Integrating Parallel File I/O and Database Support for High-Performance Scientific Data Management*

Jaechun No Rajeev Thakur
Math. and Computer Science Division
Argonne National Laboratory
{jano, thakur}@mcs.anl.gov

Alok Choudhary
Dept. of Electrical and Computer Eng.
Northwestern University
choudhar@ece.nwu.edu

Preprint ANL/MCS-P798-0300
March 2000

Abstract

Many scientific applications have large I/O requirements, in terms of both the size of data and the number of files or data sets. Management, storage, efficient access, and analysis of this data present an extremely challenging task. Traditionally, two different solutions are used for this problem: file I/O or databases. File I/O can provide high performance but is tedious to use with large numbers of files and large and complex data sets. Databases can be convenient, flexible, and powerful but do not perform and scale well for parallel supercomputing applications. We have developed a software system, called Scientific Data Manager (SDM), that combines the good features of both file I/O and databases. SDM provides a thin layer of database-like functionality on top of a high-performance, parallel file-I/O interface (MPI-IO). As a result, users can access data with the convenience of databases and the performance of MPI-IO, without having to bother with the details of either. In this paper, we describe the design and implementation of SDM. With the help of two parallel application templates, ASTRO3D and an Euler solver, we illustrate how some of the design criteria affect performance.

*This work was supported 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.

1 Introduction

Many large-scale scientific experiments and simulations generate very large amounts of data [7, 19] (on the order of several hundred gigabytes to terabytes), spanning thousands of "files" or data sets. Management, storage, efficient access, and analysis of this data present an extremely challenging task. Currently available techniques for this purpose are either raw file-I/O interfaces, such as MPI-IO [10, 15], or full-fledged databases. File-I/O interfaces provide high performance but are too cumbersome to use with large, complex data sets and large numbers of files. For example, the user must remember file names and the organization of data in a file and must specify the exact location in the file from which data must be accessed. Databases, on the other hand, provide a convenient, high-level interface and powerful data-retrieval capability, but they do not measure up to the performance requirements of large-scale scientific applications running on supercomputers.

We have developed a solution that combines the good features of both file I/O and databases. Specifically, we have developed a software system, called Scientific Data Manager (SDM), that provides a thin layer of database-like functionality on top of a high-performance, parallel file-I/O interface (MPI-IO). SDM provides a high-level, user-friendly interface. Internally, SDM interacts with a database to store application-related metadata, and it uses MPI-IO to store the real data on a high-performance parallel file system. It 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 convenience of databases and the performance of parallel file I/O, without having to bother with the details of either. Figure 1 illustrates the basic idea.

In this paper, we describe the design and implementation of SDM. SDM is currently implemented to use either MySQL [16] or PostgreSQL [20] as the database for metadata and MPI-IO for file I/O. In designing such a system, we have a wide choice of how to organize the data in files. We have implemented three different ways of organizing data in files. At one extreme, level 1, we store all data sets in separate files as they are generated. At the other extreme, level 3, we store data sets in a very small number of files and, using a database table, keep track of where in the files each data set is stored. We also have an intermediate approach, called level 2. We examine the performance implications of using each of these approaches by studying the performance results obtained for two application templates, ASTRO3D and an Euler solver, on an IBM SP and SGI Origin2000.

The rest of this paper is organized as follows. In Section 2 we discuss our goals in developing SDM. In Section 3 we describe how SDM is implemented. Performance results are presented in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2 Design Objectives

We had three major goals in developing SDM: provide high-performance parallel I/O, provide a high-level application programming interface (API) that eliminates the need for the user to bother with the details of low-level file I/O or databases, and store enough metadata in a database so that the user can easily retrieve previously stored data.

- **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 [10, 15], is rapidly emerging as the standard, portable API for I/O in parallel applications. High-performance implementations of MPI-IO, both vendor and public-domain implementations, are available for most platforms [8, 13, 21, 22, 30]. 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 with a single function, and the ability to pass hints to the implementation about access patterns, file-striping parameters, and so forth.

- **High-Level API**. Our goal was to provide an API that did not require the user to know either MPI-IO or databases. The user can specify the data with a high-level description, together with

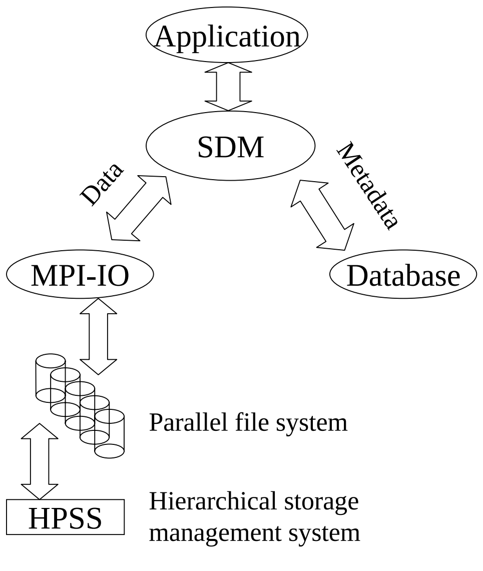


Figure 1: SDM architecture

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 [29]. SDM also interacts with the database when necessary, by using embedded SQL functions.

