Techniques to provide Run-Time Support for solving Irregular Problems *

Jaechun No
jno@ece.nwu.edu
Electrical Engineering and Computer Science
Syracuse University

Alok Choudhary
choudhar@ece.nwu.edu
Electrical and Computer Engineering
Northwestern University

Abstract

In this paper we present a runtime library design based on the two-phase collective I/O technique for irregular applications. The design is motivated by the requirements of a large number of ASCI (Accelerated Strategic Computing Initiative) applications, although the design and interface is general enough to be used from any irregular applications. We present two designs, namely, "Collective I/O" and "Pipelined Collective I/O". In the first scheme, all processors participate in the I/O at the same time, 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. This reduces the contention at the I/O nodes but requires more complicated scheduling and a possible degradation in communication performance. We obtained up to 40 MBytes/sec. application level performance on the Caltech's Intel Paragon (with 16 IO nodes, each containing one disk) which includes on-the-fly reordering costs. We observed up to 60 MBytes/sec on the ASCI/Red machine with only three I/O nodes (with RAIDS).

1 Introduction

Parallel computers are being used increasingly to solve large computationally intensive as well as data-intensive applications, such as large-scale computations in physics, chemistry, biology, engineering, medicine, and other sciences. Most of these applications are I/O intensive and have tremendous I/O requirements [9, 6]. Applications involve different types of I/O. These include checkpointing of large-scale data sets for restart, periodically writing snapshots of computations for subsequent visualization, and in many cases, data sets are so large that they reside on disk, so called, out-of-core applications.

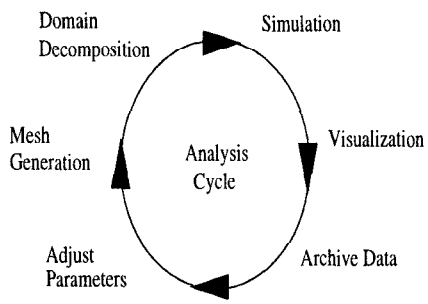

Figure 1: Components of the traditional analysis cycle.

A large number of these applications are "irregular" applications, where accesses to data are performed through one or more level of indirections. Sparse matrix computations, particle codes, and many CFD applications where geometries and meshes are described via indirections, exhibit these characteristics. In particular, the predominantly unstructured-grid simulation applications on the ASCI (Accelerated Strategic Computing Initiative) platforms all need to read/write data according to some preset global ordering specified by the indirection.

There are two reasons for supporting this "global sort" capability within an application: faster visualization and easier restart. In the tradition analysis cycle (Figure 1), a simulation application reads in a mesh and decomposition map and outputs results for visualization. Although simulation applications have become massively parallel, mesh generation and visualization software have not. To circumvent their I/O problems, most applications currently simply output each processor-local mesh, corresponding to a data file

*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.

0-8186-8227-2/97 $10.00 © 1997 IEEE

Proceedings of the 1997 International Conference on Parallel and Distributed Systems (ICPADS '97)
0-8186-8227-2/97 $10.00 © 1997 IEEE

per processor, and rely on some sequential postprocessor to recombine the data files into a single data file corresponding to the global mesh. As the number of processors scales (e.g., to more than 4500 on the ASCI/Red machine at Sandia National Labs), and the number of grid points being considered moves to the million-to-billion range, sequential recombination of the MBytes/Gbytes of data into the anticipated terabytes of data simply has to be abandoned. Thus, to accommodate the still-sequential nature of existing mesh generation and visualization software, and to avoid the sequential data recombination bottleneck, we need to transfer data that logically correspond to the global mesh by performing the "global sort" within the simulation application. Additionally, even as the rest of the analysis cycle becomes more parallelized, we still need to address the issue of restart with a different number of processors. By always maintaining the global-mesh data in some canonical ordering, we can avoid the explosive number of different file formats and share data among different platforms and processor configurations.

