Profile-Guided I/O Partitioning

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

ABSTRACT

In the field of high performance computing there is a growing need to process large, complex datasets. Many of these applications are *file-intensive* workloads, performing a large number of reads from and writes to a small number of files. When executing these workloads on cluster-based systems, performance cannot scale by simply increasing the number of compute nodes. To effectively exploit parallel resources we need to parallelize file I/O. The potential impact of exploiting parallel I/O grows as the gap between CPU and disk speeds continues to increase.

While parallel I/O middleware systems (e.g., MPI I/O) provide users with environments where large datasets can be shared among multiple distributed processes, the performance of file-intensive applications depends heavily on how the data is accessed and where the data is physically located on disk. I/O operations need to be parallelized both at the application level (using middleware) and at the disk level (using partitioning).

In this paper, we present a new profile-guided greedy partitioning algorithm to parallelize I/O access for file-intensive applications run on cluster-based systems. We are using MPI and MPI I/O to provide parallelization at the application level. We utilize I/O profiling to capture relevant information about the I/O stream. We then use these profiles to guide file partitioning across multiple disks to significantly improve I/O throughput.

Categories and Subject Descriptors

B.4.3 [**Input/Output and Data Communications**]: Parallel I/O

General Terms

Performance

Keywords

parallel I/O, profile-guided I/O, clusters

ICS'03, June 23–26, 2003, San Francisco, California, USA.

1. INTRODUCTION

In the field of high-performance parallel computing we are seeing an increased focus on applications that are both compute-bound and I/O-bound. This trend is especially evident when analyzing massive amounts of image or sensor data. I/O-intensive applications fall into one of two categories:

1. *out-of-core* applications, and
2. *file-intensive* applications.

Applications that work with large data sets that cannot fit in main memory are called *out-of-core* applications. Due to the limited size of main memory, data must be paged between main memory and swap space on disk. A number of techniques have been proposed to improve I/O bandwidth for this class of applications [2, 8, 13, 25], as well as to reduce the number of disk accesses [12].

File-intensive applications access file-based data frequently. Their performance is limited by the large number of file operations (e.g., file open, file read, file seek) that are inherent to the application. One key attribute of file-intensive applications is that we can uniquely identify logical file addresses. Out-of-core file access patterns are dependent on the operating system and the runtime system state. I/O reference patterns can differ significantly between two different executions of the same program. Since file I/O addresses typically remain constant across different runs of a program [1], we can utilize profiling to tune file access in a distributed application. We have also found that when the number of processes changes or the size of the input data file changes, the pattern of file accesses changes very predictably, so re-profiling can be avoided.

In this work, we are focused on improving I/O performance for file-intensive parallel applications. Many scientific and visualization applications exhibit file-intensive access properties. Our goal is to develop efficient I/O partitioning algorithms that can increase parallelism in the I/O subsystem.

1.1 Related Work

There has been a large amount of previous work focused on improving I/O access. Improvements have been

[1]In scientific applications, we have found that file access patterns are generally independent of the data values stored. This assumption may not hold for applications where dynamic decisions are made based on data values, such as database applications.

made at the disk device level and at the application level. Disk manufacturers have provided us with faster disks (in terms of rotational latency), smarter disk controllers (e.g., caching devices), and parallel device access (RAIDed disks). *Striping* data across RAID devices has been shown to be highly beneficial in improving disk throughput [4, 21]. These advances provide for higher disk bandwidth and lower I/O latency. The ideas presented in this paper are orthogonal to these approaches. We attempt to increase the spatial locality in a single I/O stream, while also reducing the contention for a shared device.

There have also been *disk prefetching* strategies proposed that reduce disk read latency. Prefetching predicts the future reference stream, typically based on past reference behavior. Early compiler-based approaches include work by Trivedi that proposed *prepaging*. Patterson et al. proposed the *Transparent Informed Prefetching and Caching* (TIP) system [22], that utilizes explicit hints inserted by the programmer to perform cost-effective prefetching.

