotz D. Disk-directed I/O for an Out-of-core Computation. Technical report, Dartmouth College, 1995.

[8] S. M. Extending SUIF for Machine-dependent Optimizations, 1996.

[9] Madhyastha T.M. and Reed D.A. Learning to Classify Parallel Input/Output Access Patterns. *IEEE Transactions on Parallel and Distributed Systems*, 13(8), 2002.

[10] Memik G., Kandemir M.T.,and Choudhary A. Design and Evaluation of a Compiler-directed Collective I/O technique. In *Submitted to Euro-Par 2000*, 2000.

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

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

[13] Message Passing Interface Forum: Extensions to the Message-Passing Interface, 1997. http://www.mpi-forum.org/docs/docs.html.

[14] Patterson D.A., Gibson G.A and Katz R.H. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *ACM SIGMOD International Conference on Management of Data*, pages 109–116, June 1988.

[15] C. P.H., L. W.B., R. R.B., and T. R. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[16] Sang J., Kim C.M., Kollar T.J. and Lopez I. High-performance cluster computing over gigabit/fast ethernet. *Informatica (Slovenia)*, 23(1), 1999.

[17] Schmuck F. and Haskin R. GPFS: A Shared-Disk File System for Large Computing Centers. In *Proceedings of Conference on File and Storage Technologies (FAST'02)*, January 2002.

[18] Seamons K., Chen Y., Jones P., Jozwiak J. and Winslett M. Server-Directed Collective I/O in Panda. In *Proceedings of Supercomputing*, December 1995.

[19] Smirni E. and Reed D.A. Workload Characterization of Input/Output Intensive Parallel Applications. In *Proceedings of 9th International Conference on Computer Performance Evaluation*, June 1997.

[20] Thakur R., Choudhary A., Bordawekar R., More S., and Kuditipudi S. Passion: Optimized I/O for Parallel Applications. *IEEE Computer Magazine*, 29(6), 1996.

[21] Thakur R., Gropp W., and Lusk E. An Abstract-Device Interface for Implementing Portable Parallel-IO Interfaces. In *Proceedings of the 6th Symposium on Frontiers of Massively Parallel Computation*, October 1996.

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

[23] Performance study of parallel Jacobi. URL: (http://webster.cs.uga.edu/ yan/.

[24] Wang Y. and Kaeli D. Profile-Guided I/O Partitioning. In *Proceedings of the International Conference on Supercompuing*, June 2003.

[25] Xu Z. and Hwang K. Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2. *IEEE Concurrency*, 4(1):9–23, 1996.