For example, in many ASCI applications, we anticipate transferring between 100K to 2M bytes of data per processor(per variable) during the I/O phase of a 500- to 4000-processor job. To reach the desired, sustainable 1 GByte/second I/O bandwidth on ASCI/Red, the parallel file system requires transferring of large blocks of data on the compute nodes, with stripe unit being 512 Kbyte. ASCI/Red provides 18 I/O nodes to service the I/O requests of approximately 4500 compute nodes. Clearly, we cannot simply rely on the traditional method of "staging" the I/O requests whereby only a group of processors performs I/O while others idle, because the data for each processor is in general noncontiguous within the global data file, and this will generate large amounts of small data requests. To enable large data block transfers, some more sophisticated form of collective I/O is needed.

In this paper we present the design and implementation of a high-performance runtime system for 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; that is, data sets that are accessed via indirections. The data is reorganized (sorted) on the fly to eliminate postprocessing time. In this method, the requesting processors cooperate in reading or writing data—a process known as collective I/O, first proposed in general [2, 4, 5, 10]. Specifically, processors cooperate to combine several I/O requests into fewer larger granularity requests, reorder requests so that the file is accessed in proper sequence, and eliminate simultaneous I/O requests for the same data. In addition, I/O workload is partitioned among processors dynamically in a balanced fashion.

The rest of the paper is organized as follows. In Section 2, we present the problem description and design considerations for parallel I/O library to support irregular computations and on-the-fly reorganization of data. Section 3 presents two collective I/O techniques. Section 4 presents performance results and various tradeoffs observed when evaluated on large systems; specifically on the Intel Paragon machine at Caltech and Sandia National Labs. Section 5 contains summary and conclusions.

2 Design Considerations

2.1 Preliminaries

In this paper, computations in which data accesses are performed through a level of indirection are considered irregular computations. Many scientific applications make extensive use of indirection arrays. Examples include Computational Fluid Dynamics Codes (CFD), Particle Codes, Finite Element Codes etc. The domain of these problems is normally irregular describing, for example, a physical structure, which is discretized. Figure 2 illustrates an irregular loop [3, 7]. This example shows the code that sweeps over *nedge* mesh edges. Arrays x and y are *data arrays*. Loop iteration *i* carries out a computation involving the edge that connects vertices *edge1(i)* and *edge2(i)*. Arrays such as *edge1* and *edge2* which are used to index data arrays are called *indirection arrays*.

```
do n = 1, n_step
...
  do i = 1, nedge
    x(edge1(i)) = x(edge1(i)) + y(edge2(i))
    x(edge2(i)) = x(edge2(i)) + y(edge1(i))
  end do
...
end do
```

Figure 2: An Example with an Irregular Loop.

2.2 Abstraction

This problem can be abstracted into the one shown in Figure 3. We assume that data is distributed using some partitioning scheme (which may be application dependent). On each node, there is an indirection array which describes the location of the corresponding data element in a global array. For example, in Figure 3, processor 0 accesses *D(I0), D(I4), D(I5), D(I2)*, processor 1 accesses *D(I7), D(I8), D(I6), D(I9)*, etc.

Proceedings of the 1997 International Conference on Parallel and Distributed Systems (ICPADS '97)
0-8186-8227-2/97 $10.00 © 1997 IEEE

