High-Performance Scientific Data Management System[*]

Jaechun No
Dept. of Software Engineering
Sejong University
Seoul, Republic of Korea

Rajeev Thakur
Math. and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA

Alok Choudhary
Dept. of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208, USA

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 have been used for this task: 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 high-level API to the user and, internally, uses a parallel file system to store real data (using various I/O optimizations available in MPI-IO) and a database to store application-related metadata. In order to support I/O in irregular applications, SDM makes extensive use of MPI-IO's noncontiguous collective I/O functions. Moreover, SDM uses the concept of a *history file* to optimize the cost of the index distribution using the metadata stored in database. We describe the design and implementation of SDM and present performance results with two regular applications, ASTRO3D and an Euler solver, and with two irregular applications, a CFD code called FUN3D and a Rayleigh-Taylor instability code.

Keywords: scientific data management, parallel I/O, MPI-IO, database, metadata

Proposed running head: High-Performance Scientific Data Management System

[*]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 experiments and simulations generate very large amounts of data [2, 9] (on the order of several hundred gigabytes to terabytes), spanning thousands of files or data sets. Moreover, such applications have different patterns of accessing data from files. Some applications access data in regular block or cyclic access patterns. Others have more complex irregular access patterns that cannot be detected at compile time. These different ways of accessing data in applications, along with the large number of files and large size of data being generated, make high-performance access, management, storage, and analysis of the data an extremely challenging task.

Current techniques for data management are either raw file-I/O interfaces, such as MPI-IO [13, 24], 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 the 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 software system, called Scientific Data Manager (SDM), that combines the good features of both file I/O and databases [28, 29]. 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.

SDM provides efficient I/O support for irregular applications, while maintaining a high-level unified API for both regular and irregular applications. In irregular applications, the data accesses make extensive use of arrays, called indirection arrays [8] or map arrays [13], in which each value of the array denotes the corresponding data position in memory or in the file. The data distribution in irregular applications can be done either by using compiler directives with the support of runtime preprocessing [15, 14] or by using a runtime library [8]. Most of the previous work in the area of unstructured-grid applications focuses mainly on computation and communication in such applications, not on I/O. SDM takes advantage of MPI-IO's support for noncontiguous data accesses and, therefore, can efficiently handle the reading and writing of data in an irregular mesh, as well as the distribution of index values.

The rest of this paper is organized as follows. In Section 2 we discuss our goals in developing SDM. In Section 3 we present the design and implementation of SDM. Performance results on the SGI Origin2000 and on the IBM SP at Argonne National Laboratory are presented in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2 Design Objectives

Our main objectives in developing SDM were to provide high-performance parallel I/O, to provide a high-level application programming interface (API), to support a convenient data-retrieval capability, and to optimize the execution time of both regular and 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 [13, 24], 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 [10, 19, 32, 33, 42]. 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, 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 a high-level unified API for any kind of application (regular or irregular) while encapsulating the details of either MPI-IO or a database. 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 [41]. 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 for the desired data. SDM also allows the user to query the metadata for a stored data set and then select data with specific attributes.

- **Optimization of 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 by using the metadata stored in the database, and thereby the user can avoid repeating the communication and computation for the same index distribution. SDM also uses MPI datatypes and collective I/O functions to optimize I/O for irregular access patterns.

3 Implementation

With the help of sample regular and irregular problems, we describe the design and implementation of SDM.

3.1 Regular Applications

We present a sample regular problem and describe the metadata storage in the database, the SDM API, and the organization of data in files.

3.1.1 Problem Description and SDM API

ASTRO3D is a three-dimensional astrophysics application 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 several arrays that are block distributed in each dimension. At various time steps, several of these arrays are written to files for data analysis, restart, and visualization. Six floating-point arrays are written for data analysis and another six for restart; seven character arrays are written for visualization. The frequencies of the writes can be varied.

We use the term *data set* to refer to each array being written and *data group* to refer to all the arrays written at a time step for a particular purpose such as data analysis. For simplicity of explanation, let us assume that three arrays are written for data analysis, another three for restart, and four for visualization.

```
SDM_initialize(ASTRO3D);
A = SDM_make_datalist(3, {$a_0, a_1, a_2$});
A[0].data_type = FLOAT;
A[0].access_pattern[0] = BLOCK;
A[0].access_pattern[1] = BLOCK;
SDM_associate_attributes(3, &A[0]);

handle = SDM_set_attributes(A, 3);
SDM_subarray(handle,3,0,starts,subsizes,NULL);
SDM_data_view(handle,3,0,NULL,NULL);

For (i=1; $i < maxStep$; i++) {
        ......
        Do computation;
        ......
        SDM_write(handle, $a_0$, i, $a_0buf$);
        SDM_write(handle, $a_1$, i, $a_1buf$);
        SDM_write(handle, $a_2$, i, $a_2buf$);

}

SDM_finalize(handle, 3);

                    (a)
```

```
SDM_initialize(ASTRO3D);
A = SDM_make_datalist(3, {$a_0, a_1, a_2$});
initialize(&date);
date.year = 2000;
date.month = 10;
date.day = 10;

handle = SDM_select_attributes(A, 3);
SDM_subarray(handle,3,0,starts,subsizes,NULL);
SDM_data_view(handle,3,0,NULL,NULL);

For (i=1; $i < maxStep$; i++) {

        SDM_read(handle, $a_0$, i, $a_0buf$);
        SDM_read(handle, $a_1$, i, $a_1buf$);
        SDM_read(handle, $a_2$, i, $a_2buf$);
        ......
        Do computation;
        ......
}

SDM_finalize(handle, 3);

                    (b)
```

