Querying Very Large Multi-dimensional Datasets in ADR - Extended Abstract *

Tahsin Kurc†, Chialin Chang†, Renato Ferreira†, Alan Sussman†, Joel Saltz†+

† Institute for Advanced Computer Studies and Dept. of Computer Science University of Maryland College Park, MD 20742

+ Dept. of Pathology Johns Hopkins Medical Institutions Baltimore, MD 21287

{kurc,chialin,renato,als,saltz}@cs.umd.edu

1 Introduction

Analysis and processing of very large multi-dimensional scientific datasets (i.e. where data items are associated with points in a multi-dimensional attribute space) is an important component of science and engineering. Moreover, an increasing number of applications make use of very large multi-dimensional datasets. Examples of such datasets include raw and processed sensor data from satellites [12], output from hydrodynamics and chemical transport simulations [10], and archives of medical images [1].

Many applications that make use of multi-dimensional datasets have several important characteristics. Both the input and the output are often disk-resident datasets. Applications may use only a subset of all the data available in input and output datasets. Access to data items is described by a *range query*, namely a multi-dimensional bounding box in the underlying multi-dimensional attribute space of the dataset. Only the data items whose associated coordinates fall within the multi-dimensional box are retrieved. The processing structures of these applications also share common characteristics. Figure 1 shows high-level pseudo-code for the basic processing loop in these applications. The processing steps consist of retrieving input and output data items that intersect the range query (steps 1–2 and 4–5), mapping the coordinates of the retrieved input items to the corresponding output items (step 6), and aggregating, in some way, all the retrieved input items mapped to the same output data items (steps 7–8). Correctness of the output usually does not depend on the order input data items are aggregated. The mapping function, $Map(i_e)$, maps an input item to a set of output items. An intermediate data structure, referred to as an *accumulator*, is used to hold intermediate results during processing. For example, an accumulator can be used to keep a running sum for an averaging operation. The aggregation function, $Aggregate(i_e, a_e)$, aggregates the value of an input item with the intermediate result stored in the accumulator element (a_e). The output dataset from a query is usually much smaller than the input dataset, hence steps 4–8 are called the *reduction* phase of the processing. Accumulator elements are allocated and initialized (step 3) before the reduction phase. Another constraint is that there is

*This research was supported by the National Science Foundation under Grants #BIR9318183 and #ACI-9619020 (UC Subcontract # 10152408), and the Office of Naval Research under Grant #N6600197C8534. The Maryland IBM SP2 used for the experiments was provided by NSF CISE Institutional Infrastructure Award #CDA9401151 and a grant from IBM.

$O \leftarrow$ Output Dataset, $I \leftarrow$ Input Dataset
(* Initialization *)
1. **foreach** o_e **in** O **do**
2. **read** o_e
3. $a_e \leftarrow Initialize(o_e)$
(* Reduction *)
4. **foreach** i_e **in** I **do**
5. **read** i_e
6. $S_A \leftarrow Map(i_e)$
7. **foreach** a_e **in** S_A **do**
8. $a_e \leftarrow Aggregate(i_e, a_e)$
(* Output *)
9. **foreach** a_e **do**
10. $o_e \leftarrow Output(a_e)$
11. **write** o_e

Figure 1: The basic processing loop in the target applications.

a one-to-one mapping between output items and accumulator elements. The intermediate results stored in the accumulator are post-processed to produce final results (steps 9–11).

Typical examples of application classes that make use of multi-dimensional scientific datasets are satellite data processing applications [15, 5], the Virtual Microscope and analysis of microscopy data [1], and simulation systems for water contamination studies [10]. Due to limited space, we briefly describe the satellite data processing application here. In satellite data processing, earth scientists study the earth by processing remotely-sensed data continuously acquired from satellite-based sensors. Each sensor reading is associated with a position (longitude and latitude) and the time the reading was recorded. In a typical analysis [15], a range query defines a bounding box that covers a part or all of the surface of the earth over a period of time. Data items retrieved from one or more datasets are processed to generate one or more composite images of the area under study. Generating a composite image requires projection of the selected area of the earth onto a two-dimensional grid [18]; each pixel in the composite image is computed by selecting the "best" sensor value that maps to the associated grid point. Composite images can be added to the database as new datasets.