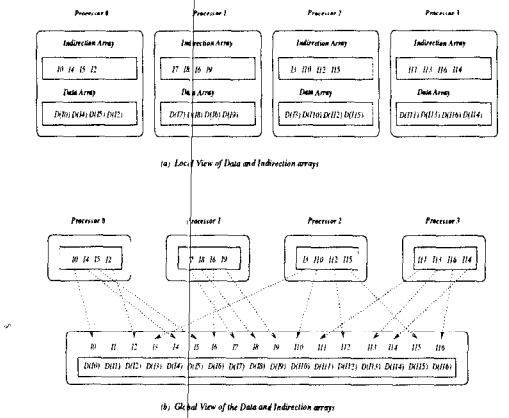

Figure 3: Local and Global View of Data and Indirection Arrays

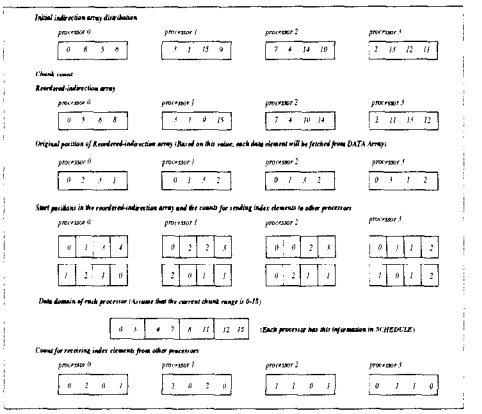

Figure 4: Schedule construction

The objective of the I/O library is to read data from files or write data into files using a high-level interface in the order imposed by the global array. There are several characteristics to be considered to develop an I/O library to support irregular problems [8]. One characteristic of irregular problems is that it often generates fine-grained data distribution requiring access to non-contiguous locations in global array. Therefore, appropriate collective I/O method is necessary to obtain high I/O performance. Another characteristic is that irregular problems often generate small messages. This communication pattern results in poor performance on message-passing machines favoring large-sized messages.

The design of the collective I/O library function is based on the two-phase I/O strategies [2, 10]. The basic idea behind two-phase collective I/O is at runtime to reorder the access patterns seen by the I/O system such that the patterns are optimized. In other words, large number of small and disjoint I/O requests are converted into small number and large contiguous requests. This optimization incurs costs in terms of additional communication and buffer space requirements. However, since communication speed is normally several order of magnitude faster than the I/O speeds, the additional cost is much smaller than the reduction in the I/O cost. Several factors must be considered in the design of a library based on this technique. These include 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, for irregular computations, since data accesses are performed using a level of indirection, the number of passes through the data sets to compute schedule and reorganize data also are important. In the following sections, we present two design alternatives for collective I/O operations.

3 Design of Collective I/O Operations

In the following two subsections, 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"*. The rationale for the two designs and the differences will become clear in the following.

3.1 Collective I/O

This design involves 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. Each of these steps consists of several phases, which are described below.

3.1.1 Schedule Construction

Schedule describes the communication pattern and I/O required for each node participating in the system. For regular multidimensional arrays, the accesses can be described using regular section descriptors. Access and communication schedule can be built based upon just this information. On the other hand, when indirection arrays are involved to reference data array, the indirection arrays must be scanned to consider each element of the array individually to determine its place in the global array as well as its destination processor for communication.

Figure 4 illustrates the information stored in schedule information. Several factors affect the schedule construction in particular, and overall I/O library design and performance in general. These include the following. 1) *Chunk Size,* which is the amount of

Proceedings of the 1997 International Conference on Parallel and Distributed Systems (ICPADS '97)
0-8186-8227-2/97 $10.00 © 1997 IEEE

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. Figure 4 briefly describes the steps involved in computing schedule information.

- Based upon the chunk size, each processor is assigned a data domain for which it is responsible for reading or writing. For example, if there are four processors and total 16 elements to be read/written with a buffer space of two elements on each processor, the chunk size(total) is 8 elements. Processor 0 will be responsible for elements 0,1 and 8,9, processor 1 will be responsible for elements 2,3 and 10,11, and so on. For a chunk, each processor computes its part to read or write data while balancing I/O workload. 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 elements between processors(communication coalescing).

Note that once it is constructed, the schedule information can be used repeatedly in the irregular problems whose access pattern does not change during computation.

3.1.2 Parallel Collective Read/Write Operations

A 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 balanced across processors. Figure 5