Prefetching can also be implemented in the compiler, as was done in the Profet system [3]. The main benefit of working at the compiler level is that no source code modifications are needed, though changes may be needed to the operating system to support this mechanism. Again, prefetching can also be used in cooperation with our techniques to further reduce I/O latency.

In [24], Reddy and Bannerjee studied five scientific applications taken from the PERFECT benchmark suite. They found that most applications exhibited sequential file access patterns. In [23], Bagrodia et al. used Pablo to instrument and characterize out-of-core scientific applications. In [20], Nieuwejaar et al. performed a multi-platform characterization study of file access patterns for parallel scientific workloads as part of the *CHARISMA* project. I/O access patterns were captured on both Intel iPSC/860 and CM-5 platforms. Understanding file access patterns is essential when attempting to partition files. These previous studies have helped guide our selection of workloads and develop our partitioning heuristics.

There has also been significant work to provide parallelism at the application level. Techniques such as data sieving [27] and collective I/O [17] have been successful in improving performance. Many parallel applications require access to small, potentially non-contiguous, data chunks. Collective I/O merges multiple I/O accesses (both contiguous and non-contiguous), generated by multiple processes, into a single I/O function call. MPI I/O [26, 28], which is included as part of the MPI-2 standard release, provides for collective communication so that multiple MPI processes can access a single MPI I/O file. We utilize MPI collective I/O (we will refer to collective I/O as MPI I/O in this paper) in the work presented here. But when multiple processes have to access a single shared file, severe I/O bottlenecks can occur with MPI I/O.

In this paper, we present a methodology for physically partitioning files on local disks in an attempt to remove the bottleneck of access to a single file. We profile I/O execution of parallel applications that utilize MPI and MPI I/O. We have been able to obtain significant speedups with our methodology, cutting application runtimes by 28-82%. The rest of this paper is organized as follows. Section 2 will describe our I/O partitioning algorithm. Section 3 will describe our evaluation environment. Section 4 will provide a series of performance results that illustrate the power of our algorithms. In Section 5 we discuss the implications of these results, as well as issues related to other classes of workloads, and in Section 6 we summarize the contributions of the paper.

2. PROFILE-GUIDED I/O PARTITIONING

Next we describe our profiling methodology and our partitioning algorithm.

2.1 Access Pattern Detection

The ultimate goal for parallel I/O is to increase parallelism by creating multiple, independent, I/O streams. In order to effectively exploit the potential for I/O parallelism, data must be partitioned and distributed across multiple disks. To effectively partition files, we will first profile access patterns produced by an application. Based on this profile, we will generate a file partition for each I/O process. Ideally, each I/O process should only need to access its own file partition on a local disk and then there would be no disk contention or communication overhead. Unfortunately, multiple processes may need to read and write the same file space, so proper partitioning must consider file consistency issues.

Many characterization studies have targeted I/O access patterns [9, 11, 15, 20]. Workload patterns can be recognized both statically (at compile time) or dynamically (at run time). Madhyastha and Reed [15] suggested using learning algorithms to classify I/O access patterns at execution time, guiding adaptive file system policies. Memik et al. [17] designed a compiler technique to direct collective I/O by matching read and write access patterns statically. Our approach is to profile the dynamic execution of an application and then apply a *greedy* partitioning algorithm that both partitions files, as well as improves spatial locality within the partition. We have used profile-guided partitioning in previous work to guide parallelization of MPI-based imaging applications [14]. We have also used memory profiling to study the characteristics of shared memory workloads [7].

To profile I/O accesses, we capture the following information:

- process ID
- file handle
- address of the contiguous data chunk accessed
- chunk size
- access type (read/write)
- timestamp

A library function has been developed to capture this information on every file operation. In the applications studied, we experience a 30% slowdown on average when profiling is enabled. We then use this profile to guide file partitioning across multiple disks to achieve high I/O parallelism.

2.2 Profile-Guided Optimization

The I/O file partitioning problem is an NP-hard problem. To produce the best partitioning, all possible partitions would need to be explored. To produce an effective partitioning of all files, we have developed a greedy algorithm.

