A Scientific Data Management System for Irregular Applications*

Jaechun No† *Rajeev Thakur*† *Dinesh Kaushik*† *Lori Freitag*† *Alok Choudhary*‡

†Math. and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
{jano,thakur,kaushik,freitag}@mcs.anl.gov

‡Dept. of Electrical and Computer Eng.
Northwestern University
Evanston, IL 60208
choudhar@ece.nwu.edu

Abstract

Many scientific applications are I/O intensive and generate or access large data sets, spanning hundreds or thousands of "files." Management, storage, efficient access, and analysis of this data present an extremely challenging task. We have developed a software system, called Scientific Data Manager (SDM), that uses a combination of parallel file I/O and database support for high-performance scientific data management. SDM provides a high-level API to the user and, internally, uses a parallel file system to store real data and a database to store application-related metadata.

In this paper, we describe how we designed and implemented SDM to support irregular applications. SDM can efficiently handle the reading and writing of data in an irregular mesh as well as the distribution of index values. We describe the SDM user interface and how we implemented it to achieve high performance. SDM makes extensive use of MPI-IO's noncontiguous collective I/O functions. SDM also uses the concept of a *history file* to optimize the cost of the index distribution using the metadata stored in the database. We present performance results with two irregular applications, a CFD code called FUN3D and a Rayleigh-Taylor instability code, on the SGI Origin2000 at Argonne National Laboratory.

*This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and in part by a Work-for-Others Subaward No. 751 with the University of Illinois, under NSF Cooperative Agreement #ACI-9619019.

1 Introduction

Many large-scale scientific applications are I/O intensive and generate large amounts of data (on the order of several hundred gigabytes to terabytes) [8, 24]. Many of these applications perform their computation and I/O on an irregularly discretized (and distributed) mesh. The data accesses in those applications make extensive use of arrays, called indirection arrays [7, 23] or map arrays [11], in which each value of the array denotes the corresponding data position in memory or in the file. In irregular applications, data can be distributed either by using compiler directives with the support of runtime preprocessing [9, 33] or by using a runtime library [7, 23]. Most of the previous work in the area of unstructured-grid applications focuses on computation and communication in such applications, rather than I/O.

We have developed a software system for large-scale scientific data management, called Scientific Data Manager (SDM) [22]. SDM aims to combine the good features of both file I/O and databases. SDM provides a high-level, user-friendly interface. Internally, SDM interacts with a database to store application-related metadata and uses MPI-IO to store the real data on a high-performance parallel file system. SDM takes advantage of various I/O optimizations available in MPI-IO, such as collective I/O and noncontiguous requests, in a manner that is transparent to the user. As a result, users can access data with the performance of parallel file I/O, without having to bother with the details of file I/O.

In a previous paper [22], we described the use of SDM for regular applications. In this paper, we describe the API, design, and implementation of SDM for irregular applications. SDM can efficiently handle the reading and writing of data in an irregular mesh as well as the distribution of index values. SDM also uses the concept of a *history file* to optimize the cost of the index distribution using the metadata stored in database. We present performance results with two irregular applications, a CFD code called FUN3D and a Rayleigh-Taylor instability code, on the SGI Origin2000 at Argonne National Laboratory.

The rest of this paper is organized as follows. In Section 2 we discuss our goals in developing SDM for irregular applications. In Section 3 we describe, with the help of an example, the SDM API and how it is implemented. Performance results are presented in Section 4. We discuss related work in Section 5 and conclude in Section 6 with a brief discussion of future research.

2 Design Objectives

Our main objectives in designing SDM for irregular applications were to achieve high-performance parallel I/O, to provide a convenient high-level API, and to optimize the execution cost of irregular applications.

- **High-Performance I/O**. To achieve high-performance I/O, we decided to use a parallel file-I/O system to store real data and use MPI-IO to access this data. MPI-IO, the I/O interface defined as part of the MPI-2 standard [11, 18], is rapidly emerging as the standard, portable API for I/O in parallel applications. MPI-IO is specifically designed to enable the optimizations that are critical for high-performance parallel I/O. Examples of these optimizations include collective I/O, the ability to access noncontiguous data sets, the ability to pass hints to the implementation about access patterns, and file-striping parameters.