We have developed an infrastructure, called the Active Data Repository (ADR) [3], that integrates storage, retrieval and processing of large multi-dimensional datasets on distributed memory parallel architectures with multiple disks attached to each node. ADR targets applications with the processing structure shown in Figure 1. ADR is designed as a set of modular services implemented in C++. Through use of these services, ADR allows customization for application specific processing (i.e. the $Initialize$, Map, $Aggregate$, and $Output$ functions), while providing support for common operations such as memory management, data retrieval, and scheduling of processing across a parallel machine. The system architecture of ADR consists of a front-end and a parallel back-end. The front-end interacts with clients, and forwards range queries with references to user-defined processing functions to the parallel back-end. During query execution, back-end nodes retrieve input data and perform user-defined operations over the data items retrieved to generate the output products. Output products can be returned from the back-end nodes to the requesting client, or stored in ADR.

Several runtime support libraries and file systems have been developed to support efficient I/O in a parallel environment [2, 6, 8, 9, 13, 14, 16, 17]. ADR differs from these systems in several ways. First, ADR is able to carry out range queries directed at irregular spatially indexed datasets. Second, computation is an integral part of the ADR framework. With the collective I/O interfaces provided by many parallel I/O systems, data processing usually cannot begin until the entire collective I/O operation completes. Third, data placement algorithms optimized for range queries are integrated as part of the ADR framework.

In this work, we discuss optimizing the execution of range queries (i.e. the processing loop shown in Figure 1) on distributed memory parallel machines within the ADR framework. We describe three potential strategies for efficient execution of such queries. We evaluate scalability of these strategies for different application scenarios, varying both the number of processors and the input dataset size, using *application emulators* [19] to generate various application scenarios for the applications classes that motivated the design of ADR. We present experimental evaluation of the three strategies on a 128-node IBM SP.

2 Query Execution in ADR

2.1 Storing Datasets in ADR

A dataset is partitioned into a set of chunks to achieve high bandwidth data retrieval. A chunk consists of one or more data items, and is the unit of I/O and communication in ADR. That is, a chunk is always retrieved as a whole during query processing. As every data item is associated with a point in a multi-dimensional attribute space, every chunk is associated with a minimum bounding rectangle (MBR) that encompasses the coordinates (in the associated attribute space) of all the items in the chunk. Since data is accessed through range queries, it is desirable to have data items that are close to each other in the multi-dimensional space in the same chunk. Chunks are distributed across the disks attached to ADR back-end nodes using a declustering algorithm [7, 11] to achieve I/O parallelism during query processing. Each chunk is assigned to a single disk, and is read and/or written during query processing only by the local processor to which the disk is attached. If a chunk is required for processing by one or more remote processors, it is sent to those processors by the local processor via interprocessor communication. After all data chunks are stored into the desired locations in the disk farm, an index (e.g., an R-tree) is constructed using the MBRs of the chunks. The index is used by the back-end nodes to find the local chunks with MBRs that intersect the range query.

2.2 Query Planning

A plan specifies how parts of the final output are computed and the order the input data chunks are retrieved for processing. Planning is carried out in two steps; *tiling* and *workload partitioning*. In the tiling step, if the output dataset is too large to fit entirely into the memory, it is partitioned into *tiles*. Each tile, O_t, contains a distinct subset of the output chunks, so that the total size of the chunks in a tile is less than the amount of memory available for output data. Tiling of the output implicitly results in a tiling of the input dataset. Each input tile, I_t, contains the input chunks that map to the output chunks in tile, O_t. During query processing, each output tile is cached in main memory, and input chunks from the required input tiles are retrieved. Since a mapping function may map an input element to multiple output elements, an input chunk may appear in more than one input tile if the corresponding output chunks are assigned to different tiles. Hence, an input chunk may be retrieved multiple times during execution of the processing loop. Figure 2 illustrates the processing loop with tiled input and output datasets.