Using our algorithm, for every contiguous data chunk we identify which process accesses each chunk most frequently. We then assign that data chunk to the file partition associated with the particular process-ID. For data chunks that are heavily accessed by multiple processes, we consider the following criteria. 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 processes, we create a shared file partition, and allow all processes to access this partition. In the case where a chunk is first written by a single process and later read by multiple processes, we let the write 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 the data chunks in a partition. The ordering of chunks in each partition is based on the earliest timestamp recorded for each chunk (chunks may be accessed many times, though we only consider the time of the first access in this ordering). Figure 1 provides pseudocode for 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 ENVIRONMENT

We have used MPI and MPI-2 to parallelize our applications. Next, we run each application on our Beowulf Cluster, generating profile data. Then we apply our I/O partitioning scheme and rerun.

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 single 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 the I/O node for the RAIDed SCSI traffic). All I/O is directed to a single SCSI RAID device.

Figure 2 shows a picture of our 32-node Beowulf Cluster where we performed this work. Table 1 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 2 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

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

4.1 Parallel I/O Workloads

We report on the speedup obtained from profile-guided partitioning for five 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 [19]. 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 I/O (provided in the benchmark source) and partitioned I/O (our own implementation).

```
Foreach I/O process
  Create a Partition;
Foreach 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 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 1: Pseudocode for our greedy partitioning algorithm.

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
Parallel file system	NFS 3
Network NIC	10/100 Ethernet Cisco Catalyst 2924 Switch Intel 82558 10/100Mb

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

Disk/Operation	128	512	1K	2K	4K	32K	64K	128K	512K	1MB
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 2: 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.

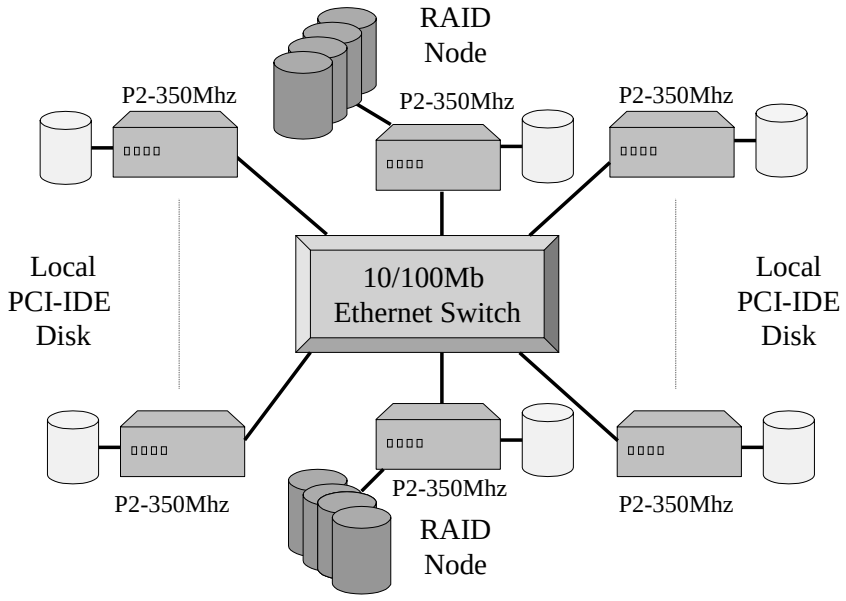

Figure 2: The 32-node Joulian Beowulf Cluster. Complete information on the cluster can be found at *joulian.hpcl.neu.edu*.

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.

