Design and Implementation of a Parallel I/O Runtime System for Irregular Applications [*]

Jaechun No [†] Sung-soon Park [‡] Jesús Carretero [§] Alok Choudhary [¶] Pang Chen [||]

Abstract

In this paper we present the design, implementation and evaluation of a runtime system based on collective I/O techniques for irregular applications. We present two models, namely, "Collective I/O" and "Pipelined Collective I/O". In the first scheme, all processors participate in the I/O simultaneously, making scheduling of I/O requests simpler but creating a possibility of contention at the I/O nodes. In the second approach, processors are grouped into several groups, so that only one group performs I/O simultaneously, while the next group performs communication to rearrange data, and this entire process is pipelined to reduce I/O node contention dynamically. Both models have been optimized by using software caching, chunking and on-line compression mechanisms. We demonstrate that we can obtain significantly high-performance for I/O above what has been possible so far. The performance results are presented on an Intel Paragon and on the ASCI/Red teraflops machine at Sandia National Labs.

1 Introduction

Parallel computers are being used increasingly to solve large irregular applications with huge I/O requirements [5]. A typical computational science analysis cycle requires storing visualization and checkpoint data (which can run into 100s of MBs to TBs range) in a canonical form so that other tools can use them easily without having to reorganize the data and data storage can be independent of the number of processors that produced it. Such requirements present challenging I/O problems that may overwhelm application programmers. As the number of processors scales (e.g., to more than 4500 on the ASCI/Red Teraflops machine at Sandia National Labs), sequential re-combination of the anticipated terabytes of data simply becomes infeasible.

In this paper we present the design and implementation of a high-performance runtime system which can support all of the types of I/O listed above on large-scale systems. In particular, the system enables accessing irregular data sets, does data reorganization (sorting) on the fly, enhances I/O operations by using collective I/O [2], and balances I/O workload among processors dynamically. First, we present a collective I/O technique that handles irregular accesses to large data sets by reorganizing the data on-the-fly. Second, we enhance this technique and introduce the notion of "pipelined collective I/O" to manage I/O node contention dynamically. Third, we present software caching techniques where data from different nodes may be reused, thereby avoiding or reducing I/O. Fourth, we incorporate compression mechanisms in the runtime system to reduce the amount of I/O required at the expense of additional computations. Finally, we present performance results on large-scale parallel systems including the teraflop machine at Sandia National Labs. These runtime functions are being incorporated in teraflop applications. We demonstrate that we can obtain significantly high-performance for I/O above what has been possible so far.

2 Design of the Collective I/O Operations

There are several characteristics to be considered to develop an I/O library in order to support irregular problems. One of them is that irregular problems often generate fine-grained data distributions requiring access to non-contiguous locations in a global array. Our library uses the two-phase I/O strategies [1] to convert a large number of small and disjoint I/O requests into a small number of large contiguous requests. Several factors must be considered in the design of a library based on this technique: buffer size used by the library, communication schedule construction and reorganization, the number of processors participating in I/O at any time, and scheduling of I/O requests. In addition, since data accesses are performed using a level of indirection in irregular computations, the number of passes through the data sets to compute schedule and reorganize

[*] This work was supported in part by Sandia National Labs award AV-6193 under the ASCI program, and in part by NSF Young Investigator Award CCR-9357840 and NSF CCR-9509143. Work performed while all authors were at Northwestern University.

[†] Dept of Electrical Engineering and Computer Systems, Syracuse University, jno@ece.nwu.edu

[‡] Dept of Computer Science, and Engineering, Anyang University

[§] Arquitectura y Tecnología de Sistemas Informáticos, Universidad Politécnica de Madrid

[¶] Dept of Electrical and Computer Engineering, Northwestern University

[||] Sandia National Laboratories, Albuquerque, NM 87185

data also are important. In this section we present the design of two alternative schemes for collective I/O. The first scheme is called *"Collective I/O"* and the second scheme is called *"Pipelined Collective I/O"*.

2.1 Collective I/O and Pipelined Collective I/O

These designs involve three basic steps. 1) Schedule construction, 2) reading data from files; and 3) redistributing data into appropriate locations of each processor. In the write operation, redistribution step precedes the file write step. Schedule describes the communication and I/O pattern required for each node participating in the I/O operation. The indirection arrays, used to reference the data array, must be scanned to consider each element of the array individually to determine its place in the global canonical representation as well as its destination processor for communication.