- **Convenient Data-Retrieval Capability**. SDM allows the user to specify names and other attributes to be associated with a data set. SDM internally selects a file name into which the data will be stored; the mapping between data sets and file names is stored in the database. The user can retrieve a data set by specifying a unique set of attributes or by specifying the date of the desired run (if no date is specified, data from the last run is retrieved).

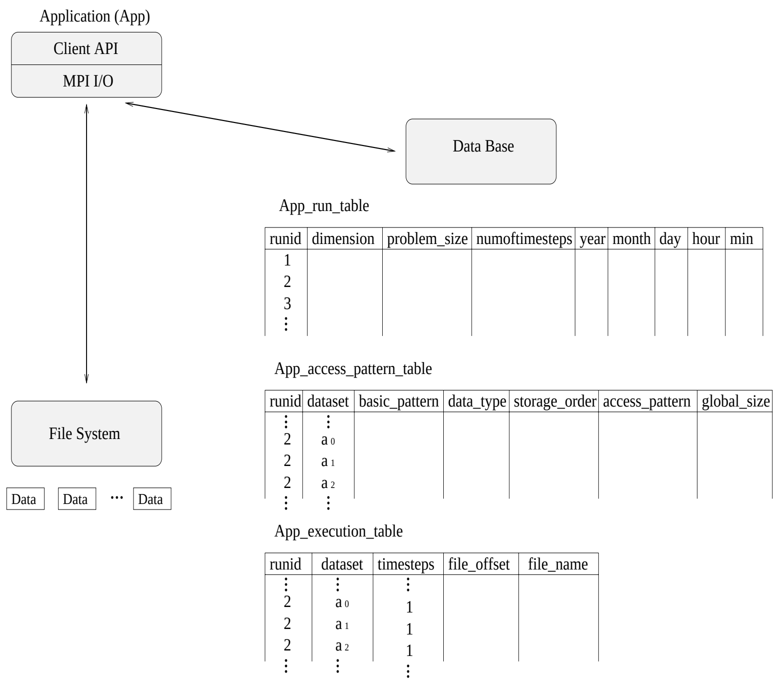
3 Implementation

We briefly describe how we implemented SDM; details will be provided in the full paper. We describe the metadata storage in the database, the SDM API, and the organization of data in files.

To explain the implementation, we use the example of an astrophysics application, ASTRO3D, developed at the University of Chicago. For simplicity of explanation, we consider the two-dimensional version of this three-dimensional application. (The performance results presented in this paper are for the full three-dimensional version.). In this application data is stored in arrays that are block-distributed in each dimension among processes. The application generates three floating-point data sets for data analysis, which are written to files at every six time steps; four character-type data sets for data visualization, which are written at every four time steps; and three floating-point data sets for restart, which are written every six time steps. (The frequencies of all these writes can be varied.) Let a_0, a_1, a_2 be the data sets for data analysis, b_0, b_1, b_2, b_3 be the data sets for visualization, and c_0, c_1, c_2 be the data sets for restart.

3.1 Database Tables to Store Metadata

SDM uses three database tables for storing metadata: *run_table*, *access_pattern_table*, and *execution_table* (see Figure 2). These tables are made for each application. Each time an application writes data sets, SDM enters the problem size, dimension, current date, and a unique identification number (runid) to the run_table. The access_pattern_table includes the properties of each data set, such as data type, storage order,



runid	dimension	problem_size	numoftimesteps	year	month	day	hour	min
1								
2								
3								
⋮								

runid	dataset	basic_pattern	data_type	storage_order	access_pattern	global_size
⋮	⋮					
2	a_0					
2	a_1					
2	a_2					
⋮	⋮					

runid	dataset	timesteps	file_offset	file_name
⋮	⋮	⋮		
2	a_0	1		
2	a_1	1		
2	a_2	1		
⋮	⋮	⋮		

Figure 2: Database tables used in SDM

data access pattern, and global size. SDM uses this information to make appropriate MPI-IO calls to access the real data. The execution_table stores a globally determined file offset denoting the starting offset in the file of each data set.

