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 Elec. and Computer Eng.
Northwestern University
Evanston, IL 60208
choudhar@ece.nwu.edu

Abstract

Many scientific applications are I/O intensive and generate 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 have 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 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.*

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, 25]. Many of these applications perform their computation and I/O on an irregularly discretized mesh. The data accesses in those applications make extensive use of arrays, called indirection array [7, 24] or map array [10], 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 [11, 12] or by using a runtime library [7, 24]. 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.

We have developed a software system for large-scale scientific data management, called Scientific Data Manager (SDM) [23], that combines 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 [23], 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 problems. In Section 3 we present a typical irregular problem and describe the detailed implementation issues of

*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.

SDM to solve the problem. Performance results on the SGI Origin2000 at Argonne National Laboratory are presented in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2. Design Objectives

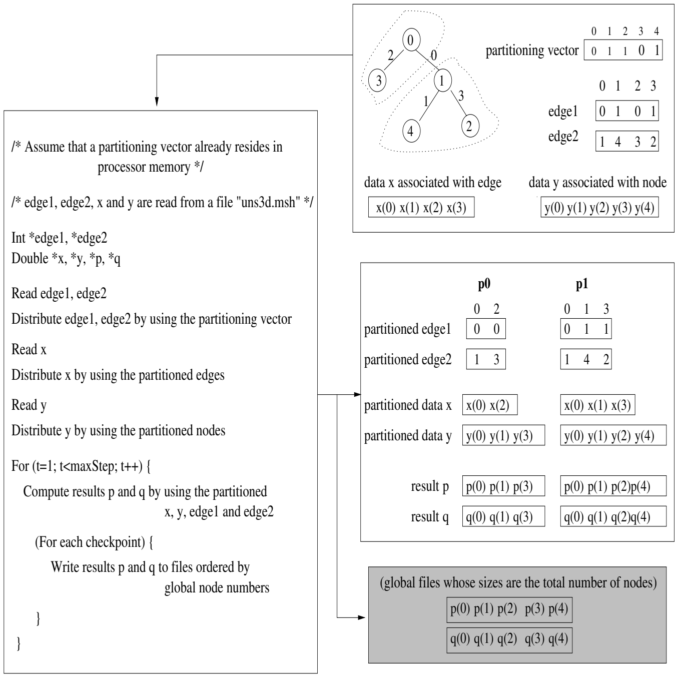
Our main objectives in designing SDM for irregular applications were to achieve high-performance parallel I/O, to provide a convenient high-level API, and to optimize the execution cost of irregular applications.

- **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 [10, 19], 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, 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 databases. With SDM, 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 [32]. 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. In SDM, after partitioning the index values among processes, the local index subsets of all processes are asynchronously written to a *history file*, and the associated metadata is stored in 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 database, and thereby the user can avoid repeating the communication and computation for the same index distribution.

3. Implementation

We discuss the SDM API for solving a sample irregular problem and show how the API is implemented.

3.1. An Irregular Problem and SDM API



```
/* 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, edge2 by using the partitioning vector
Read x
Distribute x by using the partitioned edges
Read y
Distribute y by using the partitioned nodes

For (t=1; t<maxStep; t++) {
  Compute results p and q by using the partitioned
                    x, y, edge1 and edge2
    (For each checkpoint) {
      Write results p and q to files ordered by
                    global node numbers
    }
}
```

Figure 1. A sample irregular problem and its solution

Figure 1 shows 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 and 3 respectively 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 described 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 of SDM.

3.2. Implementation Details

The *partitioning vector* is the one generated from a partitioning tool, such as MeTis[15, 26]. Each value of the vector denotes a processor rank where the node should be assigned. In SDM, the partitioning vector should be replicated among processes. Next, the *map array* is the one that

```
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
        (Figure 3) */
......
SDM_data_view(handle, 2, p, &vector, &localNodes);
For (t=1; 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);
```

Figure 2. SDM API for writing results

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 the map array can be specified by the user.

Figure 2 shows the steps involved in initializing SDM to solve the problem in Figure 1. Running the problem on SDM begins by calling the *SDM_initialize* to establish database connection (for storing metadata). Six database tables, *run_table*, *access_pattern_table*, *execution_table*, *import_table*, *index_table*, and *index_history_table*, are created to store the metadata associated with the application. Since two data sets, p and q, are produced as a result of computations and they have the same data type and global size, these data sets are grouped in a data group to experiment different ways of organizing data in files. All the metadata associated with these data sets are stored in a database in the *SDM_set_attributes*.