- **High-Level API**. Our goal was to provide a high-level unified API for any kind of application (regular or irregular) while encapsulating the details of either MPI-IO or databases. The user can specify the data with a high-level description, together with annotations, and use a similar API for data retrieval. SDM internally translates the user's request into appropriate MPI-IO calls, including creating MPI derived datatypes for noncontiguous data [31]. SDM also interacts with the database when necessary, by using embedded SQL functions.

- **Optimization for Irregular Applications**. In irregular applications, the cost of an index distribution is usually expensive, in terms of communication and computation volumes. In SDM, after the index values are partitioned among processes, the local index subsets of all processes are asynchronously written to a history file and the associated metadata is stored in the database. When the same index distribution is needed in subsequent runs, the index values are read from the history file using the metadata stored in the database, thereby avoiding communication and computation for the same index distribution.

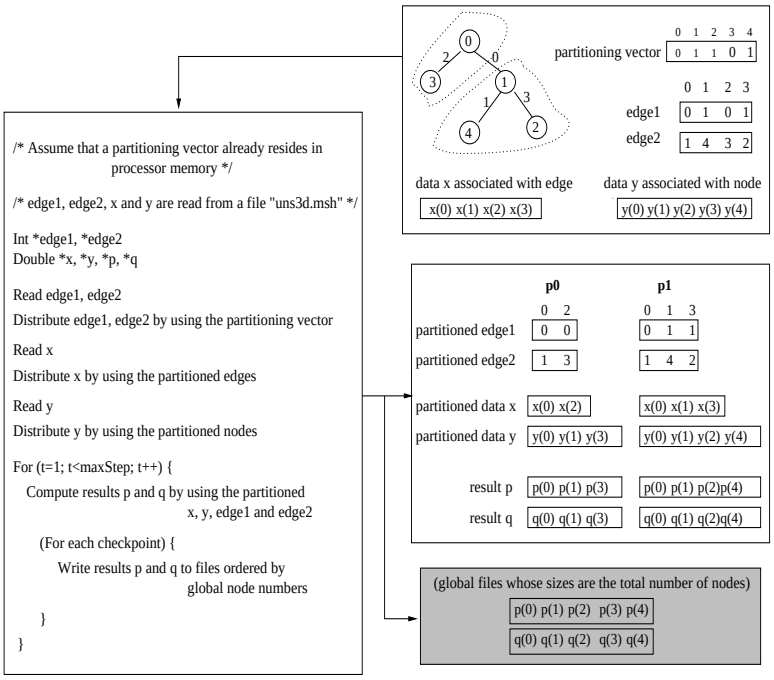

Figure 1: A sample irregular problem and its solution

3 Implementation

With the help of a sample irregular problem, we discuss the SDM API for solving the problem and how the API is implemented.

3.1 An Irregular Problem and SDM API

Figure 1 shows an example of a typical irregular problem that sweeps over the edges of an irregular mesh. In this problem, `edge1` and `edge2` are two arrays representing nodes connected by an edge, and arrays `x` and `y` are the actual data associated with each edge and node, respectively. The partitioned arrays of `edge1`, `edge2`, `x`, and `y` contain a single level of "ghost data" beyond the boundaries to minimize remote accesses. After the computation is completed, the results `p` and `q` are written to a file in the order of global node numbers.

Figures 2(a) and 2(b) show the SDM API for writing the results `p` and `q` and for partitioning `edge1`, `edge2`, `x`, and `y` among processes to solve the problem presented in Figure 1. We use the term *import* to distinguish it from a *read* operation. A read operation reads the data created in SDM, whereas an import operation reads the data created outside SDM.

3.2 Implementation Details

The *partitioning vector* is a vector generated from a graph-partitioning tool, such as MeTis[25, 14]. Each value of the vector denotes the rank of the process to which the node is assigned. The *map array* specifies the mapping of each element of the local array to the global array. This map array is created in SDM after partitioning the indexes using a partitioning vector, or it can be specified by the user.

Figure 2(a) shows the steps involved in initializing SDM. The function *SDM_initialize* is called to establish a database connection (for storing metadata). The data arrays to be written to files as a result of simulation, `p` and `q`, are grouped in *SDM_make_datalist*, and the associated metadata is stored in the database by calling *SDM_set_attributes*.