4.2 Experimental Results

Figures 3 and 4 present average read/write bandwidth numbers for the NPB/BT benchmark. 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 that the number of processes be a square number. We compared parallel Unix I/O and MPI I/O with our partitioned I/O scheme. Unix file operations are generally slow and MPI I/O achieves significant speedup in comparison. 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 5 and 6 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 on a larger number of nodes.

Figure 7 shows the read and write throughput for the MPI-Tile I/O benchmark. The left two bars 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

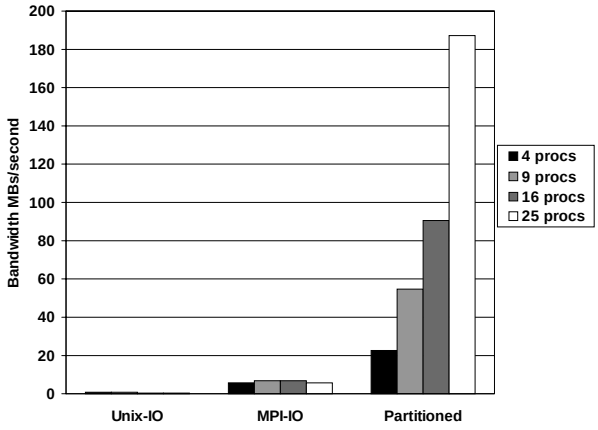

Figure 3: NPB read bandwidth in MBs/second.

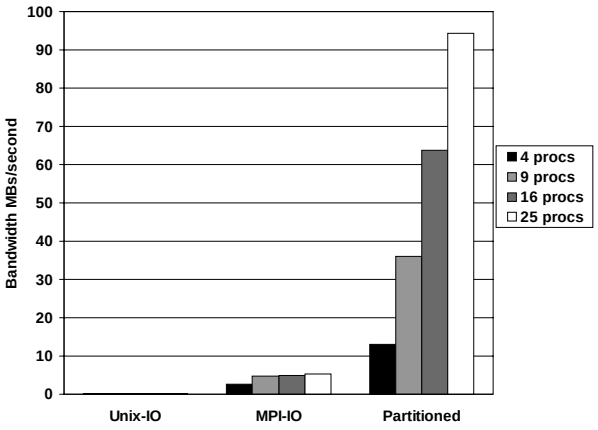

Figure 4: NPB write bandwidth in MBs/second.

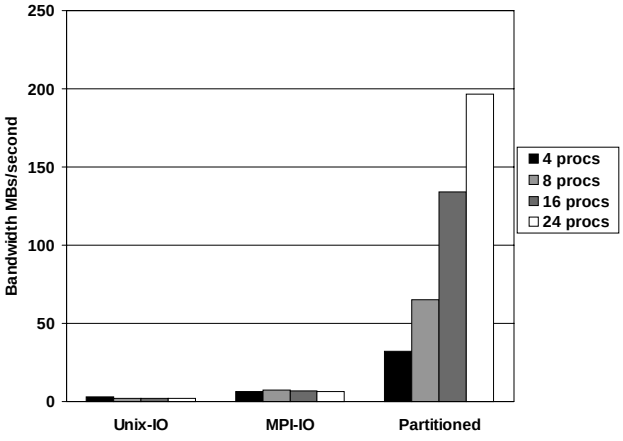

Figure 5: SPEChpc read bandwidth in MBs/second.

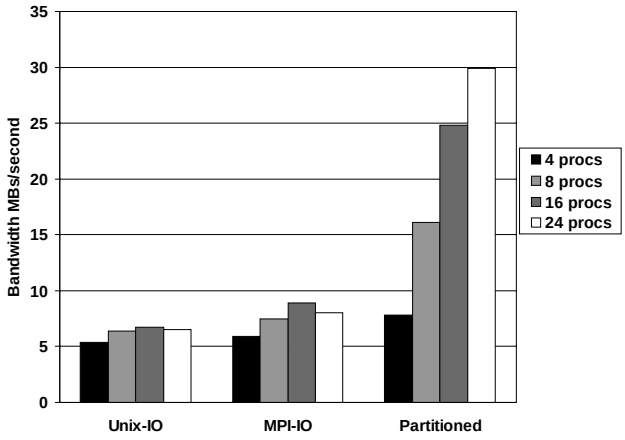

Figure 6: SPEChpc write bandwidth in MBs/second.

5.1X for reads and writes, respectively. The reduction in read bandwidth is due to the amount of overlap between data chunks.

Figure 8 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 9 shows the read and write throughput for the Mandelbrot workload. The figure shows 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 10 shows the overall execution time for our five applications as run on 24 nodes (25 for NPB-BT). 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 I/O and Perf benchmarks only issue parallel I/O's. The Mandelbrot program is I/O and compute intensive, while also performing a significant amount of visualization. The overall reduction in runtime of these five workloads are 47.2%, 31.6%, 82.6%, 81.2% and 27.8% for NPB-BT, SPEChpc, MPI-Tile, Perf, and Mandelbrot, respectively.

5. DISCUSSION

In this paper, we have presented a profile-guided approach to achieve scalable I/O speedup. We have been able to create multiple I/O channels, improve 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, the technique needs to be resilient to changes between the input training set and the runtime dataset. We have evaluated the sensitivity to various changes in terms of the stored data values, the number of processes and the dataset sizes. We have found that changes in data values have little effect on the profile data, and so there is no need to re-profile an applica-

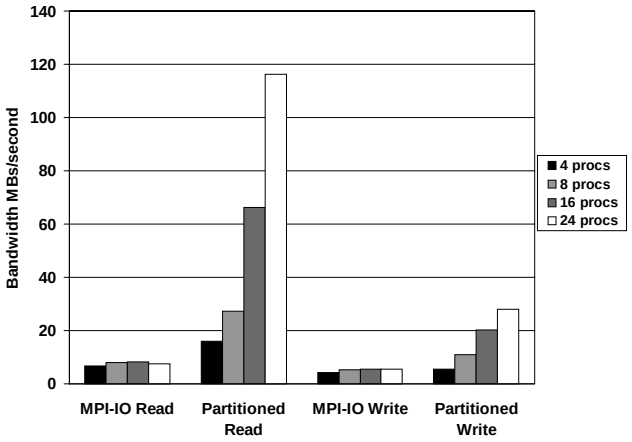

Figure 7: MPI-Tile I/O read and write bandwidth in MBs/second.

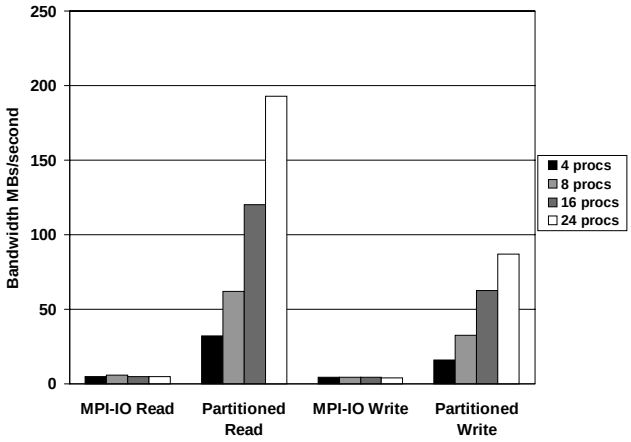

Figure 9: Mandelbrot read and write bandwidth in MBs/second.

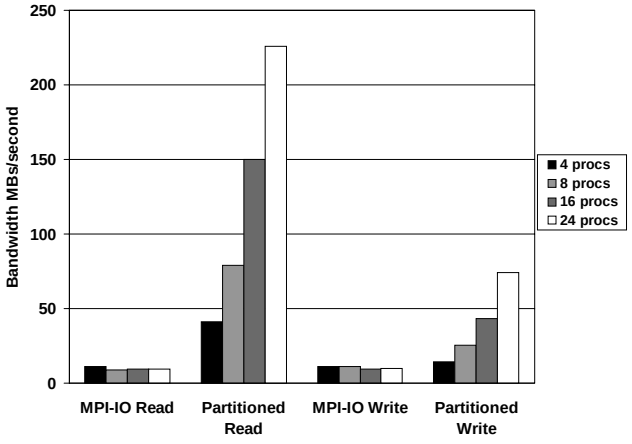

Figure 8: Perf read and write bandwidth in MBs/second.