3.2 Application Programming Interface

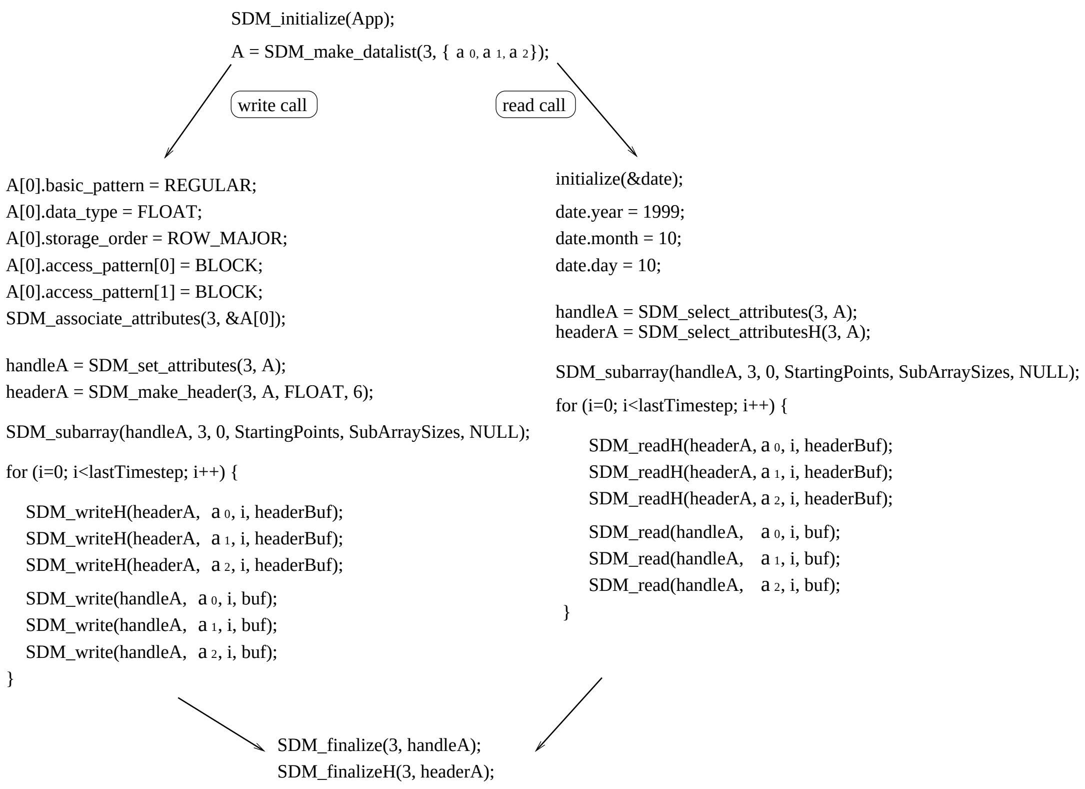
To use SDM, the user must first call the function `SDM_initialize`. This function initializes the SDM environment and establishes a connection to the database. Next, to specify the three groups of data sets (data analysis, visualization, and restart) in our example application, ASTRO3D, described above, the user must call the function `SDM_make_datalist`. This function assigns properties to the first data set in a group. The same properties can be assigned to other data sets in the same group by calling `SDM_associate_attributes`.

The main reason for making groups of data sets is that SDM can then use different ways of organizing data in files, with different performance implications. For example, each data set can be written in a separate file, or the data sets of a group can be written to a single file.

In the case of write operations, the user must call `SDM_set_attributes` to set the attributes associated with a group and to return a set of handles to be used for further I/O operations. If an application writes header information along with the data, `SDM_make_header` must be used to return an array of handles for writing the header information.

In the case of read operations, a date input can be given if the user wants to retrieve data sets from a specific run. If not, the data sets produced by the last run are read. Also, the properties of the data sets need not be specified, since a chosen process (process 0) retrieves this information from the database and broadcasts it to others. This is done in the `SDM_select_attributes` function. If the application has header information to be retrieved, `SDM_select_attributesH` must be called.

The main SDM functions for writing and reading data are `SDM_write` and `SDM_read`. Before calling these functions, the user must provide some information necessary for SDM to perform I/O, for example, the starting points and sizes of the subarray in each dimension in case of block distribution, or the size of process grids and distribution arguments in each dimension in the case of cyclic distribution. To perform I/O, the handle of a group, position of a data set within the handle (group), current time step, and pointer to the



```
SDM_initialize(App);
A = SDM_make_datalist(3, { a 0, a 1, a 2});
```

```
A[0].basic_pattern = REGULAR;
A[0].data_type = FLOAT;
A[0].storage_order = ROW_MAJOR;
A[0].access_pattern[0] = BLOCK;
A[0].access_pattern[1] = BLOCK;
SDM_associate_attributes(3, &A[0]);

handleA = SDM_set_attributes(3, A);
headerA = SDM_make_header(3, A, FLOAT, 6);

SDM_subarray(handleA, 3, 0, StartingPoints, SubArraySizes, NULL);

for (i=0; i<lastTimestep; i++) {

   SDM_writeH(headerA,  a 0, i, headerBuf);
   SDM_writeH(headerA,  a 1, i, headerBuf);
   SDM_writeH(headerA,  a 2, i, headerBuf);

   SDM_write(handleA,  a 0, i, buf);
   SDM_write(handleA,  a 1, i, buf);
   SDM_write(handleA,  a 2, i, buf);
}
```

```
initialize(&date);
date.year = 1999;
date.month = 10;
date.day = 10;

handleA = SDM_select_attributes(3, A);
headerA = SDM_select_attributesH(3, A);

SDM_subarray(handleA, 3, 0, StartingPoints, SubArraySizes, NULL);

for (i=0; i<lastTimestep; i++) {

    SDM_readH(headerA, a 0, i, headerBuf);
    SDM_readH(headerA, a 1, i, headerBuf);
    SDM_readH(headerA, a 2, i, headerBuf);

    SDM_read(handleA,   a 0, i, buf);
    SDM_read(handleA,   a 1, i, buf);
    SDM_read(handleA,   a 2, i, buf);
}
```

```
SDM_finalize(3, handleA);
SDM_finalizeH(3, headerA);
```