In the workload partitioning step, the workload associated with each tile (i.e. aggregation of items in input and accumulator chunks) is partitioned across processors. This is accomplished by assigning each

(* Output and Input Dataset Tiles *)
$O_{tiles} \leftarrow \{O_t\}$ and $I_{tiles} \leftarrow \{I_t\}$ for $1 \leq t \leq N$
foreach $[O_t, I_t]$ **in** $[O_{tiles}, I_{tiles}]$ **do**
    (* Initialization *)
    foreach o_c **in** O_t **do**
        read o_c
        $a_c \leftarrow Initialize(o_c)$
    (* Reduction *)
    foreach i_c **in** I_t **do**
        read i_c
        $S_A \leftarrow Map(i_c) \cap O_t$
        foreach a_c **in** S_A **do**
            $a_c \leftarrow Aggregate(i_c, a_c)$
    (* Output *)
    foreach a_c **do**
        $o_c \leftarrow Output(a_c)$
        write o_c

Figure 2: Basic processing loop with tiled input and output datasets. N is the total number of tiles after tiling step.

processor the responsibility for processing a subset of the input and/or accumulator chunks.

2.3 Query Execution

The processing of a query on a back-end processor progresses through four phases for each tile:

1. **Initialization**. Accumulator chunks in the current tile are allocated space in memory and initialized. If an existing output dataset is required to initialize accumulator elements, an output chunk is retrieved by the processor that has the chunk on its local disk, and the chunk is forwarded to the processors that require it.

2. **Local Reduction**. Input data chunks on the local disks of each back-end node are retrieved and aggregated into the accumulator chunks allocated in each processor's memory in phase 1.

3. **Global Combine**. If necessary, results computed in each processor in phase 2 are combined across all processors to compute final results for the accumulator chunks.

4. **Output Handling**. The final output chunks for the current tile are computed from the corresponding accumulator chunks computed in phase 3. If the query creates a new dataset, output chunks are declustered across the available disks, and each output chunk is written to the assigned disk. If the query updates an already existing dataset, the updated output chunks are written back to their original locations on the disks.

A query iterates through these phases repeatedly until all tiles have been processed and the entire output dataset has been computed. Note that the reduction phase in Figure 2 is divided into two phases, *local* *reduction* and *global combine*. This is a result of workload partitioning for parallel query execution, as will be discussed in Section 3.

To reduce query execution time, ADR overlaps disk operations, network operations and processing as much as possible during query processing. Overlap is achieved by maintaining explicit queues for each kind of operation (data retrieval, message sends and receives, data processing) and switching between queued operations as required. Pending asynchronous I/O and communication operations in the operation queues are polled and, upon their completion, new asynchronous operations are initiated when more work is expected and memory buffer space is available. Data chunks are therefore retrieved and processed in a pipelined fashion.

3 Query Processing Strategies

In this section we briefly describe three strategies that use different tiling and workload partitioning approaches. More detailed descriptions of these strategies can be found in [4]. We refer to an input/output data chunk stored on one of the disks attached to a processor as a *local chunk* on that processor. Otherwise, it is a *remote* chunk. A processor *owns* an input or output chunk if it is a local input or output chunk. A *ghost chunk* is a copy of an accumulator chunk allocated in the memory of a processor that does not own the corresponding output chunk.

In the tiling phase of all the strategies described in this section, we use a *Hilbert space-filling curve* [7] to create the tiles. The goal is to minimize the total length of the boundaries of the tiles, by assigning chunks that are spatially close in the multi-dimensional attribute space to the same tile, to reduce the number of input chunks crossing tile boundaries. The advantage of using Hilbert curves is that they have good clustering properties [11], since they preserve locality. In our implementation, the mid-point of the bounding box of each output chunk is used to generate a Hilbert curve index. The chunks are sorted with respect to this index, and selected in this order for tiling. The current implementation, however, does not explicitly take into account the mapping between the input and output chunks, and therefore in some cases there can still be many input chunks intersecting multiple tiles, despite a small boundary length.