Figure 1: SDM API to perform (a) write and (b) read operations for data group A in ASTRO3D

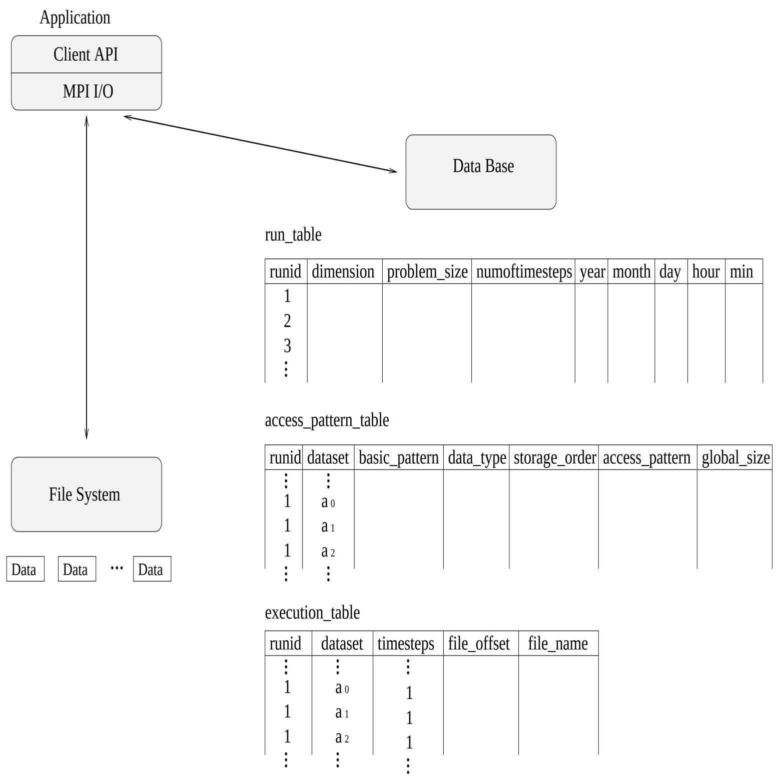
(Note that all arrays—six, six, and seven—were used in the performance experiments reported in this paper.) Let us further assume that the data-analysis and restart dumps are performed every six time steps and the visualization dumps are performed every four time steps. Let a_0, a_1, a_2 be the three data sets for data analysis and $A = (a_0, a_1, a_2)$ be the data group for data analysis. Similarly, we have $B = (b_0, b_1, b_2, b_3)$ for visualization and $C = (c_0, c_1, c_2)$ for restart.

Figure 1 shows how the SDM API is used to perform (a) write and (b) read operations for data analysis (data group A) in a two-dimensional version of ASTRO3D.

3.1.2 Implementation Details

SDM provides a high-level API and stores application-related metadata in a database. For regular applications where the data access pattern can be predicted before runtime, SDM creates three database tables: *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, 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.

In SDM, users can specify groups of data sets by assigning properties to the first data set in a group and by propagating them to the other data sets belonging to the same group. 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



run_table

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

access_pattern_table

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

execution_table

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

Figure 2: Database tables used for solving regular applications in SDM

written to a single file. For ASTRO3D, we created the three data groups, A = a_0, a_1, a_2, B = b_0, b_1, b_2, b_3, and C = c_0, c_1, c_2, for data analysis, visualization, and restart, respectively. The properties assigned to each data set are stored in the database by calling `SDM_set_attributes`.

In case of read operations, data from a specific run can be retrieved by specifying attributes of the data, such as the date of the run. Also, the properties of the data sets need not be specified because SDM retrieves this information from the database.

The main SDM functions for writing and reading data are `SDM_write` and `SDM_read`. Before calling these functions, the user must provide the information necessary for SDM to perform I/O, such as the starting points and sizes of the subarray in each dimension in the case of block distribution, or the size of process grids and distribution arguments in each dimension in the case of cyclic distribution. Also, `SDM_data_view` must be called to perform the data mapping between memory and file.

In order 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 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. SDM calls MPI-IO's collective I/O functions to perform I/O efficiently and in parallel from all processes.

3.1.3 File Organization

SDM supports three different ways of organizing data in files. In level 1, each data set generated at each time step is written to a separate file, as shown in Figure 3. 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

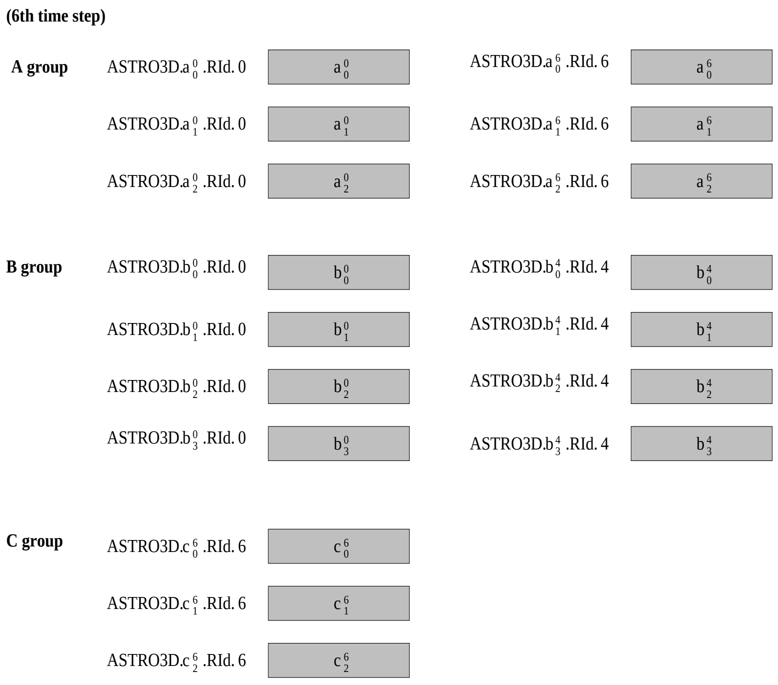


Figure 3: Level-1 file organization at 6th time step in ASTRO3D. The superscript on a data set denotes the time step at which the data set has been written to a file, RId denotes the current identification number (runid), and each shadowed box (along with the name beside it) shows the SDM-generated file for storing the corresponding data set.