Figure 3: Example of using the SDM API to perform I/O in the astrophysics application

application buffer are passed to the SDM I/O function. Note that the user does not have to provide file names. SDM generates the file name and records the name in the database.

Finally, the user must call `SDM_finalize` and `SDM_finalizeH`, to close all files, close the connection to the database server, and free all memory allocated by SDM.

Figure 3 shows how the SDM API is used to perform I/O in a two-dimensional version of the astrophysics application.

3.3 File Organization

SDM supports three different ways of organizing data in files. In level-1 file organization, each data set generated at each time step is written to a separate file, as shown in Figure 4. This file organization is simple, but it incurs the cost of a file open and close at each time step, which on some file systems can be quite high, as we shall see in the performance results. For a large number of data sets and time steps, this method can be expensive because of the large number of file opens.

In level 2, each data set (within a group) is written to a separate file, but different iterations of the same data set are appended to the same file. This is illustrated in Figure 5. This method results in a smaller number of files and smaller file-open costs. The offset in the file where data is appended is stored in the execution_table.

In level 3, all iterations of all data sets belonging to a group are stored in a single file, as shown in Figure 6. As in level 2, the file offset for each data set is stored in the execution_table by process 0 in the `SDM_write` function. The idea is that if a file system has high open and close costs, SDM can generate a very small number of files. If an application produces a large number of data sets with a large problem size, level-3 file organization would result in very large files, which may affect performance.

We study the performance implications of the three file-organization levels in the next section.

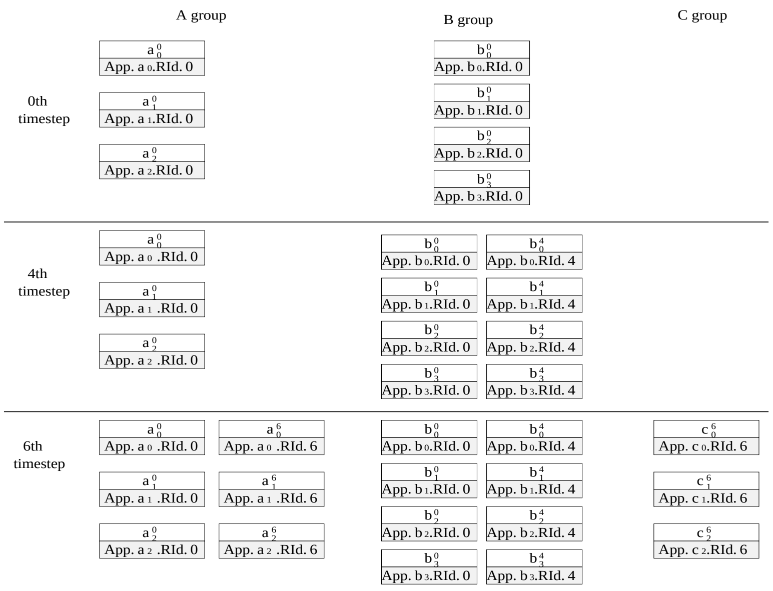


Figure 4: Level-1 file organization. The superscript on a data set denotes the time step, and the shadowed area in each box shows the SDM-generated name of the file in which the corresponding data set is written.