Figure 2(b) describes the steps in SDM to import and partition the indices and data. The four arrays, `edge1`, `edge2`, `x`, and `y`, are grouped to be imported into SDM, and the metadata is stored in the database by *SDM_make_importlist*.

In order to partition `edge1` and `edge2`, *SDM_import* is called to import the arrays contiguously. In our example, edges `0` and `1` are imported to process 0, and edges `2` and `3` are imported to process 1.

If there is a history file for this problem size, the function *SDM_partition_index* reads the already partitioned `edge1` and `edge2` from the history file and converts them to localized edges by using the partitioning vector. This

```
SDM_initialize (nameOfApplication);
result = SDM_make_datalist (2, {p,q});
result[0].data_type = DOUBLE;
SDM_associate_attributes (2, &result[0]);
handle = SDM_set_attributes (2, result);

.....
/* Partition edge1, edge2, x and y among processes */
.....
SDM_data_view (handle, 2, p, &vector, &localNodes);
For (t=0; t<maxStep; t++) {

    Do Computation and produce results p and q;
    For (each checkpoint) {

        SDM_write(handle, p, t, pBuf);
        SDM_write(handle, q, t, qBuf);

    }
}
SDM_finalize (handle, 2);
```

(a)

```
import = SDM_make_datalist (4, {edge1, edge2, x, y});
import[2].data_type = DOUBLE;
SDM_associate_attributes(2, &import[2]);
SDM_make_importlist(handle, 4, import);


SDM_import(handle, edge1, 0, totalEdges, tmp);
SDM_import(handle, edge2, (totalEdges*sizeof(int)), totalEdges,
                                        tmp+(totalEdges*sizeof(int)));


/* Distribute edge1 and edge2 among processes */
vector = SDM_partition_table (handle, partitioning_vector, totalNodes);
partitioned_edge = SDM_partition_index(handle, partitioning_vector,
                                totalNodes, &tmp, &vector);

localEdges = SDM_partition_index_size(handle);
localNodes = SDM_partition_data_size(handle);


/* Make a history of this index distribution */
SDM_index_registry(handle, partitioned_edge, &localEdges);

/* Import x*/
file_offset = 2*totalEdges*sizeof(int);
SDM_data_view(handle, 1, x, &partitioned_edge, &localEdges);
SDM_import(handle, x, file_offset, totalEdges, xBuf);

/* Import y */
file_offset += totalEdges*sizeof(double);
SDM_data_view(handle, 1, y, &vector, &localNodes);
SDM_import(handle, y, file_offset, totalNodes, yBuf);

SDM_release_importlist(handle, 4);
```

(b)

Figure 2: SDM API: (a) for writing results and (b) for partitioning indices and data

approach avoids the communication cost to exchange each process's edges and the computation cost to choose the edges to be assigned. The disadvantage of the history file is that it cannot be used if the program is run on a different number of processes from when the file was created.

One efficient use of the history file is to create it in advance for the various numbers of processes of interest. As long as the user runs the application with any of those numbers of processes, an appropriate history can be chosen, thereby reducing communication and computation costs. If there is no history file, the edges in each process are distributed by reading all the data in parallel and performing a ring-oriented communication.

If at least one node of an edge has been partitioned to a process, the edge is assigned to that process. For example, edge `0` is assigned to both process 0 and process 1 because one node of the edge has been partitioned to process 0 and the other node has been partitioned to process 1. This edge is a ghost edge for both processes and is stored to minimize communication.

In Figure 2(b), `partitioned_edge` contains the edges assigned to each process, and `vector` contains the nodes assigned to it. These are the two map arrays used to distribute the physical data associated with each edge and node, respectively.

In *SDM_index_registry*, the index distribution is registered to the database, and the partitioned edges are asynchronously written to a history file to be retrieved in subsequent runs requiring the same edge distribution. The use of the *SDM_index_registry* is optional.

In order to import and partition data `x` and `y` in *SDM_import*, *SDM_data_view* needs to be called to define a data mapping between a noncontiguous global view of the file and a local view of the processor memory. In *SDM_import*, the associated data is irregularly distributed using this data mapping by calling a collective MPI-IO function. In *SDM_release_importlist*, the internal data structures used by SDM to import data are freed.