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, as illustrated in Figure 4. 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 5. 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. If a file system has high open and close costs, SDM can generate a very small number of files by choosing the level-3 file organization. On the other hand, if an application produces a large number of data sets of large size, level 3 would result in very large files, which may affect performance.

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

3.2 Irregular Applications

We now describe the SDM architecture for supporting irregular applications.

3.2.1 Problem Description and SDM API

Figure 6 shows a typical irregular application 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`,

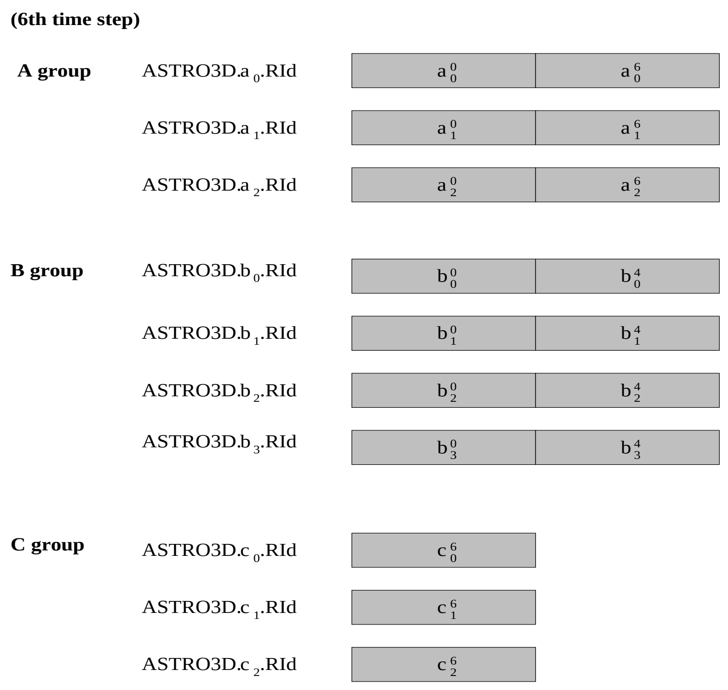


Figure 4: Level-2 file organization at 6th time step in ASTRO3D. The superscript on a data set denotes the time step at which the data set has been written to a file, RId denotes the current identification number (runid), and each shadowed box (along with the name beside it) shows the SDM-generated file for storing the corresponding data set.

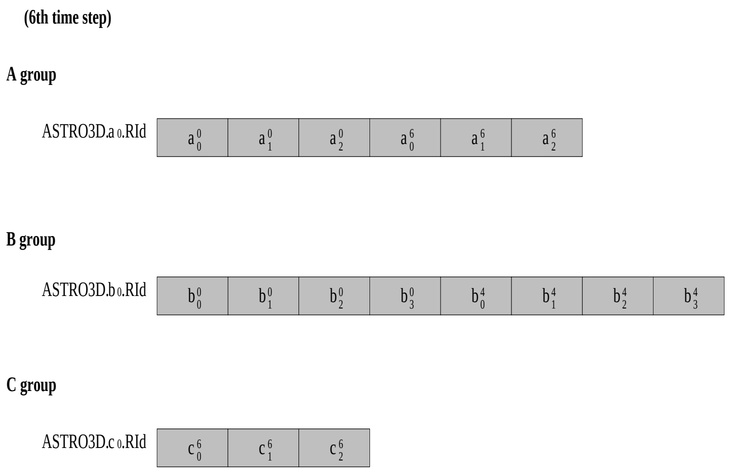
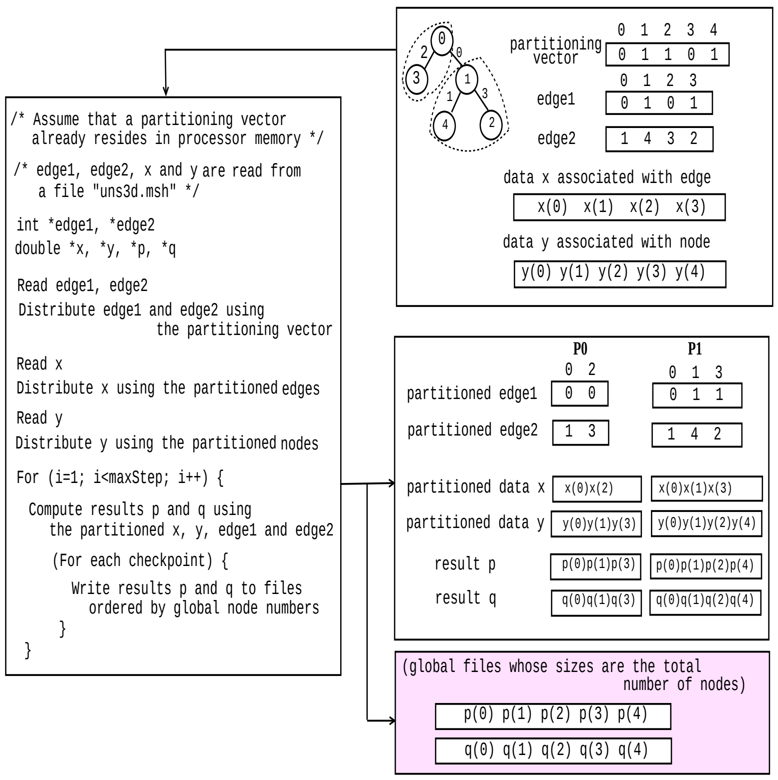