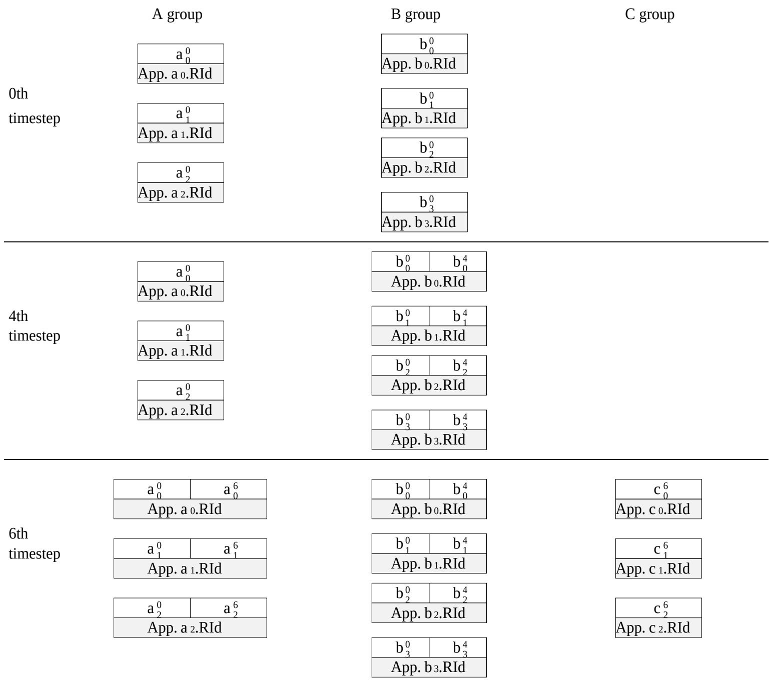


Figure 5: Level-2 file organization. The superscript on a data set denotes the time step, and the shadowed area in each box shows the SDM-generated name of the file in which the corresponding data set is written.

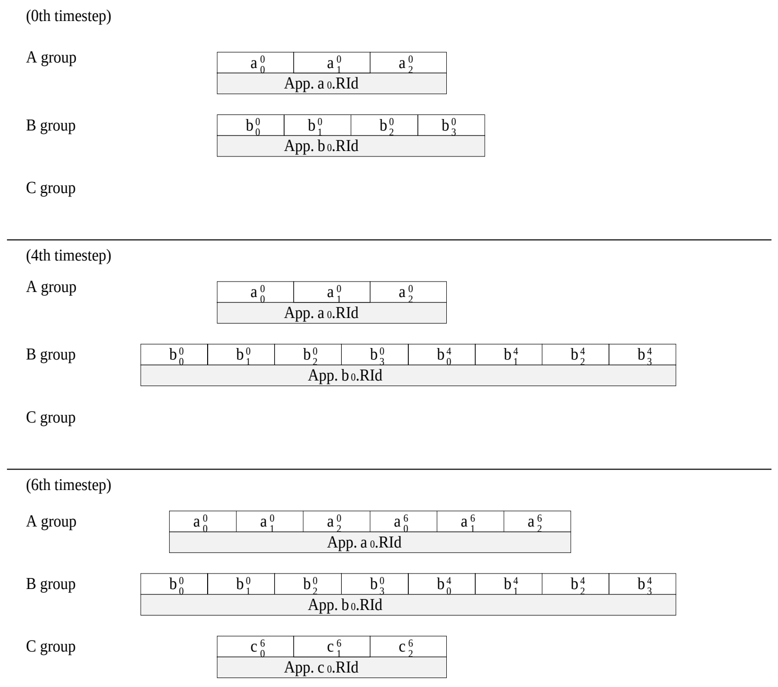


Figure 6: Level-3 file organization. The superscript on a data set denotes the time step, and the shadowed area in each box shows the SDM-generated name of the file in which the corresponding data set is written.

4 Performance Results

We obtained all performance results on the IBM SP and SGI Origin2000 at Argonne National Laboratory. The IBM SP has 80 compute nodes and 4 I/O nodes. Each I/O nodes controls four SSA disks, each of 9 Gbyte capacity. The parallel file system on the machine is IBM's PIOFS [2]. The SGI Origin2000 has 128 processors and 10 Fibre Channel controllers connected to a total of 110 disks of 9 Gbyte capacity each. The file system on the Origin2000 is SGI's XFS [11, 26]. XFS supports an optimization called direct I/O, which we used in our experiments. When certain alignment restrictions are met, the user can choose the direct-I/O option, in which the file system moves data directly between the user's buffer and the storage device, bypassing the file-system cache. Direct I/O eliminates an extra memory copy into the cache and can perform well for large I/O and high-bandwidth storage systems. Direct I/O can be used from an MPI-IO program—the ROMIO implementation of MPI-IO that we used supports direct I/O [31]. We present performance results with both direct I/O and regular (buffered) I/O.

We used two application templates, ASTRO3D and a three-dimensional Euler solver, in our performance experiments. As mentioned in Section 3, ASTRO3D is an astrophysics application, developed at the University of Chicago, that writes several three-dimensional distributed arrays for visualization, restart, and data analysis [28]. We used a problem size of $256 \times 256 \times 256$, which resulted in a total of around 880 Mbytes of data per iteration for all arrays.

The second application is a three-dimensional Euler solver for the problem of three-dimensional transonic flow about an M6 wing [9]. This application is a mesh-structured code that writes the physical values and residual of each node at certain iterations. The structure of these values is a distributed global vector, and each value has five components (density, energy, and three coordinates of momentum). In addition, the application writes the physical coordinates and pressure at each mesh point. In our experiments, we ran the code for 50 iterations and wrote data at every 5 iterations. The problem size was $194 \times 34 \times 34$.