Several factors affect the schedule construction in particular, and the overall I/O library design and performance in general. These include the following. 1) *Chunk Size*, which is the amount of buffer space available to the runtime library for the I/O operations. For example, if the total size of the data per processor to be read/written is 8 MB and the chunk size is 2MB, then the I/O operation will require four iterations to complete. A schedule must be built for each of these iterations. 2) *Number of processors involved in I/O*, which determines the communication among processors to redistribute data. The steps involved in computing schedule information are briefly described below:

- Based on the chunk size, each processor is assigned a data domain for which it is responsible for reading or writing. Next, with each index value in its local memory, processor first decides from which chunk the appropriate data must be accessed and then determines which processor is responsible for reading the data from or writing the data to the chunk.

- Index values in the local memory are rearranged into the *reordered-indirection array* based on the order of destination processors to receive them. Therefore, we can communicate consecutive data between processors(communication coalescing).

Note that once it is constructed, the scheduling information can be used repeatedly in the irregular problems whose access pattern does not change during computation, and thereby amortizing its cost. For collective I/O, each processor involved in the computation is also responsible for reading data from files or writing data into files. Let *D* bytes be the total size of data and *P* be the number of processors. If the size of data chunk is the same as the total size of data, each processor then reads *D/P* bytes of data from the file and distributes it among processors based on schedule information. In case of writing, each processor collects *D/P* bytes of data from other processors and then writes it to the file. By performing I/O this way, the workload can be evenly spread among processors. However, with collective I/O, all processors issue I/O requests to the I/O system simultaneously. As a result, contention at the I/O system may occur if there are large number of compute nodes compared to the number of I/O nodes. In the pipelined collective I/O, we divide processors into multiple processor groups so that contention may be managed dynamically. Only processors in a group issue I/O request simultaneously to reduce I/O node contention. The expectation is that while one group of processors is performing I/O operations, another group performs communication in order to collect(redistribute) data for write(read) operations. Thus, with *G* groups, there will be *G* interleaved communication and I/O steps, where in step *g*, $0 \leq g < G$, group *g* is responsible for the I/O operations.

3 Software Caching and Data Reuse

In irregular applications, same data may be accessed repeatedly during the execution of irregular loops. In global data area, overlap data area between two loops may exist. *Re-referenced data* is that data which has been accessed in a prior loop and will be accessed in a subsequent loop. *Released data* is defined to be the one which once accessed in a prior loop will not be accessed again. Therefore, it should be written back to the file if modified. *New data* is defined to be data referred in a loop but has never been accessed in a prior loop. We have used a software caching scheme to reduce I/O cost by reusing the data. With this method, the communication and I/O patterns required for each processor are determined in the `schedule` phase. I/O phases to execute irregular loops are inserted before and after each loop. The *re-referenced data* need not be written back to file in the first write phase because it will be needed again in further read phases. The basic goals and design of the software caching scheme are as follows. First and foremost, it is to reduce I/O to the maximum extent possible. To achieve this, a read and write I/O phases is divided into two read and write I/O phases, where the second read and write phase only access data that is "new-data". Second goal is to utilize the schedule information which is constructed at the beginning and to build only the incremental schedule for "new-data" needed. The execution phases include two read and write steps, `read1` and `write1`, which are almost the same as for the I/O operations without software caching. A second phase of the algorithm executes partially reading and writing of data, `read2` and `write2`, based upon the overlap area. In this phase, data which will not be used in a subsequent loop is written to file. Similarly, data which has never been read but will be referred in a subsequent loop is read from the file in this phase. Since only some part of data is written to file or read from file, I/O cost for the second phase is expected to be smaller than for the first one. Be-