Figure 2(a) shows the steps to write two data arrays, `p` and `q`, after completing the computations at each checkpoint. Before writing `p` and `q`, the data mapping to write is defined in *SDM_data_view* using the map array (`vector`) associated with the node partition.

SDM supports three different ways of organizing data in files. In level 1, each data generated at each time step is written to a separate file. In level 2, each data (within a group) is written to a separate file, but different iterations of the same data are appended to the same file. This method results in a smaller number of files. In level 3, all iterations of all data belonging to a group are stored in a single file. The idea is that if a file system has high file-open and file-close costs, and an application has a high file-view cost (such as in irregular applications), SDM can generate a very small number of files. On the other hand, if an application produces a large amount of data with a large problem size, level-3 file organization would result in very large files, which may degrade performance.

Figure 3 depicts the metadata storage in the database and the organization of data in files in SDM for the example

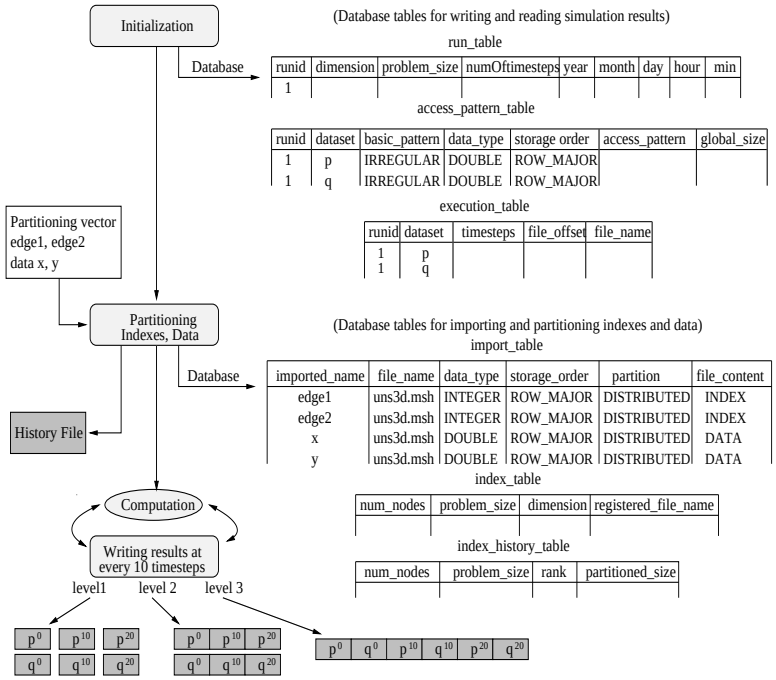

Figure 3: SDM execution flow for solving the example in Figure 1

in Figure 1.

4 Performance Results

We obtained performance results on the SGI Origin2000 at Argonne National Laboratory. The file system on the Origin2000 is XFS [12, 29], and we used MySQL [19] as the database to store metadata. The first application template we benchmarked was a tetrahedral vertex-centered unstructured grid CFD code called FUN3D [1], and the second application template was a Rayleigh-Taylor instability application (RT) [10].

4.1 Results for FUN3D

We used a grid of about 18 million edges and ran the code on 64 processors. Figure 4 shows the bandwidth to import and partition the 18 million edges, four sets each of about 144 Mbytes of data associated with edges, and another four sets of 21 MBytes of data associated with nodes. In the original version of the application—without using SDM—all the I/O operations were performed by a single process (process 0), which then broadcast data to other processes. SDM, on the other hand, performs I/O in parallel from all processes using MPI-IO. The bar labeled `index distri` in Figure 4 shows the communication and computation costs to partition the edges after importing them into the application. The bar labeled `import` shows the cost of reading the edges and eight data arrays.

In the original application, the edges are read in two steps: one step to determine the amount of memory necessary to store the partitioned edges and the other step to actually read the edges. SDM, however, extends the allocated memory dynamically as needed (using the C function `realloc`) and is therefore able to read the partitioned edges in a single step. This contributes to the reduced cost of `index distri` when using SDM. When partitioning the edges with a history file, the cost of `index distri` is nothing but reading the history file of the edges in a contiguous way, including the database cost to access the metadata.