Figure 3 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. Since these arrays have been created outside of SDM, the user has no control over the arrays except to read them, by specifying their data type, appropriate file offset, and length. The user need not create several data groups to import the arrays. In the *SDM_make_importlist*, the metadata of this imported data group, including a mechanism for the import (partition), is stored in the *import_table* for a later use.

```
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 3. SDM API for partitioning indexes and data

In order to partition edge1 and edge2, the *SDM_import* is called to import the arrays with the parameters of file handle, their position in the data group, file offset, file length, and user buffer to hold the data. The *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 history file name, is retrieved from the *index_table* and *index_history_table*, and the control exits the *SDM_import*. Otherwise, the desired data is imported to the application. Since edge1 and edge2 are being imported in a contiguous way, there is no need to specify data mapping between the file and processor memory. In the *SDM_import*, the total domain (file length) is equally divided among processes, and the data in the domain is contiguously imported into the application. In our example, edges 0 and 1 are imported to process 0, and edges 2 and 3 are imported to process 1.

In the *SDM_partition_table*, the global partitioning vector, partitioning_vector in Figure 3, is converted to the local vector, vector in Figure 3, to determine which node should be assigned to which process. In the example, nodes 0 and 3 are assigned to process 0, and nodes 1, 2, and 4 are assigned to process 1.

If there is a history file for this problem size, the *SDM_partition_index* reads the already partitioned edge1 and edge2 from the history file and converts them to the localized edges by using the partitioning vector. This 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, 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. 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 a node of an edge has been partitioned to a process, the edge is assigned to the process. For example, edge 0 is assigned both to process 0 and 1 because one node of the edge, edge1 0, has been partitioned to process 0 and the other node, edge2 1, has been partitioned to process 1. This edge is a ghost edge of both processes being stored to minimize communication volumes.

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 located at a ring network. In the example, process 0 receives edges 2 and 3, and process 1 receives edges 0 and 1 to partition them as described above. After finishing the edge distribution, edges 0 and 2 are assigned to process 0, and edges 0, 1, and 3 are assigned to process 1. Similarly, nodes 0, 1, and 3 are assigned to process 0, and nodes 0, 1, 2, and 4 are assigned to process 1. In Figure 3, partitioned_edge contains the edges assigned to each process, and vector contains the nodes assigned to it. These are the two map arrays to distribute the physical data associated with each edge and node, respectively.

If the *SDM_index_registry* was executed for the first time and no history file was created earlier, the metadata of the partitioned edges, such as the partitioned size of each process, is stored in the database tables *index_table* and *index_history_table*. Also, 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. If the user does not call the *SDM_index_registry*, no history file is created after partitioning the edges.

In order to import and partition data x and y in the *SDM_import*, the *SDM_data_view* must be called to define the data mapping between a noncontiguous global view of the file and a local view of the processor memory. Using the data mapping, in the *SDM_import*, the associated data is irregularly distributed by calling a collective MPI-IO function. In the *SDM_release_importlist*, the structures being used to import data in the file handle are free.

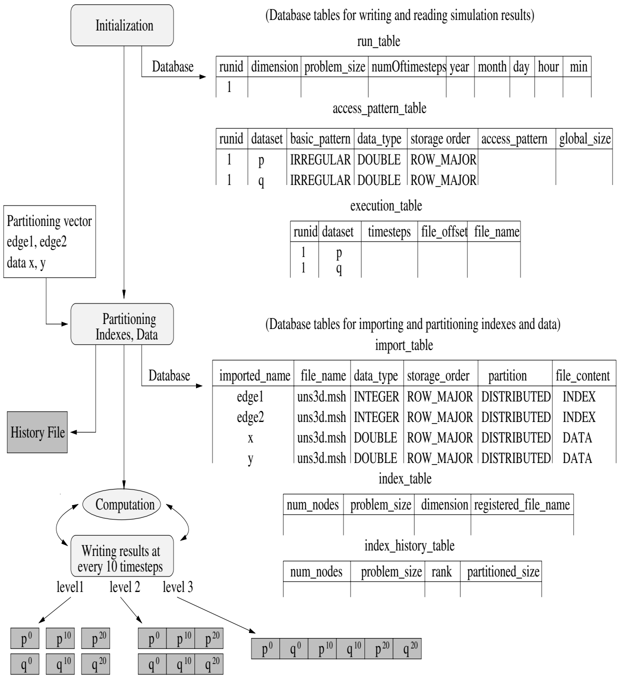