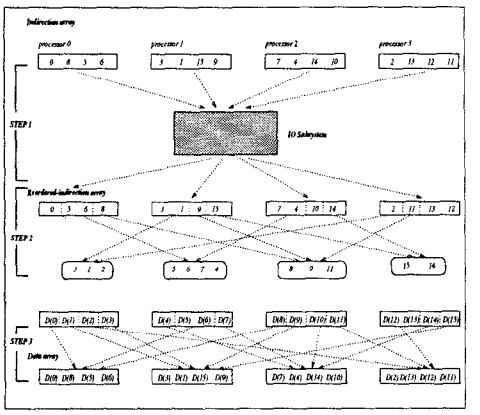

Figure 5: Parallel Collective Read Operation

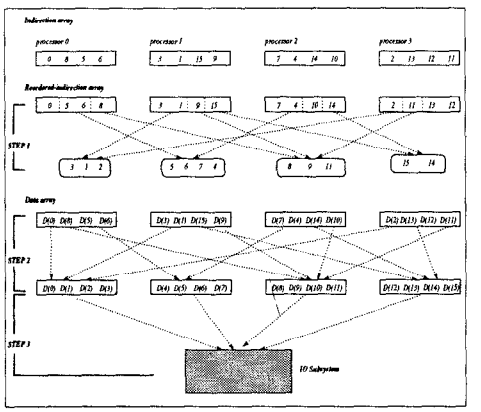

Figure 6: Parallel Collective Write Operation

and Figure 6 represent the overall designs for Parallel Collective I/O.

3.2 Pipelined Collective I/O

In the previous approach, all processors issue I/O requests to the I/O system simultaneously. As a result, contention at the I/O system may occur for a large number of compute nodes. This is particularly possible in a parallel machine where the number of I/O nodes and disks are unbalanced with respect to the number of compute nodes. In the pipelined collective I/O, we divide processors into multiple processor groups. Only processors in a group issue I/O request simultaneously to reduce disk contention. The expectation is that while one group of processors is performing I/O operation, another group performs communication in order to collect(redistribute) data for write(read) operation. Thus, if there are G groups, then 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 operation.

Proceedings of the 1997 International Conference on Parallel and Distributed Systems (ICPADS '97)
0-8186-8227-2/97 $10.00 © 1997 IEEE

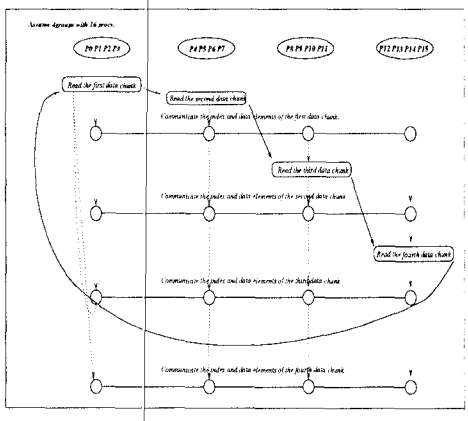

Figure 7: Parallel Pipelined Read Operation

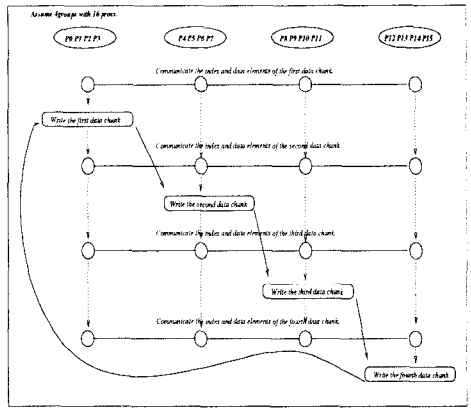

Figure 8: Parallel Pipelined Write Operation

Figures 7 - 8 illustrates the steps involved in the pipelined collective I/O operation. This also permits one group to perform asynchronous I/O while communication for another group is taking place, thereby making it possible to overlap I/O and communication. Consider the illustration shown in Figure 8. Note that schedule for a write operation is constructed as before, except that there is a schedule for each group. From Figure 8, it is clear that there are four distinct steps. Communication is performed for the first group to collect data in its data domain. Then processors in the first group issue write requests to the I/O system. While this write is being performed, communication for the second group takes place. This process continues until all groups have performed their write operation. This entire process repeats if the chunk size is less than the amount of the data to be read/written. For example, if the amount of data to be written is 1Mbyte per processor, and if the buffer space available to each processor is only 0.5Mbyte, then two iterations of the group write operation will be performed. We discuss these tradeoffs in the next section where we present performance results.