3.1 Fully Replicated Accumulator (FRA) Strategy

In this scheme each processor performs processing associated with its local input chunks. The output chunks are partitioned into tiles, each of which fits into the available local memory of a single back-end processor. When an output chunk is assigned to a tile, the corresponding accumulator chunk is put into the set of local accumulator chunks in the processor that owns the output chunk, and is assigned as a ghost chunk on all other processors. This scheme effectively replicates all of the accumulator chunks in a tile on each processor, and during the *local reduction* phase, each processor generates partial results for the accumulator chunks using only its local input chunks. Ghost chunks with partial results are then forwarded to the processors that own the corresponding output (accumulator) chunks during the *global combine* phase to produce the complete intermediate result, and eventually the final output product.

3.2 Sparsely Replicated Accumulator (SRA) Strategy

The FRA strategy replicates each accumulator chunk in every processor even if no input chunks will be aggregated into the accumulator chunks in some processors. This results in unnecessary initialization overhead in the *initialization* phase of query execution, and extra communication and computation in the *global combine* phase. The available memory in the system also is not efficiently employed, because of unnecessary replication. Such replication may result in more tiles being created than necessary, which may cause a large number of input chunks to be retrieved from disk more than once.

In SRA strategy, a ghost chunk is allocated only on processors owning at least one input chunk that projects to the corresponding accumulator chunk. The index, which contains the MBRs of all the chunks, is used to decide where ghost chunks need to be allocated. Note that the index is constructed when the dataset is loaded into ADR.

3.3 Distributed Accumulator (DA) Strategy

In this scheme, every processor is responsible for all processing associated with its local output chunks. Tiling is done by selecting from each processor its local output chunks until the memory space allocated for the corresponding accumulator chunks is filled. As in the other schemes, output chunks are selected in Hilbert curve order.

Since no accumulator chunks are replicated by the DA strategy, no ghost chunks are allocated. This allows DA to make more effective use of memory and produce fewer tiles than the other two schemes. As a result, fewer input chunks are likely to be retrieved for multiple tiles. Furthermore, DA avoids interprocessor communication for accumulator chunks during the *initialization* phase and for ghost chunks during the *global combine* phase, and also requires no computation in the *global combine* phase. The FRA and SRA strategies eliminate interprocessor communication for input chunks, by replicating all accumulator chunks. On the other hand, the distributed accumulator strategy introduces communication in the *local reduction* phase for input chunks; all the remote input chunks that map to the same output chunk must be forwarded to the processor that owns the output chunk. Since a projection function may map an input chunk to multiple output chunks, an input chunk may be forwarded to multiple processors. In addition, a good declustering strategy could cause almost all input chunks to be forwarded to other processors, because an input chunk and the output chunk(s) that it projects to are unlikely to be assigned to the same processor.

4 Experimental Results

We present an experimental evaluation of the three query execution strategies on a 128-node IBM SP multicomputer. Each node of the SP is a thin node with 256 MB of memory; the nodes are connected via a High Performance Switch that provides 110MB/sec peak communication bandwidth per node. Each node has one local disk with 500MB of available scratch space. We allocated 225MB of that space for the input dataset and 25MB for the output dataset for these experiments. The AIX filesystem on the SP nodes uses a main memory file cache, so we used the remaining 250MB on the disk to clean the file cache before each experiment to obtain more reliable performance results.

We evaluate the query execution strategies for different application scenarios, varying the number of processors and the input dataset size. We used *application emulators* [19] to generate various application scenarios for the applications classes that motivated the design of ADR (see Section 1). An application emulator provides a parameterized model of an application class; adjusting the parameter values makes it possible to generate different application scenarios within the application class and scale applications in a controlled way. The assignment of both input and output chunks to the disks was done using a Hilbert curve based declustering algorithm [7].

Table 1 summarizes dataset sizes and application characteristics for three application classes; *satellite data processing* (SAT), analysis of microscopy data with the *Virtual Microscope* (VM), and *water contamination studies* (WCS). The output dataset size remained fixed for all experiments. The column labeled *Fan-in* shows the average number of input chunks that map to each output chunk, while the *Fan-out* column shows the average number of output chunks to which an input chunk maps for both the smallest and