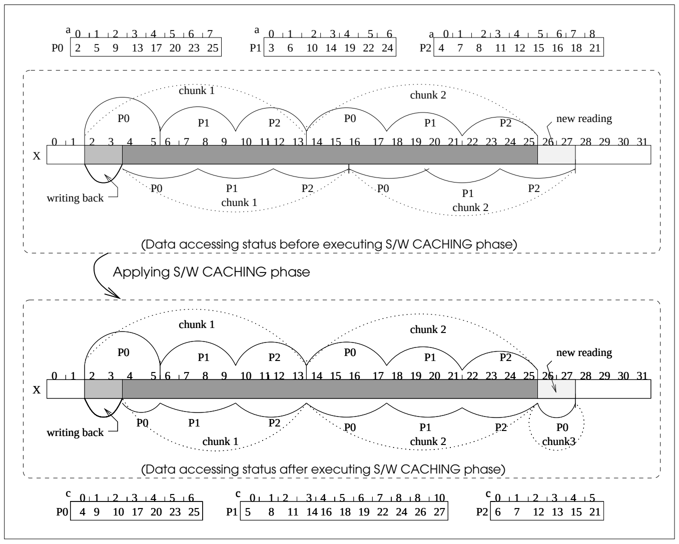
chunk 1
chunk 2
P0
P1
P2
new reading
writing back
X
(Data accessing status before executing S/W CACHING phase)
Applying S/W CACHING phase
chunk3
(Data accessing status after executing S/W CACHING phase)

Figure 1. Data access pattern before and after executing `s/w caching` phase

tween both phases, schedule information for the new loop is reconstructed in the `s/w caching` step. Some details are omitted due to lack of space.

Figure 1 shows the data domain of each processor for an irregular loop before and after executing `s/w caching` step with 3 processors and 2 data chunks, where *X* is global data. Before executing `s/w caching` step, *writing back* area is that which it is referred in only the prior loop and *new reading* area is one which is referred in only later loop. Dark area of *X* represents data referred in both loops. In first phase, `read1`, each processor reads data from file using 2 data chunks. For example, processor *P0* reads *X(2)* ∼ *X(5)* and distributes *X(3)* and *X(4)* to *P1* and *P2* by using collective I/O method. After finishing the first loop, `write2` is executed to write only data in *writing back* area back to the file. The range of *writing back* area is determined in `write2`. `S/w caching` step modifies schedule information for executing the second loop. Also, Figure 1 shows the change in the data domain of each processor after executing `s/w caching` step. In second phase, `read2`, only data in *new reading* area is read from file. Finally, all data which is referred in the second loop is written back to the file in `write1`.

4 Collective I/O Operations with Chunking and Compression

To further improve performance by reducing both execution time and storage space, our library provides chunking and in-memory compression. To provide a chunked scheme, the global array is organized as an array of chunks [3]. As the collective I/O library reorders the data to achieve a *row major* order, our chunking scheme follows this access pattern to describe the data. The correspondence between chunks and data array is established using a *chunk index*, composed of fixed-length records, each including a chunk identification, length, file offset, compression algorithm used, compressed length of the chunk, and portion of the array corresponding to the chunk. Even when several publicly available lossless compression algorithms were studied, *lzrw3* was chosen because it had the better time results.

Compression and I/O part are clearly decoupled in our library, so that compression, communication, and I/O operations can be overlapped using non-blocking operations. Decoupling both phases is important, because when writing a global array using compressed chunks, each processor can not compute in advance the offset of each chunk, needing communication with the previous writer to know the offset of the previous chunk. In the Parallel Collective Write operations, a non-blocking receive operation is set to get the new offset before compressing the data collected. Then the new offset is actually received and the compressed buffer is written to the file, setting a new entry in the local index vector to store the chunk features. As a compressed chunk does not fit exactly on file system blocks, the new offset is aligned to the next file system block. When all processors have written their portion of data, they asynchronously send the local indexes to the first processor, which generates a global index and writes it to the index file in a single write operation. In case of reading when the data file is opened, the associated index file is opened and read by the first processor, which generates the global index and distributes it to the other processors. Then each processor can get the offset into the file and the compression algorithm used for each chunk, read the compressed chunk, uncompress it, and distribute it to other processors. In the Parallel Pipelined Collective I/O the communication to get the offset is executed on a per-group basis, so that each processor waits for the offset only in its own group. It should be noted that several values associated with the compression operations are very small (the file offset are only 4 bytes and the local index are 48 bytes) to influence significantly the total execution time.