Figure 2 shows the steps to write two data sets, p and q, after completing the computations at each checkpoint. Before writing p and q, the data mapping to write is defined in the *SDM_data_view* using the map array (vector) associated with the node partition.

SDM supports three different ways of organizing data in files. In level 1, each data set generated at each time step is written to a separate file. This file organization is simple, but it incurs the cost of a file-open, file-view to define the visible portion of a file for each process and a file-close at each time step. 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 method results in a smaller number of files and smaller file-open and file-view 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 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 file-open and file-close costs, and an application generates a high file-view cost, as in irregular applications, SDM can generate a very small number of files. However, 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 degrade the performance.

Figure 4 depicts the metadata storage in the database and the organization of data in files in SDM for the example in Figure 1.

4. Performance Results

We obtained performance results on the SGI Origin2000 at Argonne National Laboratory. The Origin2000 has 128



(Database tables for writing and reading simulation results)

run_table

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	DOUBLE	ROW_MAJOR		
1	q	IRREGULAR	DOUBLE	ROW_MAJOR		

execution_table

runid	dataset	timesteps	file_offset	file_name
1	p			
1	q			

(Database tables for importing and partitioning indexes and data)

import_table

imported_name	file_name	data_type	storage_order	partition	file_content
edge1	uns3d.msh	INTEGER	ROW_MAJOR	DISTRIBUTED	INDEX
edge2	uns3d.msh	INTEGER	ROW_MAJOR	DISTRIBUTED	INDEX
x	uns3d.msh	DOUBLE	ROW_MAJOR	DISTRIBUTED	DATA
y	uns3d.msh	DOUBLE	ROW_MAJOR	DISTRIBUTED	DATA

index_table

num_nodes	problem_size	dimension	registered_file_name

index_history_table

num_nodes	problem_size	rank	partitioned_size

Figure 4. SDM execution flow to solve for the example in Figure 1

processors and 10 Fibre Channel controllers connected to a total of 110 disks of 9 GBytes capacity each. The file system on the Origin2000 is SGI's XFS [13, 30]. For the results, we used XFS buffered I/O and MySQL [20] to store the metadata.

The first application template that we benchmarked was a tetrahedral vertex-centered unstructured grid code 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. To evaluate SDM ported to the application, we used about 18M edges and 2M 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 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 [9] that is motivated by a joint project between the University of Chicago and Argonne to study thermonuclear flashes on astrophysical objects. Whenever the current 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.

4.1. Results for FUN3D

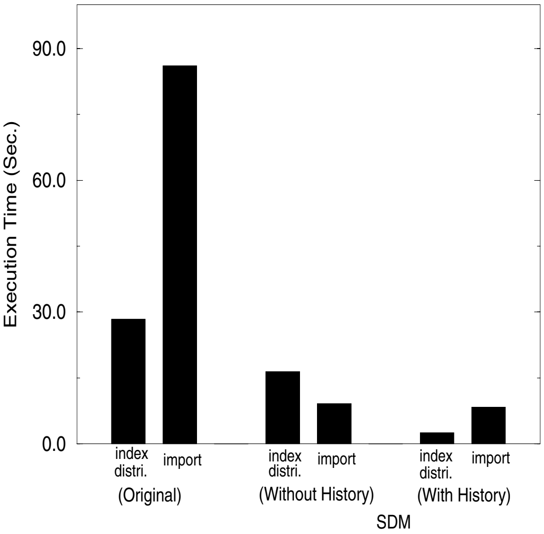


Figure 5. Execution time for partitioning indices and data in FUN3D

Figure 5 shows the bandwidth to import and partition 18M 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 5 shows the communication and computation costs to partition the edges after importing them to the application. Also, 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.

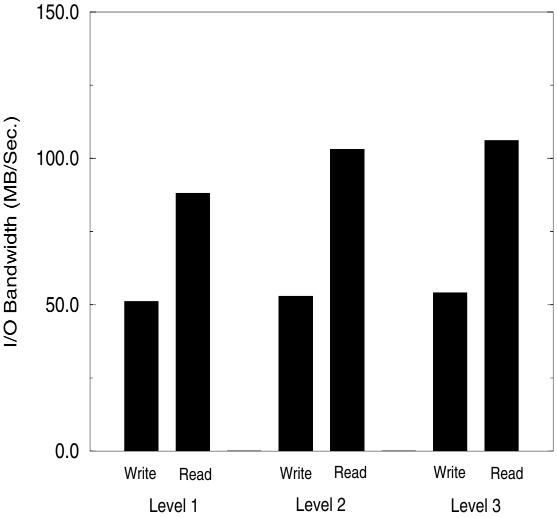


Figure 6. I/O bandwidth for reading and writing data in FUN3D

Figure 6 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. On the SGI Origin2000, the difference between three file organizations is not significant because the file-open cost is small.

4.2. Results of RT Application

Figure 7 shows the I/O bandwidth for writing approximately 550 MBytes of data. In the original application, the write operation is performed sequentially. In other words, after seeking 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 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). As can be seen in Figure 7, on the SGI Origin2000, changing the file organization does not affect the I/O performance, since the cost of file-open and file-view is very low.