App.	Input Dataset Num. of Chunks	Input Dataset Total Size	Output Dataset Num. of Chunks	Output Dataset Total Size	Average Fan-in	Average Fan-out	Computation (in milliseconds) I–LR–GC–OH
SAT	9K – 144K	1.6GB – 26GB	256	25MB	161 – 1307	4.6	1–40–20–1
WCS	7.5K – 120K	1.7GB – 27GB	150	17MB	60 – 960	1.2	1–20–1–1
VM	4K – 64K	1.5GB – 24GB	256	48MB	16 – 128	1.0	1–5–1–1

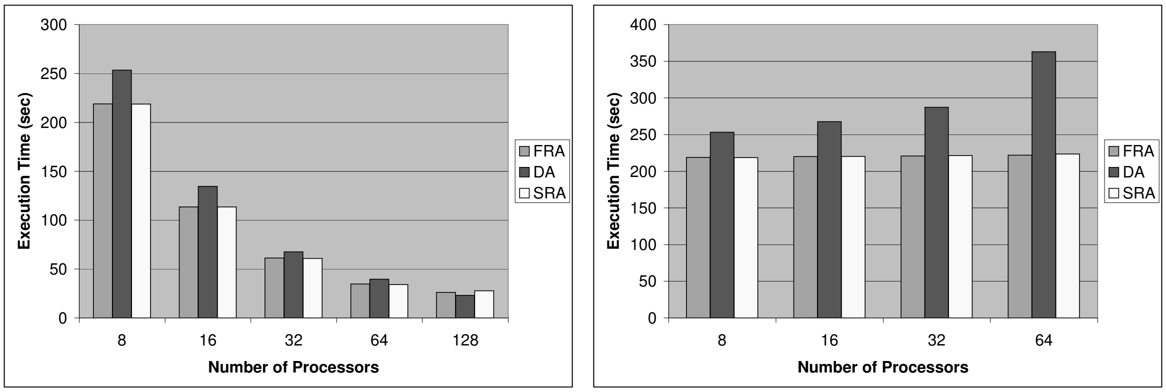
Table 1: Application characteristics.

largest input datasets. The last column shows the computation time per chunk for the different phases of query execution (see Section 2.3); I-LR-GC-OH represents the Initialization-Local Reduction-Global Combine-Output Handling phases. The computation times shown represent the relative computation cost of the different phases within and across the different applications. The LR value denotes the computation cost for each intersecting (input chunk, accumulator chunk) pair. Thus, an input chunk that maps to a larger number of accumulator chunks takes longer to process. In all of these applications the output datasets are regular arrays, hence each output dataset is divided into regular multi-dimensional rectangular regions. The distribution of the individual data items and the data chunks in the input dataset of SAT is irregular. This is because of the polar orbit of the satellite [12]; the data chunks near the poles are more elongated on the surface of the earth than those near the equator and there are more overlapping chunks near poles. The input datasets for WCS and VM are regular dense arrays, which are partitioned into equal-sized rectangular chunks. We selected the values for the various parameters to represent typical scenarios for these application classes on the SP machine, based on our experience with the complete ADR applications.