Figure 5: Level-3 file organization at 6th time step in ASTRO3D. The superscript on a data set denotes the time step at which the data set has been written to a file, RId denotes the current identification number (runid), and each shadowed box (along with the name beside it) shows the SDM-generated file for storing the corresponding data set.



```
/* Assume that a partitioning vector
   already resides in processor memory */

/* edge1, edge2, x and y are read from
    a file "uns3d.msh" */

int *edge1, *edge2
double *x, *y, *p, *q

Read edge1, edge2
Distribute edge1 and edge2 using
                  the partitioning vector

Read x
Distribute x using the partitioned edges

Read y
Distribute y using the partitioned nodes

For (i=1; i<maxStep; i++) {

  Compute results p and q using
     the partitioned x, y, edge1 and edge2

     (For each checkpoint) {

        Write results p and q to files
          ordered by global node numbers
     }
 }
```

Figure 6: A sample irregular problem

`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.

Figure 7 shows how the data and index partitioning can be specified in SDM. 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 of SDM. Figure 8 shows the use of SDM for writing the results of the computation.

3.2.2 Implementation Details

Besides providing a high-level API for convenient data retrieval and I/O, SDM provides a capability to partition the index and data arrays being used in irregular applications. To use the SDM partitioning scheme, users must provide a *partitioning vector* in which each value of the vector denotes a processor rank where the node should be assigned. The partitioning vector should be replicated among processes. The current implementation of SDM requires that the partition vector fit in memory; an out-of-core partition vector is not currently supported. For the application illustrated in Figure 6, we used the partitioning vector generated from the graph-partitioning tool Metis [20, 34].

The SDM partitioning phase generates *map arrays* that specify the mapping of each element of the local array to the global array. The map arrays are used in I/O operations. This partitioning phase is optional; users can instead provide their own map array, while bypassing this phase.

For irregular applications, in order to partition index and data arrays and to perform I/O, SDM cre-

```
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, vector);

/* 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);
```

Figure 7: Using SDM for index and data partitioning in the irregular application of Figure 6

ates six database tables: *run_table*, *access_pattern_table*, *execution_table*, *import_table*, *index_table*, and *index_history_table*. The first three tables are the same as described in Section 3.1.2 for regular applications.

In order to optimize the cost of index distributions, SDM provides a history file to store the local index subsets of all processes. The associated metadata is stored in the database. When the same index distribution is needed later, each process can retrieve the partitioned index values from the history file by using the metadata stored in the database. This saves the cost of creating the partitions each time.

The disadvantage of the history file is that it cannot be used if the problem runs on a different number of processes from that when the file was created, because the edges and nodes being assigned to each process dynamically change among different numbers of processes. 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 to reduce communication and computation costs.

Figure 7 describes the steps in SDM to partition the indexes and data. The four arrays, `edge1`, `edge2`, `x`, and `y`, are imported by creating a data group. The function `SDM_import` first accesses the index_table in the database to see whether a history file exists with this problem size. If so, the metadata, such as each process's partitioned index size and the name of the history file, is retrieved from the index_table and index_history_table. Otherwise, the desired data is imported to the application. Since `edge1` and `edge2`

```
SDM_initialize(ApplicationName);
result = SDM_make_datalist(2, {p, q});
result[0].data_type = DOUBLE;
SDM_associate_attributes(2, &result[0]);
handle = SDM_set_attributes(result, 2);
......
/* Partition edge1, edge2, x and y among processes (Figure 7) */
......
SDM_data_view(handle, 2, 0, &vector, &localNodes);
For (i=1; $i < maxStep$; i++) {
    ......
    Do computation and produce results p and q;
    ......
    For (each checkpoint) {
        SDM_write(handle, p, i, pbuf);
        SDM_write(handle, q, i, qbuf);
    }
}
SDM_finalize(handle, 2);
```

Figure 8: Using SDM for writing results in the irregular application

are being imported in a contiguous way, there is no need to specify data mapping between the file and processor memory.

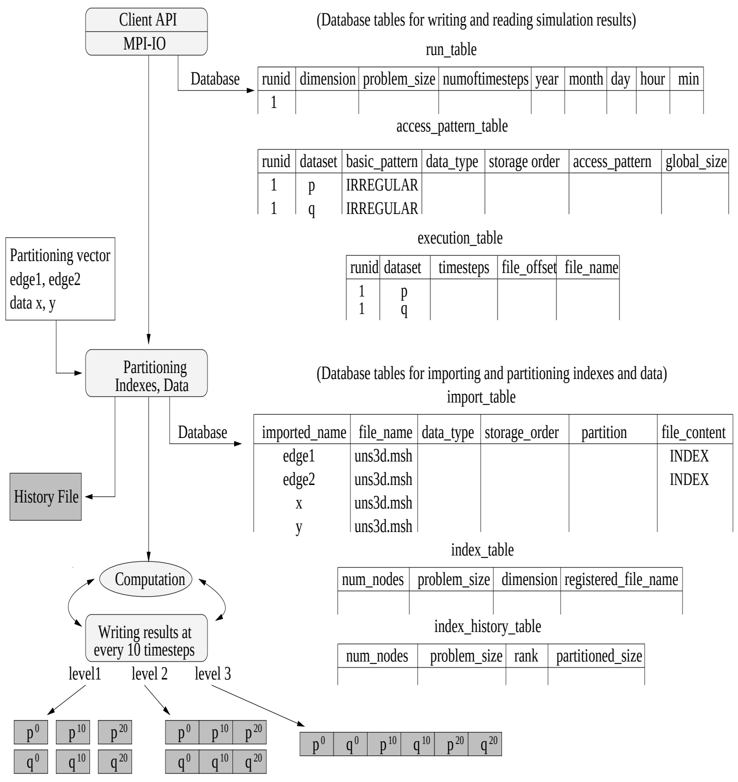
In `SDM_partition_table`, the global partitioning vector (`partitioning_vector` in Figure 7) is converted to the local vector (`vector` in Figure 7) to determine which node should be assigned to which process.