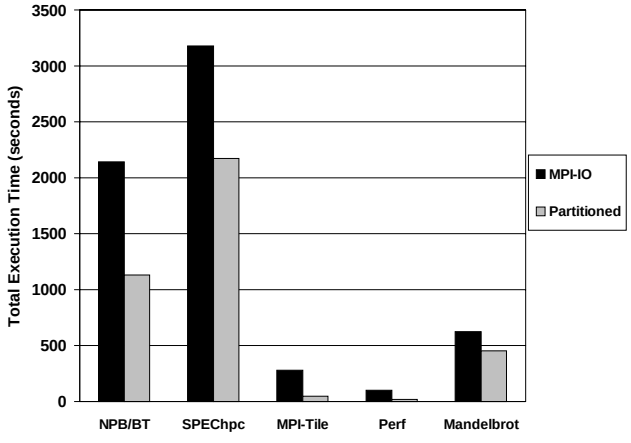

Figure 10: Performance of the entire applications comparing MPI I/O and partitioned I/O.

tion when the input data set values are changed. When we change the number of processes, the chunk size will change predictably. For instance, in NPB2.4/BT benchmark, we have made the following observations:

- file access patterns remain the same when data values change; and
- data chunk sizes vary with the number of processes.

For example, when using 16 processes, each process accesses chunks of 1040 bytes; when we move to 25 processes, the chunk size decreases to 800 bytes. The chuck size decreases by a factor governed by the square root of the number of processes. While chunk sizes change with a changing number of processes, the write and read access patterns are independent of the number of processes.

We did experience some 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 both SPEChpc96 and Mandelbrot), or
- the size of each chunk increases, while the number of I/O's remains the same (e.g., in NPB2.4/BT, MPI-Tile-IO, and Perf).

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.

For other classes of the file intensive applications, such as multimedia and database, file access patterns will differ from parallel scientific applications. This difference begins to become apparent in our imaging application (Mandelbrot), where we obtained the smallest overall speedup. Multimedia applications tend to access fixed-content files sequentially. Database applications typically access small data chunks and the access patterns can be dependent on the data values in the database. One issue that arises for database applications is that the I/O latency time, instead of the I/O throughput, becomes the dominant factor for application performance. Since chunk sizes are typically much smaller (on the order of 10-100 bytes), the benefits of partitioning may not be as dramatic as were found in the scientific applications studied.

Another direction for our work is to implement file partitioning as a step in compilation. Since many I/O reference patterns can be identified statically, we can exploit this information during compilation. This approach mimics backend compiler optimizations that have been proposed to improve the layout of static [10, 16] and dynamic data structures [5] to better utilize multi-level memory hierarchies.

While we have focused on file-intensive applications, if we begin to process larger file inputs, we will begin to see that these applications are also memory bound (i.e., out-of-core). Larger files will pose new challenges for us, trying to manage the allocation of disk bandwidth between file access traffic and swap traffic. We are considering how to combine compile-time out-of-core techniques [1, 18] with compile-time file partitioning, all in one compiler framework. We have already begun to study some out-of-core sorting and permutation codes that are also file-I/O intensive [6].