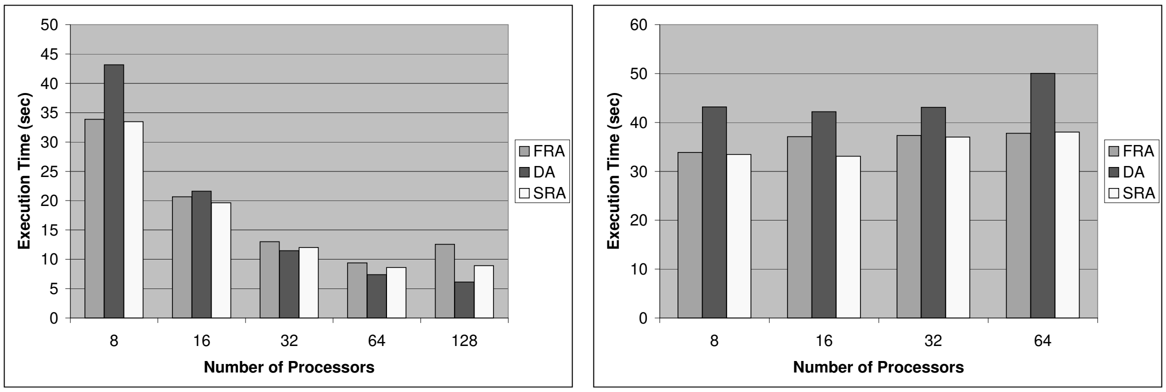
Figure 3 shows query execution times for the different applications. The graphs in the left column display the performance of the different strategies when the input dataset size is fixed, varying the number of processors. As is seen from the figure, execution time decreases with increasing number of processors. The FRA and SRA strategies achieve better performance than the DA strategy on small numbers of processors for the SAT and WCS applications. However, the difference between DA and the other strategies decreases as the number of processors increases. This is because both communication volume and computation time per processor for DA decrease as the number of processors increases, whereas the overheads from the *initialization* and *global combine* phases for FRA and SRA remain almost constant. The graphs in the right column in Figure 3 display query execution time when the input dataset is scaled with the number of processors[1]. As is seen from the figure, execution time increases for DA as the number of processors (and dataset size) increases, whereas it remains almost constant for the FRA and SRA strategies for the SAT and WCS applications. This is because the DA strategy has both higher communication volume and more load imbalance than the FRA and SRA strategies. For VM, we observed a large fluctuation in I/O times across processors, especially for large configurations, even though each processor reads the same amount of data. That is why overall execution time increases, although it should remain approximately the same. We would expect that the DA strategy should achieve better performance for VM, since the computation cost per block in VM is small, and it is a highly regular application with low fan-out of an input block to output blocks.

Figures 4(a)-(d) display the change in volume of communication and computation time per processor for fixed and scaled input datasets on varying number of processors[1]. Note that all strategies read the same amount of data from disks. However, the volume of communication for DA is proportional to the number of input chunks in each processor and the average fan-out of each input chunk, while for FRA it is proportional to the number of total output chunks. As is seen in Figure 4(a), as the number of processors

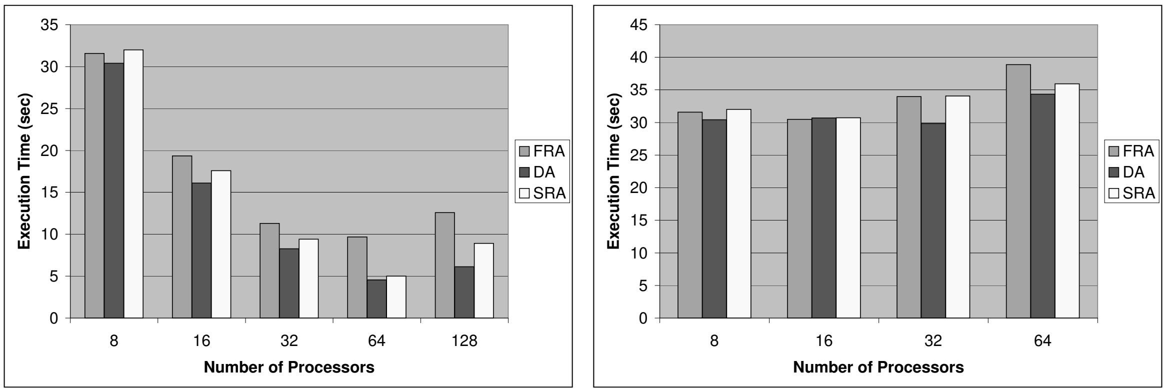
[1]We could not get all of the results for scaled input experiments on 128 processors before submitting the abstract. We will include those results in the full paper.



(a) SAT application.



(b) WCS application.



(c) VM application.

Figure 3: Query execution time for various applications with fixed input size (left), and scaled input size (right).

increases, communication volume for DA decreases since there are fewer input chunks per processor, while communication volume remains almost constant for FRA. On the other hand, as seen in Figure 4(b), the volume of communication for DA increases for scaled input size.