If there is a history file for this problem size, `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 approach avoids the communication cost to exchange each process's edges and the computation cost to choose the edges to be assigned. If there is no history file, the edges in each process are distributed by reading all the data in parallel and by performing a ring-oriented communication.

For storing the partitioned edges and nodes, including the ghost ones, a certain amount of memory space is initially allocated to each process. When the entire memory space is occupied by the partitioned data, it is automatically doubled by adjusting the memory size. This prevents the system from looking through the entire data in two steps, one step to decide the size of memory space and the other step to actually store the data in the memory space. After the edges and nodes are distributed, the edges in each process are moved to the next process in the virtual ring. As a result, two map arrays, `partitioned_edge` and `vector`, are generated that are used to distribute the physical data associated with each edge and node, respectively.

`SDM_index_registry` creates a history file containing the index distribution if no history file had been created earlier for the index distribution. The information to partition index arrays, such as the partition size for each process, is stored in the database tables `index_table` and `index_history_table`. Also, the partitioned index values (`partitioned_edge` in our example) are written asynchronously to a history file to be retrieved in subsequent runs requiring the same edge distribution.

Figure 8 shows the steps to write two data sets, `p` and `q`, after completing the computations at each checkpoint. Before writing `p` and `q`, the data mapping is defined in the `SDM_data_view` by using the map array (`vector`) associated with the node partition. The data sets are written to files according to one of the file organizations described in Section 3.1.3. All the I/O is performed by using MPI-IO's collective I/O



Client API
MPI-IO
(Database tables for writing and reading simulation results)
run_table
Database
runid dimension problem_size numoftimesteps year month day hour min
1
access_pattern_table
runid dataset basic_pattern data_type storage order access_pattern global_size
1 p IRREGULAR
1 q IRREGULAR
execution_table
runid dataset timesteps file_offset file_name
1 p
1 q
Partitioning vector
edge1, edge2
data x, y
Partitioning Indexes, Data
(Database tables for importing and partitioning indexes and data)
import_table
Database
imported_name file_name data_type storage_order partition file_content
edge1 uns3d.msh INDEX
edge2 uns3d.msh INDEX
x uns3d.msh
y uns3d.msh
History File
index_table
num_nodes problem_size dimension registered_file_name
index_history_table
num_nodes problem_size rank partitioned_size
Computation
Writing results at every 10 timesteps
level1
level 2
level 3
p0 p10 p20
q0 q10 q20
p0 p10 p20
q0 q10 q20
p0 q0 p10 q10 p20 q20

Figure 9: SDM execution flow to solve the example in Figure 6

functions and derived datatypes to describe noncontiguous access patterns.

Figure 9 depicts the metadata storage in the database and the organization of data in files in SDM for the example in Figure 6.

4 Performance Results

We used the IBM SP and SGI Origin2000 at Argonne National Laboratory for all our performance tests. At the time the experiments were run, these machines were configured as follows. The IBM SP had 80 compute nodes and 4 I/O nodes. Each I/O node controlled four SSA disks, each of 9 Gbyte capacity. The parallel file system on the machine was IBM's PIOFS [18]. The SGI Origin2000 had 128 processors and 10 Fibre Channel controllers connected to a total of 110 disks, each of 9 Gbyte capacity. The file system on the Origin2000 was SGI's XFS [16, 38].

4.1 Performance Results for Regular Applications

For performance evaluation of regular applications, we used an optimization called direct I/O on XFS. 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 thus eliminates an extra memory copy into the cache and can perform well if the I/O size is large and the machine has a high-bandwidth I/O system. Direct I/O can be used from an MPI-IO program: the ROMIO

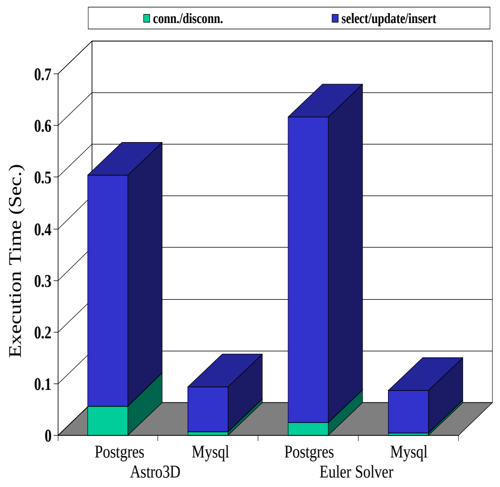


Figure 10: Cost of accessing the database for the two applications on the SGI Origin2000

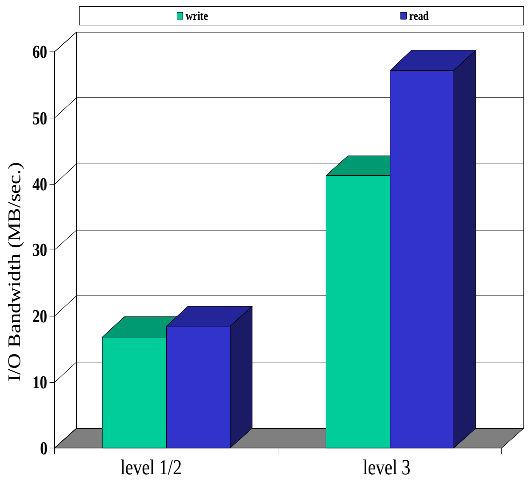


Figure 11: I/O bandwidth for ASTRO3D on the IBM SP

implementation of MPI-IO that we used supports direct I/O [43]. We present performance results with both direct I/O and regular (buffered) I/O.