4 Performance Evaluation

We ran our experiments on the Intel Paragon machine at Caltech called *TREX*. *TREX* is a 550 node Paragon XP/S with 448 GP (with 32MB memory), 86 MP (with 64MB memory) nodes [1]. We performed our experiments on 64 and 128 compute nodes. The partition uses 16 I/O servers, each with 64MB memory and a 4GB Seagate disk, with listed 9MB/sec peak speed. The parameters considered for performance evaluation include a) number of processors, b) chunk size, which represents the amount of buffer space available to the runtime library, c) stripe unit, representing the amount of logically contiguous data

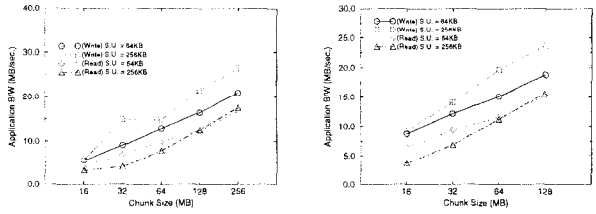

Figure 9: Application B/W for Collective I/O Operations. The first graph shows the application B/W with 128 compute nodes on 512MB of data size. The second graph shows the application B/W with 64 compute nodes on 256MB of data size. Each processor has 2MB of indirection array in its local memory. The stripe units (S.U.) are 64KB and 256KB respectively.

read/written from each I/O node, and d) number of processor groups (for the pipelined collective I/O implementation).

4.1 Overall I/O Bandwidth

First set of results show the overall I/O bandwidth observed at the application level. In other words, it is a measure of the duration of time from the library function call until its completion. Note that when data is written, it is globally sorted before actually being stored, and when it is read, it is reordered and redistributed according to the distribution specified by the indirection arrays. That is, the bandwidth measure takes into account communication, computation overhead and the actual I/O operation.

Figure 9 shows results by varying different parameters for the collective I/O operations. Clearly, as expected, the observed bandwidth increases as the chunk size is increased because larger buffers are used. Stripe size of 256K consistently produces better performance for write operations, and it performs better for read

Proceedings of the 1997 International Conference on Parallel and Distributed Systems (ICPADS '97)
0-8186-8227-2/97 $10.00 © 1997 IEEE

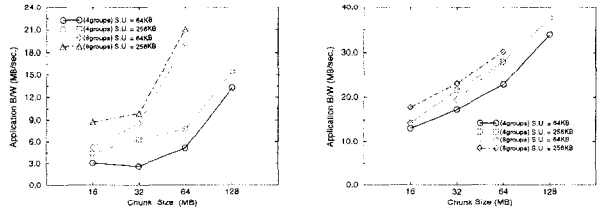

Figure 10: Application B/W for Pipelined Collective I/O Operations. The first graph shows the application B/W of Pipelined Collective Read. The second graph shows the application B/W of Pipelined Collective Write. The total data size is 512MB with 128 compute nodes. Each processor has 2MB of indirection array in its local memory. The stripe units (S.U.) are 64KB and 256KB respectively.

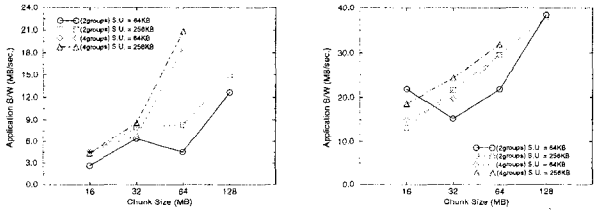

Figure 11: Application B/W for Pipelined Collective I/O Operations. The first graph shows the application B/W of Pipelined Collective Read. The second graph shows the application B/W of Pipelined Collective Write. The total data size is 256MB with 64 compute nodes. Each processor has 2MB of indirection array in its local memory. The stripe units (S.U.) are 64KB and 256KB, respectively.

operations when the chunk size is large. This is observed for both 64 and 128 processor cases. Through various experiments we determined that 256K stripe unit provided better results than those for smaller and larger stripe units (of up to 1M) on this machine. In the latter case, concurrency in I/O is reduced and communication messages become too large.

Figure 10 shows the corresponding results for the pipelined collective I/O implementation. It should be noted that in order to fairly compare two different group size's performance, buffer space per node was kept constant so that the overall buffer space available to the library function remains the same. Figure 10 shows performance results for number of groups 4 (group size 32 processors) and 8 (group size 16 processors). Comparing Figures 9 and 10, it is clear that 4 groups with 256K stripe unit performance the best for write operations, giving almost 40 MB/sec application level bandwidth. Furthermore, it can be observed that there exists a tradeoff in the number of processors per group. Figure 11 compares 2 and 4 processor groups for 64 processor configuration, and similar tradeoffs exist as before.