4.1 Costs of Database Access

SDM uses TCP/IP to connect to the database servers. We performed our experiments with two different databases, MySQL [16] and PostgreSQL [20]. Figure 7 shows the database-access cost of the write operation in the ASTRO3D and Euler solver on the SGI. As described in Section 3, the connection and disconnection to the database server occur once in `SDM_initialize` and `SDM_finalize`, respectively. In `SDM_set_attributes`, process 0 accesses the run_table and access_pattern_table to store attributes, and in the write operation, it stores the file offset into the execution_table. In ASTRO3D, the access to the execution_table occurred 19 times, and in the Euler solver, the access to the execution_table occurred 60 times. As can be seen in Figure 7, the database-access cost using both the database servers is less than 0.6 sec. This cost, however, will change according to the number of I/O operations occurring in the applications.

We observed that MySQL performs better than PostgreSQL. Therefore, we used only MySQL in the rest of the performance experiments.

4.2 Results for ASTRO3D

Figure 8 shows the write and read bandwidths for ASTRO3D on the IBM SP using 32 processors for the three levels of file organization. We used only one iteration of the program; therefore, levels 1 and 2 resulted in the same file organization. Level 3 results in much higher bandwidth because only three different files are created, and, therefore, only three file opens occur. The high cost of file opens on the PIOFS file system [28] results in lower performance for levels 1 and 2, where 19 separate files are created. The impact of file-open time can indeed be quite large.

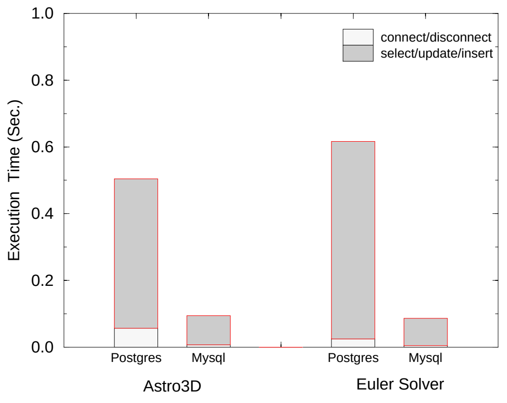


Figure 7: Cost of accessing the database for the two applications

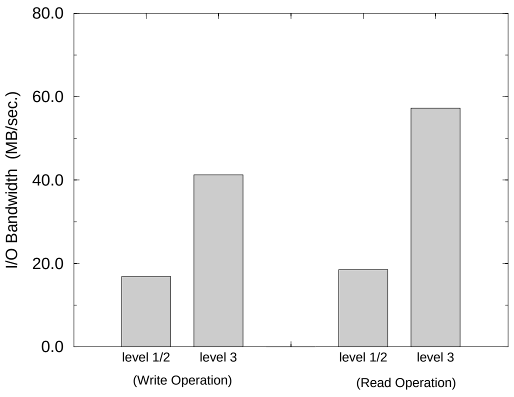


Figure 8: I/O bandwidth for ASTRO3D on the IBM SP

Figures 9 and 10 show the write and read bandwidths for ASTRO3D on the SGI using 16 processors. We measured performance for both direct I/O and buffered I/O. For writing data, direct I/O performed better than buffered I/O. There are two reasons for this. First, with buffered I/O, XFS serializes concurrent writes to the same file, whereas with direct I/O, concurrent writes are allowed to proceed in parallel. Second, direct I/O eliminates a copy into the file-system cache. For reading data, buffered I/O performed better. Again, there are two reasons for this. One reason is that XFS does not serialize buffered reads, so direct reads do not have any extra advantage in the area of parallelism. The second reason is that XFS performs a read-ahead (prefetch) in the case of buffered reads, and not in case of direct reads. The read-ahead policy works well for this application, and buffered reads therefore perform better.

Since the cost of file opens is small on XFS, the three levels of file organization perform nearly the same.