We used two regular application templates, ASTRO3D and a three-dimensional Euler solver, in our performance experiments. For ASTRO3D, we used a problem size of $256 \times 256 \times 256$. We ran the program for one time step and performed the data analysis, restart, and visualization dumps at that time step. This resulted in a total of around 880 Mbytes of data.

The second application is a three-dimensional Euler solver for the problem of three-dimensional transonic flow about an M6 wing [12]. 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 $194 \times 34 \times 34$.

4.1.1 Cost of Database Access

SDM uses TCP/IP to connect to the database servers. We performed our experiments with two different databases, MySQL [25] and PostgreSQL [31]. Figure 10 shows the database-access cost in the SDM write operation on the Origin2000. The connection to and disconnection from 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. In the write operation, process 0 stores the file offset in the `execution_table`. Access to the `execution_table` occurred 19 times in ASTRO3D versus 60 times in the Euler solver. As can be seen in Figure 10, the database-access cost using either of the database servers was 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 for the rest of the performance experiments.

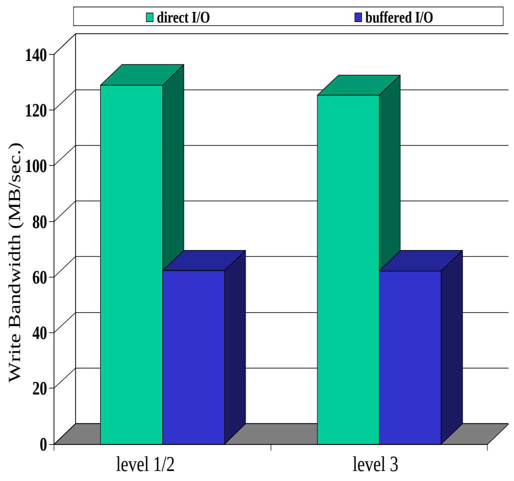


Figure 12: Write bandwidth for ASTRO3D on the SGI Origin2000

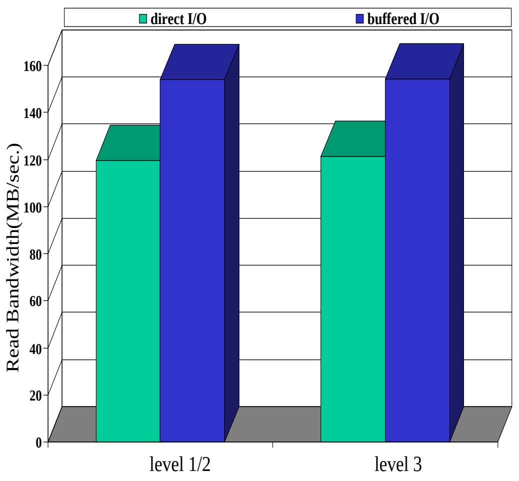


Figure 13: Read bandwidth for ASTRO3D on the SGI Origin2000

4.1.2 Results for ASTRO3D

Figure 11 shows the write and read bandwidths for ASTRO3D on the IBM SP using 32 processors for the three levels of file organization. Since we ran only one iteration of the program, levels 1 and 2 resulted in the same file organization. Level 3 achieved much higher bandwidth because only three different files were created, and, therefore, only three file opens occurred. The high cost of file opens on the PIOFS file system [40] resulted in lower performance for levels 1 and 2, where 19 separate files were created. The impact of file-open time can indeed be quite large.

Figures 12 and 13 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; therefore, 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, but 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 performed nearly the same.

4.1.3 Results for the Euler Solver

Figure 14 shows the write and read bandwidths for the Euler solver on the IBM SP using 32 processors. The total data 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 Figure 14 shows, level 3 performed the best because of the high open cost on PIOFS. In level 1, the file-open cost took around 80% of the total

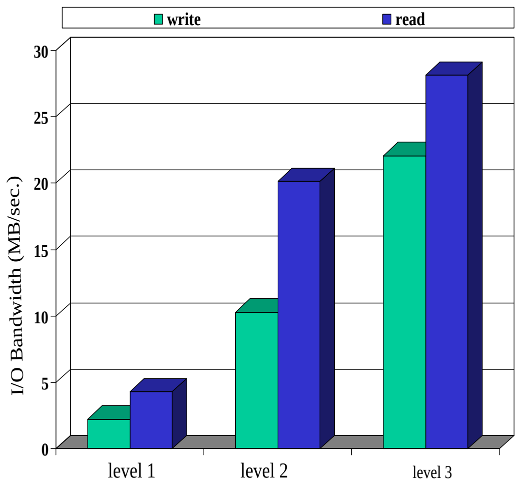


Figure 14: I/O bandwidth for the Euler solver on the IBM SP

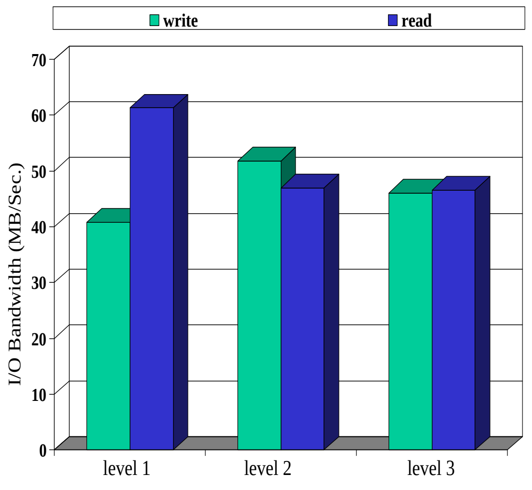


Figure 15: I/O bandwidth for the Euler solver on the SGI Origin2000

execution time; in level 2, it took around 30%; and in level 3, it took around 20% of the total execution time.