When the number of processors increases to write the same data size, we can see the degradation of the I/O performance. With 32 processors, the data size being written at each time step is about 1 MByte for the node data set and 2 MBytes for the triangle data set. If the number of processors goes up to 64, the buffer size of each process becomes smaller, resulting in the performance reduction. Clearly, there is an optimal buffer size that shows the best I/O performance.

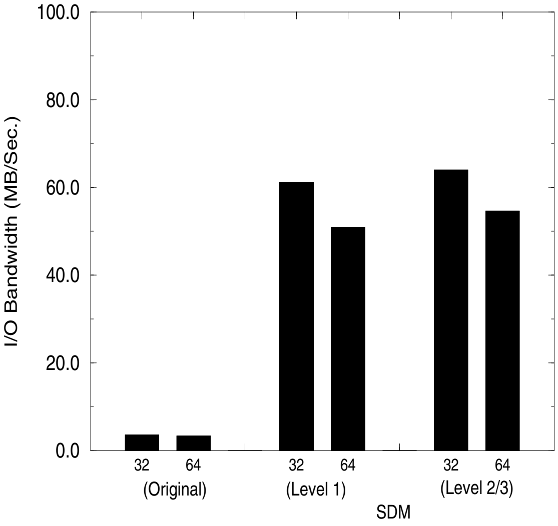


Figure 7. I/O bandwidth for RT

5. Related Work

Several efforts have sought to optimize I/O in parallel file systems and runtime libraries [3, 5, 6, 14, 16, 18, 22, 27, 31]. SRB (Storage Resource Broker) [2] provides an 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. [28, 29] describe an architecture for optimizing access to large volumes of scientific data stored on tapes. The Active Data Repository [17] and DataCutter [4] optimize storage, retrieval, and processing of very large multidimensional datasets. The main difference between our work and other efforts in I/O is that SDM 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 to 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 [21] and netCDF [33].

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] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC Storage Resource Broker. In *Proceedings of CASCON '98*, December 1998.

[3] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz. Jovian: A Framework for Optimizing Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–20. IEEE Computer Society Press, Oct. 1994.

[4] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. 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] R. Bordawekar, J. M. del Rosario, and A. Choudhary. Design and Evaluation of Primitives for Parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, November 1993.

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

[7] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[8] J. M. del Rosario and A. Choudhary. High performance I/O for parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[9] 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.

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

[11] R. V. Hanxleden, K. Kennedy, and J. 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.

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

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

[14] J. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. 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.

[15] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *Journal on Scientific Computing*, 1997.

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

[17] T. Kurc, C. Chang, R. Ferreira, A. Sussman, and J. Saltz. Querying Very Large Multi-dimensional Datasets in ADR. In *Proceedings of SC99: High Performance Networking and Computing*, November 1999.

[18] T. M. Madhyastha and D. 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.

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

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

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

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

[23] J. No, R. Thakur, and A. 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.

[24] 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.

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

[26] K. Schloegel, G. Karypis, and V. Kumar. Graph Partitioning for High Performance Scientific Simulations. 2000.

[27] 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.

[28] 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.

[29] 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.

[30] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proc. of USENIX 1996 Annual Technical Conference*, San Diego, CA, January 1996.

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

[32] R. Thakur, W. Gropp, and E. 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.

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