Figure 5 shows the I/O bandwidth for writing and then reading back the data generated from the application using 64 processors. The total data size was approximately 379 MBytes. In level 1, each data array was written to separate files, resulting in the creation of 10 different files. Each time a data array was written in level 1, it incurred the cost of opening a file and defining an MPI-IO file view to access the data from the portion of the file pointed to by the global file offset. In level 2, however, each data array generated at each time step was appended in five files, generating only five file-open and file-view costs. This reduced number of files resulted in a slight improvement in performance. In level 3, only two files were generated, resulting in the best I/O performance among the three file

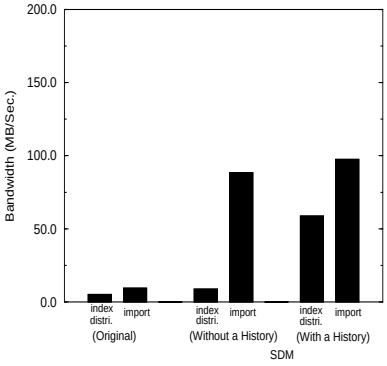

Figure 4: I/O bandwidth for partitioning indices and data in FUN3D

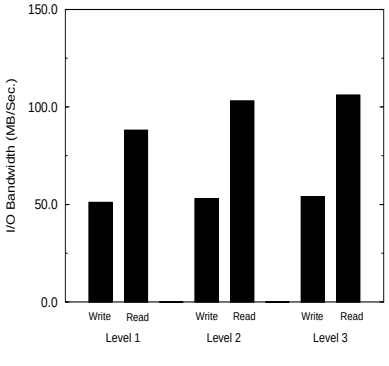

Figure 5: I/O bandwidth for reading and writing data in FUN3D

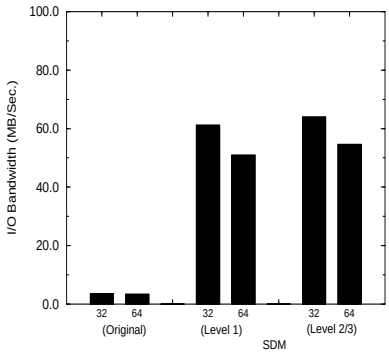

Figure 6: I/O bandwidth for RT

organizations. The difference between the three file organizations was not large because the file-open cost is small on XFS.

4.2 Results for RT

Figure 6 shows the I/O Bandwidth for writing approximately 550 MBytes of data. In the original application, the write operation was performed in a sequential way from one process. When we ported the application to SDM, the I/O performance increased significantly as a result of the I/O optimizations of MPI-IO. Since two data arrays are written to files separately, SDM supports two different ways of file organization: level 1 and level 2/3 (levels 2 and 3 are identical in this case). As shown in Figure 6, on the SGI Origin2000, changing the file organization did not affect the I/O performance because the cost of file open and file view is low. When the number of processors was increased from 32 to 64 to write the same data size, the I/O performance decreased because of the smaller granularity of data transfer.

5 Related Work

Numerous efforts have been made to optimize I/O in parallel file systems and runtime libraries [3, 5, 6, 13, 15, 17, 21, 26, 30]. SRB (Storage Resource Broker) [2] provides a uniform interface to access various storage systems, such as file systems, Unitree, HPSS, and database objects; however, it does not support optimizations such as collective I/O that MPI-IO provides. Shoshani et al. [27, 28] describe an architecture for optimizing access to large volumes of scientific data stored on tapes. The Active Data Repository [16] and DataCutter [4] optimize storage, retrieval, and processing of very large multidimensional datasets. The main difference between this work and other efforts in I/O is that this work aims to combine the good features of parallel file I/O and databases, whereas other efforts focus on either parallel I/O or data management, not both.

6 Summary

We have described the SDM system, API, and implementation for I/O in irregular applications. SDM provides an easy-to-use user interface for managing large data sets and internally uses MPI-IO for high-performance I/O and a database for storing metadata. We studied the performance of SDM using two irregular applications: FUN3D and RT. When we ported both applications to use SDM, there was a significant improvement in I/O performance compared with the original application. Also, we observed that using a history file for the index distribution helped reduce the computation and communication costs. However, changing the SDM file organization from level 1 to level 3 did not greatly affect the performance on the SGI Origin2000 because of its low file-open and file-view costs.

We plan to develop SDM further to support visualization applications and to investigate whether SDM can effectively be used as a strategy for implementing libraries such as HDF [20] and netCDF [32].