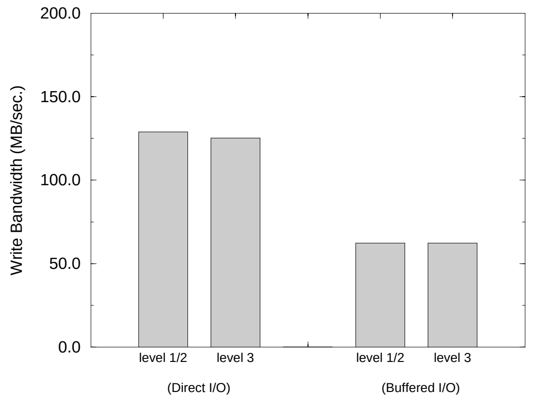


Figure 9: Write bandwidth for ASTRO3D on the SGI Origin2000

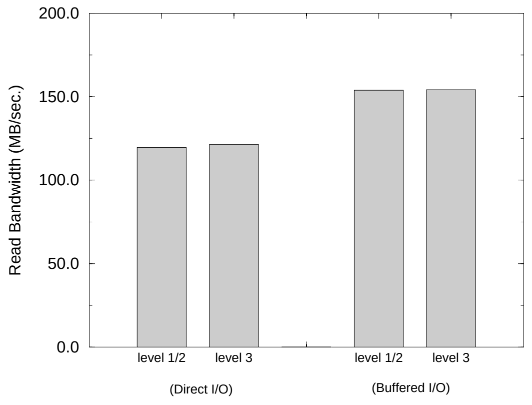


Figure 10: Read bandwidth for ASTRO3D on the SGI Origin2000

4.3 Results for the Euler Solver

Figures 11 and 12 show the write and read bandwidths for the Euler solver on the IBM SP using 32 processors. The total data size written was around 240 Mbytes. In level 3, only two files were generated, one for writing the coordinates and pressure at each mesh node and the other for writing the physical values and residual at each node. In level 2, six vectors (that is, the three coordinates, pressure, physical values of each node, and nodal residual) were written separately, resulting in a total of six files. In level 1, the six vectors generated every five iterations were written separately, resulting in a total of 60 files. As Figures 11 and 12 show, level 3 performs the best because of the high open cost on PIOFS. In level 1, the file open cost takes around 80% of the total execution time; in level 2, it takes around 30%; and in level 3, it takes around 20% of the total execution time.

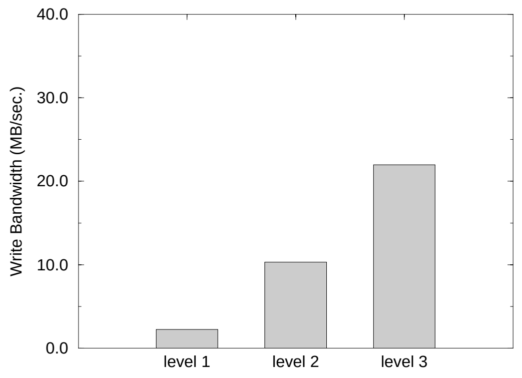


Figure 11: Write bandwidth for the Euler solver on the IBM SP

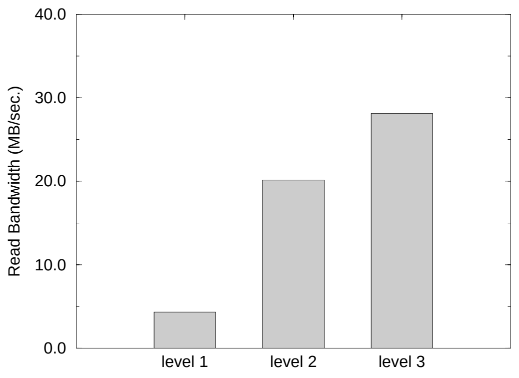


Figure 12: Read bandwidth for the Euler solver on the IBM SP

Figures 13 and 14 show the write and read bandwidths for the Euler solver using 16 processors on the SGI. For this application, we used only buffered I/O. We could not use direct I/O because the memory allocation for distributed vectors was done inside the numerical library (PETSc [18]) that the application uses, and thus we could not align the buffers to the cache line as required for direct I/O. For the write operation, levels 2 and 3 performed slightly better than level 1. For the read operation, however, level 1 performed the best. The reason is that the read-ahead policy of XFS for buffered reads operates on a per-file basis, and therefore works to the application's advantage when it has more number of files.

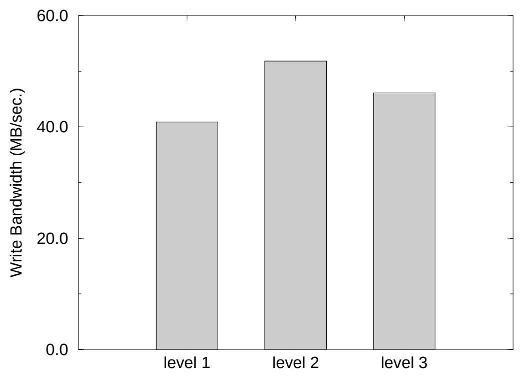


Figure 13: Write bandwidth for the Euler solver on the SGI Origin2000

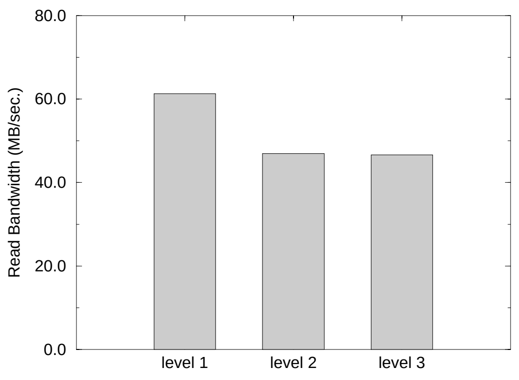


Figure 14: Read bandwidth for the Euler solver on the SGI Origin2000

5 Related Work

SRB (Storage Resource Broker) [1] provides a uniform interface to access various storage systems, such as file systems, Unitree, HPSS, and database objects. However, it does not fully support the optimizations implemented in MPI-IO. Shoshani et al. [24, 25] describe an architecture for optimizing access to large volumes of scientific data stored on tapes. Chervenak et al. [4] describe a general architecture for managing distributed scientific data sets in a *grid* environment. An architecture for data-intensive distributed computing using DPSS is described in [32, 33]. An initial discussion of a framework for scientific data management similar to the one described in this paper is given in [5].