Figure 15 shows 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 [30]) 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 a greater number of files.

4.2 Performance Results for Irregular Applications

We obtained performance results for irregular applications on the SGI Origin2000 at Argonne National Laboratory. The first application template that we benchmarked was a tetrahedral vertex-centered unstructured grid code called FUN3D developed by W. K. Anderson of the NASA Langley Research Center [1]. This application uses a partitioning vector generated from METIS to partition the nodes and edges in a mesh. The mesh we used had about 18 million edges and 2 million nodes. At the initial stage, the application imports edges, four data arrays associated with edges, and another four data arrays associated with nodes. The total imported data size was about 807 Mbytes. As a result of computations, the application wrote about 21 Mbytes of four data sets each and 105 Mbytes of a single data set. Using 64 processors, we iterated the application template for two time steps; at each time step, five data sets were written to files.

The second application template that we ran was a Rayleigh-Taylor instability application [11] that is motivated by a joint project between the University of Chicago and Argonne to study thermonuclear flashes on astrophysical objects. Whenever the simulation time reaches a certain point, the application writes two data sets: a single node data set associated with vertices in a mesh, and a triangle data set associated with triangles on tetrahedral faces. In the application template, we wrote about 36 Mbytes of the node data set and about 74 Mbytes of the triangle data set at each time step. Since we iterated the template five times, the total data size written was approximately 550 Mbytes.

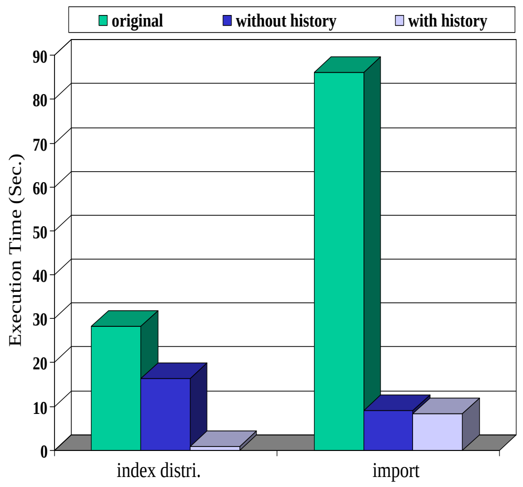


Figure 16: Execution time for partitioning indices and data in FUN3D on SGI Origin2000

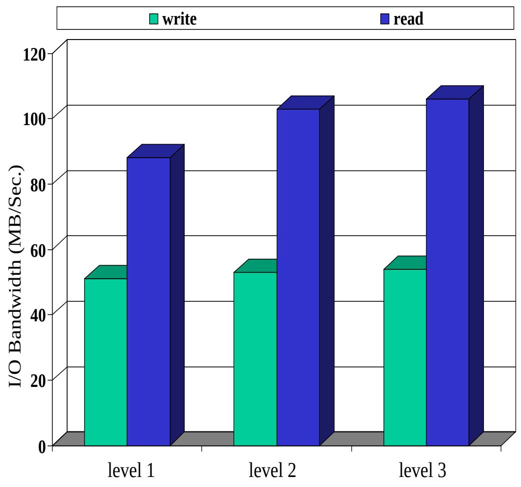


Figure 17: I/O bandwidth for reading and writing data in FUN3D on SGI Origin2000

4.2.1 Results for FUN3D

Figure 16 shows the bandwidth to import and partition 18 million edges, four data sets each of 144 Mbytes of data associated with edges, and another four data sets each of 21 Mbytes of data associated with nodes. The original version of the application—without using SDM—performs all the I/O operations by a single process (process 0), which then broadcasts data to other processes. SDM performs I/O in parallel from all processes using MPI-IO. The bar labeled `index distri` in Figure 16 shows the communication and computation costs to partition the edges after importing them to the application. The bar labeled `import` shows the cost of reading the edges and eight data arrays.

The original application reads the edges in two steps: one step to determine the amount of memory to store the partitioned edges and the other step to actually read the edges. SDM, however, extends the allocated memory dynamically as needed (using 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. Since the history file contains the already partitioned edges, there is no need to import the edges; hence, the read cost in `import` is reduced.

Figure 17 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 is written to separate files, resulting in the creation of 10 different files. Each time the data array is written to files, level 1 requires the cost for opening a file and defining an MPI-IO *file view* to access the data from the portion of the file pointed by the global file offset. In level 2, however, each data array generated at each time step is appended in five files, generating five file-open and file-view costs. This reduced number of files improves the I/O performance slightly. In level 3, only two files are generated, resulting in the best I/O performance among the three file organizations. The difference between three file organizations is not significant because the file-open cost is small on the SGI Origin2000.