Figure 12 shows that there is a tradeoff in varying the number of processor groups and the number of processors per group. It also shows that optimal points may exist for best performance, but we believe more experiments need to be performed to determine them.

4.2 Performance of the Components

So far we considered the overall performance using the observed I/O bandwidth as the measure (which is the most important from a user's perspective in addition to the ease of use). From the design perspective,

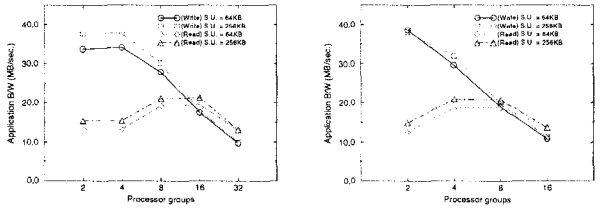

Figure 12: Application B/W for Pipelined Collective I/O Operations as a function of processor groups. The first group shows the application B/W of Pipelined Collective I/O with 128 compute nodes on the 512MB of data size. The second group shows the application B/W of Pipelined Collective I/O with 64 compute nodes on the 256MB of data size. Each processor has 4MB of chunk size and 2MB of indirection array. The stripe units (S.U.) are 64KB and 256KB respectively.

the behavior of the components, such as I/O operations, communication overhead for sort and reorganization, and computation overhead for reordering data and housekeeping operations are important. These provide insight into possible improvement in each component's performance.

Figures 13 - 14 show a subset of performance results for different components for collective I/O and pipelined collective I/O operations, respectively. From Figure 9 we earlier observed that for a write operation, large chunk size always results in better I/O and communication performance, where as for a read operation that is true for the most part except for those cases where the chunk size is large. Large chunk size results in comparatively poorer performance for read operations because messages become very large and

Proceedings of the 1997 International Conference on Parallel and Distributed Systems (ICPADS '97)
0-8186-8227-2/97 $10.00 © 1997 IEEE

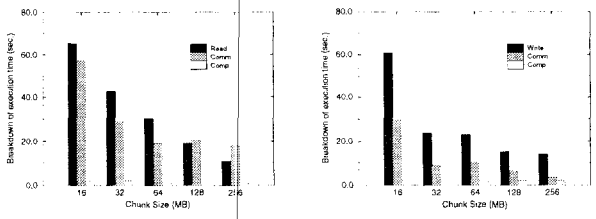

Figure 13: Breakdown of total execution time (in sec.) for Collective I/O Operations as a function of chunk size. The first graph shows the breakdown of execution time of Collective Read with 64KB of stripe unit. and the second graph shows the breakdown of execution time of Collective Write with 256KB of stripe unit. Total data size is 512MB and the number of processors is 128.

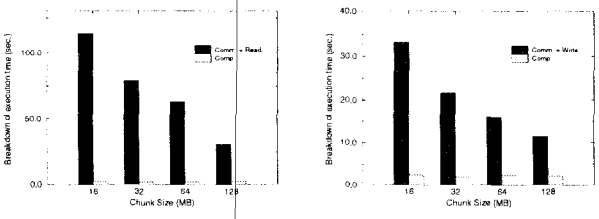

Figure 14: Breakdown of total execution time (in sec.) for Pipelined Collective I/O Operations as a function of chunk size. The first graph shows the breakdown of execution time of Pipelined Collective Read with 4 processor groups and and the second graph shows the breakdown of execution time of Pipelined Collective Write with 4 processor groups. Total data size is 512MB and the number of processors is 128. The stripe unit is 256KB.

get blocked in the network. In the future, we plan to strip-mine communication to determine if communication overhead can be further reduced. Also, it is easier to improve communication performance because networks are becoming faster (compared to the corresponding improvement in the I/O systems). In our experiments, the data distribution did not have locality, that is, the data was uniformly and randomly distributed across processors. In general, applications use mappers using which data is distributed such that data locality is enhanced. That will reduce communication requirements because a processor will communicate only with a subset of other processors reducing communication traffic. In both cases, it is clear that performance improvements can be obtained by reducing both communication and I/O times.