In the full version of this paper, we will provide more details about the SDM API for irregular applications and how it is implemented to take advantage of MPI-IO optimizations. We will also provide more performance results.

References

[1] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Achieving High Sustained Performance in an Unstructured Mesh CFD Application. In *Proc. of SC1999*, Winter 1999.

[2] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The SDSC Storage Resource Broker. In *Proceedings of CASCON '98*, December 1998.

[3] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A Framework for Optimizing Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–20. IEEE Computer Society Press, October 1994.

[4] Michael D. Beynon, Renato Ferreira, Tahsin Kurc, Alan Sussman, and Joel Saltz. DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage Systems. In *Proceedings of the Eighth Goddard Conference on Mass Storage Systems and Technologies*, March 2000.

[5] 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, November 1993.

[6] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[7] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[8] 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.

[9] High Performance Fortran Forum. High performance Fortran language specification, version 1.0. Technical Report Version 1.0, Rice University Houston Texas, May 1993.

[10] L. Freitag, M. Jones, and P. Plassmann. The Scalability of Mesh Improvement Algorithms. *IMA Volumes in Mathematics and Its Applications*, 105:185–212, May 1998.

[11] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface.* MIT Press, Cambridge, MA, 1999.

[12] Mike Holton and Raj Das. XFS: A Next Generation Journalled 64-Bit Filesystem With Guaranteed Rate I/O. Technical report, SGI, Inc, 1994.

[13] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A High Performance Portable Parallel File System. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394. ACM Press, July 1995.

[14] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *Journal on Scientific Computing*, 1997.

[15] David Kotz. Disk-directed I/O for MIMD Multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, February 1997.

[16] Tahsin Kurc, Chialin Chang, Renato Ferreira, Alan Sussman, and Joel Saltz. Querying Very Large Multi-dimensional Datasets in ADR. In *Proceedings of SC99: High Performance Networking and Computing*, November 1999.

[17] Tara M. Madhyastha and Daniel A. Reed. Intelligent, Adaptive File System Policy Selection. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 172–179. IEEE Computer Society Press, October 1996.

[18] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.

[19] MySQL Reference Manual. `http://www.mysql.com`, 1999. Version 3.23.10-alpha.

[20] National Center for Supercomputing Applications, University of Illinois. *NCSA HDF Reference Manual.* Version 3.3, February 1994.

[21] Nils Nieuwejaar and David Kotz. The Galley Parallel File System. *Parallel Computing*, 23(4):447–476, June 1997.

[22] Jaechun No, Rajeev Thakur, and Alok Choudhary. Integrating Parallel File I/O and Database Support for High-Performance Scientific Data Management. In *Proc. of SC2000: High Performance Networking and Computing*, November 2000.

[23] 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.

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

[25] Kirk Schloegel, George Karypis, and Vipin Kumar. Graph Partitioning for High Performance Scientific Simulations. 2000.

[26] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-Directed Collective I/O in Panda. In *Proceedings of Supercomputing '95*. ACM Press, December 1995.

[27] A. Shoshani, L. M. Bernardo, H. Nordberg, D. Rotem, and A. Sim. Storage Management for High Energy Physics Applications. In *Proceedings of Computing in High Energy Physics (CHEP '98)*, 1998.

[28] A. Shoshani, L. M. Bernardo, H. Nordberg, D. Rotem, and A. Sim. Multidimensional Indexing and Query Coordination for Tertiary Storage Management. In *Proc. of SSDBM'99*, pages 214–225, July 1999.

[29] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proc. of USENIX 1996 Annual Technical Conference*, San Diego, CA, January 1996.

[30] Rajeev Thakur and Alok Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.

[31] Rajeev Thakur, William Gropp, and Ewing Lusk. A Case for Using MPI's Derived Datatypes to Improve I/O Performance. In *Proceedings of SC98: High Performance Networking and Computing*, November 1998.

[32] Unidata Program Center, University Corporation for Atmospheric Research. *netCDF User's Guide*. Version 2.0, October 1991.

[33] Reinhard v. Hanxleden, Ken Kennedy, and Joel Saltz. Value-Based Distributions and Alignments in Fortran D. *Journal of Programming Languages - Special Issue on Compiling and Run-Time Issues for Distributed Address Space Machines*, May 1994.