5 Performance Evaluation

The evaluation tests of the runtime system, with and without compression, were executed on TREX, an Intel PARAGON machine located at Caltech, and on the ASCI/Red machine, located at Sandia National Labs. The Intel Paragon TREX is a 550 node Paragon XP/S with 448 GP (with 32 MB memory), 86 MP (with 64 MB memory) nodes, and 16 or 64 I/O nodes with 64 MB memory and a 4 GB Seagate disk. Intel's TFLOPS (ASCI/Red) is a 4500+ compute nodes and 19 I/O nodes machine [4]. Each I/O node also has two 200 MHz Pentium Pro and 256 MB, hav-

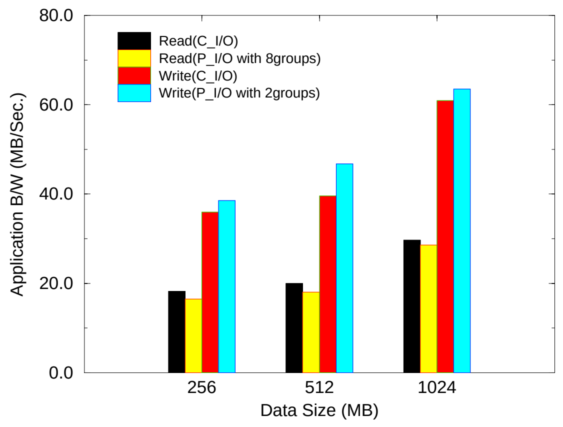

Figure 2. Application B/W for Collective and Pipelined Collective I/O on the Intel Paragon.

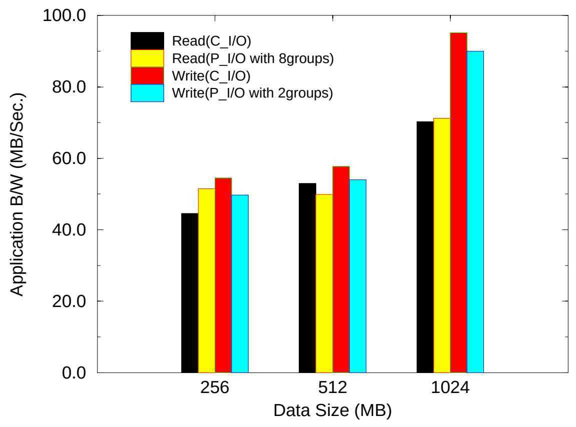

Figure 3. Application B/W for Collective and Pipelined Collective I/O on the ASCI/Red.

ing access to two 32 GB RAIDs. The filesystem used in both machines is Intel PFS.

First set of results show the overall I/O bandwidth without using compression which is observed at the application level, and take into account communication, computation overhead and I/O operations. Figure 2 shows performance results by varying the total size of data and number of processors on a 64 I/O node partition with a 512 KB stripe unit. To extend data size from 256MB to 512MB, we increased the number of processors from 64 to 128 with a data domain size of 4MB per processor. As the data size is increased from 256MB to 512MB, we noticed slight performance improvement in most cases because as the number of processors increases, it requires more communication steps to distribute/collect data to/from other processors even though the size of index and data values to be communicated becomes smaller, degrading the overall performance. Also, with the same number of processors, we increased the size of data from 512MB to 1024MB by using multiple data set (32bytes in this experiment) referenced by each index value. This extends the data domain of each processor from 4MB to 8MB, increasing the I/O cost of each processor. The performance obtained was better because the index propagation and other overheads are amortized over larger data elements. Figure 3 shows the performance results obtained on the ASCI/Red machine by varying the size of data and number of processors. We were able to obtain up to 90 MByte/sec application level I/O bandwidth, which is 50% of the peak performance. In both tests, the number of processor groups in the Pipelined Collective I/O was 2 for writing and 8 for reading.

To evaluate software caching performance, we compared the *software caching method* (S/W) to the basic read/write collective I/O (NC) implementation, called *non-caching method*. Figure 4 represents the execution time for the two implementations when we increase the size of indirection array in each processor's local memory from 4KB to 1MB, which results in the increment in data size from 256KB to 64MB. These results show that total execution time is reduced when software caching is applied, assuming 32 processors, 16 I/O nodes partition, and 7/8 of overlap range between irregular loops. The time for `READ1` and `WRITE1` phases involving read and write operations for data referenced from total data domain is almost the same in both implementations. However, since software caching method does not access overlapped data, the time for partially reading and writing data is much smaller than the time for reading and writing all data including overlapped area. As a result, the overall execution time in software caching method is much smaller than in non-caching method, although the additional time to modify schedule information is required in `S/W CACHING` phase.