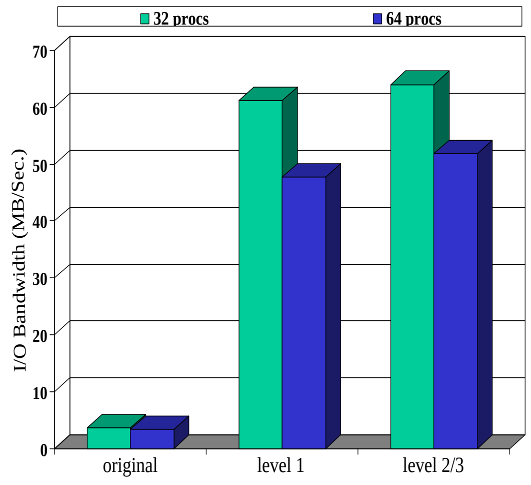


Figure 18: I/O bandwidth for RT on SGI Origin2000

4.2.2 Results of RT Application

Figure 18 shows the I/O bandwidth in the RT application for writing approximately 550 Mbytes of data. In the original application, the write operation is performed sequentially. In other words, after seeking to the starting position in a file, processes write their local portion of data one by one. When we ported the application to SDM, the I/O performance increased significantly because of the parallel I/O optimizations of MPI-IO.

In SDM, we wrote the node data set according to the global node number of the partitioned nodes and wrote the triangle data set contiguously. Since two data sets 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). When the number of processors was increased from 32 to 64, but the total data size remained constant, the I/O performance declined. With 32 processors, the data size being written at each time step was about 1 Mbyte for the node data set and 2 Mbytes for the triangle data set. When the number of processors was increased to 64, the size of each I/O operation became smaller, reducing performance.

5 Related Work

Related work in this area falls into the categories of I/O libraries and parallel file systems for high-performance I/O and other libraries that aim to provide data-management capabilities.

Several efforts have sought to optimize I/O in parallel file systems and runtime libraries [4, 6, 7, 17, 21, 23, 27, 35, 39]. SDM uses MPI-IO and parallel file systems to access real data, and therefore SDM is a consumer of the results of such research efforts as they become available through MPI-IO implementations.

Other efforts provide data-management capability. For example, HDF [44] and NetCDF [26] are popular libraries for data management in scientific applications, and they provide a self-describing data format. SRB (Storage Resource Broker) [3] provides uniform interface to access various storage systems, such as file systems, Unitree, HPSS and database objects. Shoshani et al. [36, 37] describe an architecture to store large volumes of scientific data on tertiary storage systems in a way that the cost of data retrieval is minimized. The Active Data Repository [22] and DataCutter [5] optimize storage, retrieval, and processing of very large multidimensional datasets.

The main differences between our work and the above efforts are that we strive for performance in addition to flexibility and that we aim to be general purpose, not tied to a particular application or particular class of applications. The technique we use of separating data and metadata and storing the data in a parallel file system and metadata in a database could be used to implement other libraries, such as HDF and NetCDF, and we plan to investigate that in the future.

6 Conclusions

We have presented the design and implementation of an environment for high-performance scientific data management, called Scientific Data Manager (SDM), that 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 transparently 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 affect 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 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 performed much better than buffered I/O by avoiding the overhead of copying the data into the XFS buffer cache and also because XFS allows direct writes to proceed concurrently. For reading data, however, buffered I/O performed better because of its read-ahead policy.

We have also presented the SDM API and architecture for I/O in irregular applications. Besides providing an easy-to-use user interface for managing large data sets, SDM uses the concept of a history file to optimize the cost of the index distribution. We studied the performance of SDM using two irregular applications: FUN3D and RT. When we used SDM in both applications, there was a significant improvement in I/O performance compared with that of the original applications. Also, we observed that using a history file for the index distribution helped to reduce the computation and communication costs.

In the future, we plan to use SDM with more applications and evaluate both the usability and performance. We plan to refine the SDM API so that it can be used on a wide range of applications. We plan to study and improve the scalability of the system, particularly the scalability of the metadata. We also plan to investigate how our implementation approach could be used to implement HDF and NetCDF and whether such an approach would improve performance and flexibility of those libraries.

Acknowledgments

We thank Lois Curfman McInnes, Lori Freitag, and Dinesh Kaushik at Argonne National Laboratory for their invaluable help in providing the applications used for the performance evaluation of SDM.

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 *Proceedings of SC99: High Performance Networking and Computing*, November 1999.

[2] Applications Working Group of the Scalable I/O Initiative. Preliminary Survey of I/O Intensive Applications. `http://www.cacr.caltech.edu/SIO/pubs/SIO_apps.ps`, 1994.

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] Juan Miguel del Rosario and Alok Choudhary. High Performance I/O for Parallel Computers: Problems and Prospects. *Computer*, 27(3):59–68, March 1994.

[10] 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.

[11] 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.

[12] 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 ANL/MCS-P788-0100, 1998.

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

[14] 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.

[15] High Performance Fortran Forum. High Performance Fortran Language Specification, Version 1.0. Technical Report Version 1.0, Rice University, Houston, TX, May 1993.

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

[17] 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.

[18] IBM Corp. IBM AIX Parallel I/O File System: Installation, Administration, and Use. Document Number SH34-6065-01, August 1995.

[19] 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.

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

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

[22] 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.

[23] 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.

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

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

[26] NetCDF. `http://www.unidata.ucar.edu/packages/netcdf`.

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

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

[29] Jaechun No, Rajeev Thakur, Dinesh Kaushik, Lori Freitag, and Alok Choudhary. A Scientific Data Management System for Irregular Applications. In *Proceedings of IPDPS'01*, April 2001.

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

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

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

[33] 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.

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

[35] 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.

[36] 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.

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

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] The NCSA HDF home page. `http://hdf.ncsa.uiuc.edu`.