6. 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), and by and the NSF Major Research Instrumentation Program (Award Number MRI-9871022).

7. SUMMARY

To achieve scalable I/O performance of parallel applications, it is important to parallelize I/O streams at both the application and file levels. In this paper, we present a new approach to parallelize I/O accesses by generating I/O profiles. We then use these profiles to determine how best to partition the data across multiple disks to achieve high I/O parallelism. For the applications we have studied, partitioned I/O reduced overall execution time by as much as 82% compared with MPI collective I/O.

In future work we are extending our set of workloads to consider both multimedia and database file-intensive applications. We are also considering how best to estimate file access characteristics at compile time, removing the need to obtain profile runs of the program.

8. REFERENCES

[1] R. Bordawekar, A. Choudhary, and R. Thakur. Data Access Reorganizations in Compiling Out-of-Core Parallel Programs on Distributed Memory Machines. Technical report, September 1994.

[2] P. Brezany, A. Choudhary, and M. Dang. Language and Compiler Support for Out-of-Core Irregular Applications on Distributed-Memory Multiprocessors. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 343–350, 1998.

[3] A. D. Brown, T. Mowry, and O. Krieger. Compiler-Based I/O Prefetching for Out-of-Core Applications. *ACM Transactions on Computer Systems*, 19(2):111–170, 2001.

[4] P. Chen and E. Lee. Striping in a RAID Level 5 Disk Array. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 136–145, Ottawa, Canada, 15–19 1995.

[5] T. Chilimbi, M. Hill, and J. Larus. Cache-Conscious Structure Layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[6] G. Cooperman and X. Ma. Overcoming the Memory Wall in Symbolic Algebra: A Faster Permutation Algorithm. In *SIGSAM Bulletin*, 2003.

[7] D. Kaeli, L. Fong, D. Renfrew, K. Imming and R. Booth. Performance of a CC-NUMA Prototype. *IBM Journal of Research and Development*, 41(3):205–214, 1997.

[8] D. Kotz. Disk-directed i/o for an out-of-core computation. Technical report, 1995.

[9] E. Smini and D.A. Reed. Lessons from Characterizing the Input/Output Behavior of Parallel Scientific Applications. *Performance Evaluation*, 3:27–44, 1998.

[10] D. Genius and S. Lelait. Improving Data Layout through Coloring-directed Array Merging. Technical Report iratr-1999-3, Universität Karlsruhe, 1999.