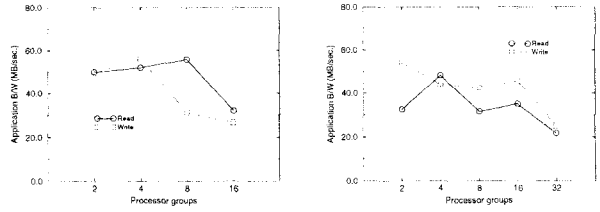

Figure 15: Application B/W for Pipelined Collective I/O Operations based on the processor groups on ASCI/Red. The first graph shows the application B/W of Pipelined Collective I/O with 64 compute nodes on 256MB of data size. The second graph shows the application B/W of Pipelined Collective I/O with 128 compute nodes on 512MB of data size. The stripe unit (S.U.) is 512KB.

4.3 Performance on the ASCI/Red Machine

ASCI/Red machine is a Teraflop machine being deployed at Sandia National Labs (a description was provided earlier). Currently, only three I/O nodes[1] are available, but each I/O node employs High-Performance RAID systems (unlike the Caltech paragon system used for earlier experiments where each I/O node has a disk).

Figure 15 summarizes the application level bandwidth observed for the *pipelined collective I/O* implementation. Again, bandwidth of up to 60MBytes/sec on three I/O nodes is observed. Also, the figure clearly illustrates than an optimal range of processor groups exists which provides the best performance. As the machine is upgraded with larger number of I/O nodes, we plan to conduct more experiments and present those results in the final version.

5 Summary and Conclusions

In this paper we presented a design of a runtime I/O library for large-scale irregular problems, such as those found in ASCI and other applications. 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 pre-

[1] In the next several months, ASCI/Red machine will be fully developed with improvements in the file system as well as the addition of new I/O nodes. We will evaluate the performances on a large configuration and present these results in the final version.

Proceedings of the 1997 International Conference on Parallel and Distributed Systems (ICPADS '97)
0-8186-8227-2/97 $10.00 © 1997 IEEE

sented 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. Preliminary results show that good performance can be obtained on a large number of processors. Clear tradeoffs exist for different parameters including buffer space, number of processor groups, stripe unit etc. We presented performance results on both Intel Paragon at Caltech and Sandia National Lab's ASCI/Red machines. We were able to obtain application level bandwidth of up to 60MB/sec. There are many performance improvements possible, which we intend to develop in the future.

References

[1] Rajesh Bordawekar. Implementation and evaluation of collective i/o in the intel paragon parallel file system. Technical report, California Institute of Technology, November 1996.

[2] 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, 1993.

[3] Peter Brezany and Alok Choudhary. Techniques and optimizations for developing irregular out-of-core applications on distributed-memory systems. Technical report, Institute for Software Technology and Parallel Systems, University of Vienna, November 1996.

[4] 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.

[5] Alok Choudhary, Rajesh Bordawekar, Sachin More, K. Sivaram, and Rajeev Thakur. PASSION runtime library for the Intel Paragon. In *Proceedings of the Intel Supercomputer User's Group Conference*, June 1995.

[6] Juan Miguel del Rosario and Alok Choudhary. High performance I/O for parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[7] R. Ponnusamy, Y.-S. Hwang, R. Das, J. Saltz, A. Choudhary, and G. Fox. Supporting irregular distributions in FORTRAN 90D/HPF compliers. Technical report, University of Maryland, Syracuse University, Spring 1995.

[8] R. Ponnusamy, J. Saltz, A. Choudhary, Y.-S. Hwang, and G. Fox. Runtime-compilation techniques for data partitioning and communication schedule reuse. In *Proc. of Supercomputing'93*, Portland, OR., November 1993.

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

[10] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kudatipidi. Passion: Optimized i/o for parallel systems. *IEEE Computer*, June 1996.

Proceedings of the 1997 International Conference on Parallel and Distributed Systems (ICPADS '97)
0-8186-8227-2/97 $10.00 © 1997 IEEE