To evaluate the compression optimizations, besides the parameters used to evaluate the two models, the compression ratio and the compression time were considered. Figures 5 shows the overall bandwidth of the write operation of the compression library for both the collective (a) and pipelined collective (b) I/Os, using different compression ratios and data sizes. They were obtained by varying the compression ratio, total size of data, and number of processors with a 4 MB buffer each on a 16 I/O nodes partition with a stripe unit of 64 KBytes. The bandwidth of the compressed version is always higher than the original one (ratio 1:1). For a data size of 1 GBytes with a compression ratio of 8:1, the overall bandwidth has been enhanced by almost 200% in both models. Considering the reduction of disk space associated with compression, it can be concluded that it is always beneficial to use compression. As expected,

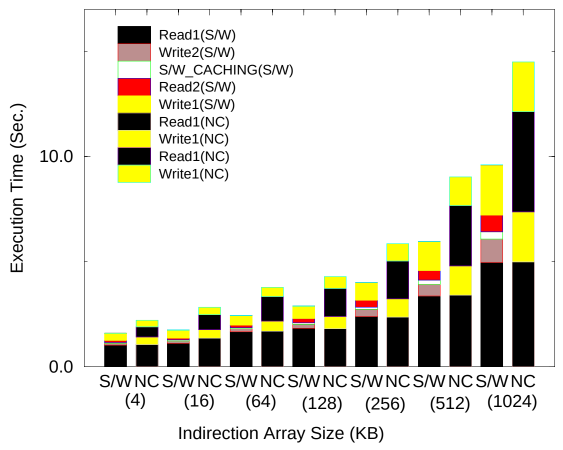

Figure 4. Execution time for S/W caching method.

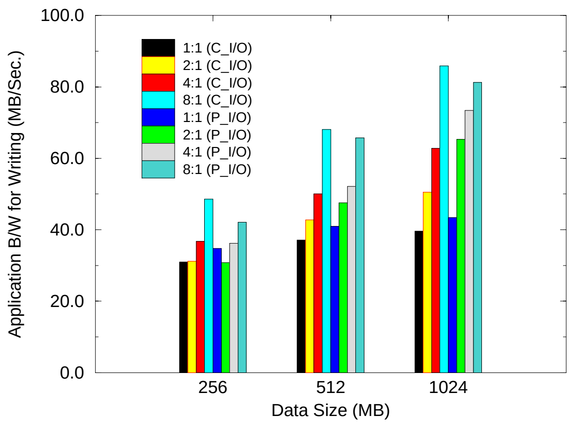

Figure 5. Application B/W with Compression on the Intel Paragon.

there are few differences between both models, although the pipelined collective I/O provides better results for low compression ratios. The higher the compression ratio, the lower the I/O time, and the lower the benefits derived from staggering I/O. The overhead such as local index management, global index management, and index file writing, is lower than 200 msec.

6 Summary and Conclusions

In this paper we presented a design of a runtime I/O library for large-scale irregular problems. The main idea is to reorder data on the fly so that it is sorted before it is stored and reorganized before it is provided to processors for read operations. This eliminates expensive post processing for both visualization as well as enables restarts on different number of processors by eliminating dependence of the checkpointed data on the number of processors that create it. We presented two designs for collective I/O for read/write operations. First design involves all processors simultaneously in an I/O operation, while in the second case, I/O is pipelined with communication. Experimental results show that good performance can be obtained on a large number of processors. Both models have been optimized with a software caching method, chunking, and compression techniques. The performance results obtained on both Intel Paragon at Caltech and Sandia National Lab's ASCI/Red machine show that we were able to obtain application level bandwidth of up to 110MB/sec, even when clear tradeoffs exist for different parameters such as buffer space, number of processor groups, stripe unit, etc. On the Intel Paragon, we were able to duplicate the application level bandwidth for a 8:1 compression ratio using the *lzrw3* compression algorithm, reducing the communication time by almost 50%. There are many performance improvements possible, such as enhancing prefetching using the chunking meta-data, using different file models, or using alternative file layouts, that we intend to develop in the future.

References

[1] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, Portland, OR, 1993. IEEE Computer Society Press.

[2] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, and Rajeev Thakur. PASSION: parallel and scalable software for input-output. Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[3] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Extensible file systems (ELFS): An object-oriented approach to high performance file I/O. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 191–204, Portland, OR, October 1994. ACM Press.

[4] T. Mattson and G. Henry. The asci option red supercomputer. In *Intel Supercomputer Users Group. Thirteenth Annual Conference*, Albuquerque, USA, June 1997.

[5] James T. Poole. Preliminary survey of I/O intensive applications. Technical Report CCSF-38, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.