There have been several efforts to optimize I/O in parallel file systems and runtime libraries [3, 6, 12, 14, 17, 23, 27]. However, file systems and libraries have a lower-level interface than SDM, requiring more work from the user.

6 Conclusions

We have presented the design and implementation of an environment for high-performance scientific data management, called Scientific Data Manager (SDM), which is built on top of MPI-IO and also interacts with a database for storing metadata. SDM provides a simple, high-level interface and performs all necessary I/O optimizations transparent to the user. We also experimented with different ways of organizing data in files, called level 1–level 3. In general, when file-open cost on a particular file system is high, level 3 performs well because it minimizes the number of files created. If the file-open cost is small, the performance of the three levels depends on how the number and size of files affects performance on the particular file system. An appropriate file organization policy can thereby be chosen for a particular file system.

On the XFS file system, we found that the file open cost was so small that it did not significantly affect the I/O performance. Instead, our experiment focused on the use of direct I/O and buffered I/O in the ASTRO3D template. For writing data, we found that direct I/O generated much better performance than buffered I/O by avoiding the overhead of copying the data into buffer cache. For reading data, however, buffered I/O performed better because of its read-ahead policy.

We are developing SDM further to support other types of applications, particularly unstructured-grid applications. We also plan to use it for efficiently retrieving subsets of data for visualization applications.

References

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

[2] Fern E. Bassow. Installing, managing, and using the IBM AIX Parallel I/O File System. IBM Document Number SH34-6065-00, February 1995. IBM Kingston, NY.

[3] 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.

[4] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets. In *Proceedings of the Network Storage Symposium (NetStore '99)*, October 1999.

[5] A. Choudhary, M. Kandemir, H. Nagesh, J. No, X. Shen, V. Taylor, S. More, and R. Thakur. Data Management for Large-Scale Scientific Computations in High Performance Distributed Systems. In *Proc. of the Eighth IEEE Int'l Symposium on High Performance Distributed Computing*, pages 263–272, August 1999.

[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] 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.

[8] Samuel A. Fineberg, Parkson Wong, Bill Nitzberg, and Chris Kuszmaul. PMPIO—A Portable Implementation of MPI-IO. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 188–195. IEEE Computer Society Press, October 1996.

[9] W. D. Gropp, D. E. Keyes, L. C. McInnes, and M. D. Tidriri. Globalized Newton-Krylov-Schwarz Algorithms and Software for Parallel Implicit CFD. Technical Report ICASE TR 98-24 (to appear in Int. J. Supercomputer Applications), 1998.

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

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

[12] 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.

[13] Terry Jones, Richard Mark, Jeanne Martin, John May, Elsie Pierce, and Linda Stanberry. An MPI-IO Interface to HPSS. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems*, pages I:37–50, September 1996.

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

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

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

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

[18] PETSc 2.0 for MPI. `http://www.mcs.anl.gov/petsc`.

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

[20] Postgres Global Development Group. *PostgreSQL User's Guide*, 1996.

[21] Jean-Pierre Prost. MPI-IO/PIOFS. World-Wide Web page at `http://www.research.ibm.com/people/p/prost/sections/mpiio.html`, 1996.

[22] D. Sanders, Y. Park, and M. Brodowicz. Implementation and Performance of MPI-IO File Access Using MPI Datatypes. Technical Report UH-CS-96-12, University of Houston, November 1996.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] Rajeev Thakur, William Gropp, and Ewing Lusk. An Experimental Evaluation of the Parallel I/O Systems of the IBM SP and Intel Paragon Using a Production Application. In *Proceedings of the 3rd International Conference of the Austrian Center for Parallel Computation (ACPC) with Special Emphasis on Parallel Databases and Parallel I/O*, pages 24–35. Lecture Notes in Computer Science 1127. Springer-Verlag, September 1996.

[29] 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.

[30] Rajeev Thakur, William Gropp, and Ewing Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.

[31] Rajeev Thakur, Ewing Lusk, and William Gropp. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, Revised December 1999.

[32] Brian Tierney, William Johnston, Jason Lee, and Mary Thompson. A Data Intensive Distributed Computing Architecture for Grid Applications. *Future Generation Computer Systems*, 2000. To appear.

[33] Brian Tierney, Jason Lee, Brian Crowley, and Mason Holding. A Network-Aware Distributed Storage Cache for Data Intensive Environments. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, August 1999.