[11] H. Simitci and D.A. Reed. A Comparison of Logical and Physical Parallel I/O Patterns. *International Journal of High Performance Computing Applications*, 12(3):364–380, 1998.

[12] M. Kandemir, R. Bordawekar, A. Choudhary, and J. Ramanujam. A Unified Titling Approach for Out-of-Core Computation. Technical report, 1996.

[13] M. Kandemir, A. Choudhary, J. Ramanujam, and R. Bordawekar. Optimizing Out-of-Core Computations in Uniprocessors. In *Proceedings of the Workshop on Interaction between Compilers and Computer Architectures*, pages 1–10, 1997.

[14] M. Ashouei, D. Jiang, W. Meleis, D. Kaeli, M. El-Shenawee, E. Mizan, Y. Wang, C. Rappaport and C. DiMarzio. Profile-based characterization and tuning for subsurface sensing and imaging applications. *International Journal of Systems, Science and Technology*, pages 40–55, Sep 2002.

[15] T. Madhyastha and D. Reed. Learning to classify parallel input/output access patterns. *IEEE Transactions On Parallel And Distributed Systems*, 13(8), 2002.

[16] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[17] G. Memik, M. Kandemir, and A. Choudhary. Design and Evaluation of a Compiler-directed Collective I/O technique. In *Proceedings of 6th Annual EuroPar Conference*, pages 1263–1272, Aug-Sept 2000.

[18] K. Moor. I/O Performance Enhancements of Out-of-Core Applications. Notre Dame University, Department of Computer Science and Engineering.

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

[20] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, and M. Best. File-Access Characteristics of Parallel Scientific Workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, 1996.

[21] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, 1995.

[22] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 79–95, Dec 1995.

[23] R. Bagrodia, A. Chien, Y. Hsu and D. Reed. Input/output: Instrumentation, characterization, modeling and management policy. Technical report, CalTech Concurrent Supercomputing Facilities, CalTech, 1994.

[24] A. N. Reddy and P. Bannerjee. A Study of I/O Behavior of Perfect Benchmarks on a Multicomputer. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 312–321, 1990.

[25] R. Thakur, R. Bordawekar, and A. Choudhary. Compiler and Runtime Support for Out-of-Core HPF Programs. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 382–391, Manchester, UK, 1994. ACM Press.

[26] R. Thakur, W. Gropp, and E. Lusk. *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation*. Mathematics and Computer Science Division, Argonne National Laboratory, Oct. 1997. ANL/MCS-TM-234.

[27] 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*, February 1999.

[28] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, 1999.