The volume of communication for SRA implicitly depends on the average fan-in of each output chunk. If fan-in is much larger than the number of processors, it is likely that each processor will have input chunks that map to all output chunks. Thus, in such cases, SRA performance is identical to FRA. When the number of processors is greater than the fan-in, there will be fewer ghost chunks than there are output chunks for SRA, thus resulting in less overhead in the *initialization* and *global combine* phases than for FRA. This effect is observed for VM for 32 or more processors, and for WCS for 64 or more processors. As is seen in Figure 4(c)-(d), the computation time does not scale perfectly. For DA this is because of load imbalance incurred during the local reduction phase, while for FRA and SRA it is due to constant overheads in the initialization and global reduction phases.

The results show that all three strategies can be usefully employed for various applications under different machine configurations. We are working on cost models to automatically select the best strategy during query planning.

References

[1] A. Afework, M. D. Beynon, F. Bustamante, A. Demarzo, R. Ferreira, R. Miller, M. Silberman, J. Saltz, A. Sussman, and H. Tsang. Digital dynamic telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, Nov. 1998.

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

[3] C. Chang, R. Ferreira, A. Sussman, and J. Saltz. Infrastructure for building parallel database systems for multi-dimensional data. In *Proceedings of the Second Merged IPPS/SPDP (13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing)*. IEEE Computer Society Press, Apr. 1999.

[4] C. Chang, T. Kurc, A. Sussman, and J. Saltz. Query planning for range queries with user-defined aggregation on multi-dimensional scientific datasets. Technical Report CS-TR-3996 and UMIACS-TR-99-15, University of Maryland, Department of Computer Science and UMIACS, Feb. 1999.

[5] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. Saltz. Titan: A high performance remote-sensing database. In *Proceedings of the 1997 International Conference on Data Engineering*, pages 375–384. IEEE Computer Society Press, Apr. 1997.

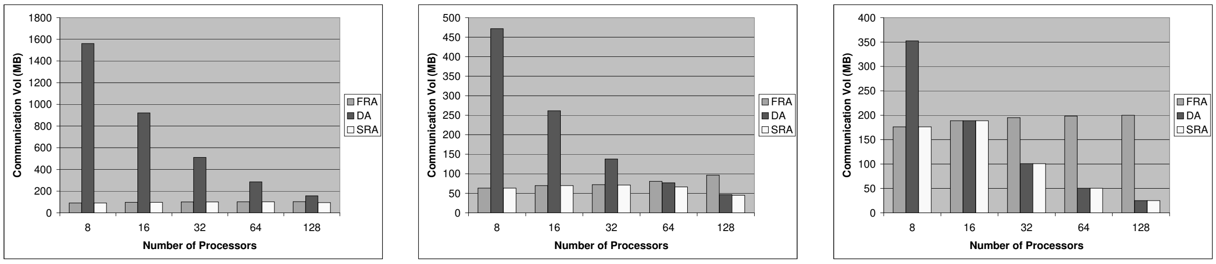
[6] P. F. Corbett and D. G. Feitelson. The Vesta parallel file system. *ACM Trans. Comput. Syst.*, 14(3):225–264, Aug. 1996.

[7] C. Faloutsos and P. Bhagwat. Declustering using fractals. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, pages 18–25, Jan. 1993.

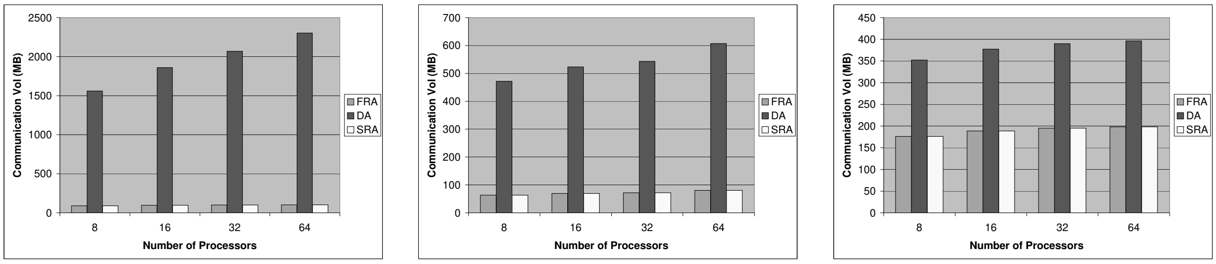
[8] J. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, Spain, July 1995.

[9] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. ACM Press, Nov. 1994.

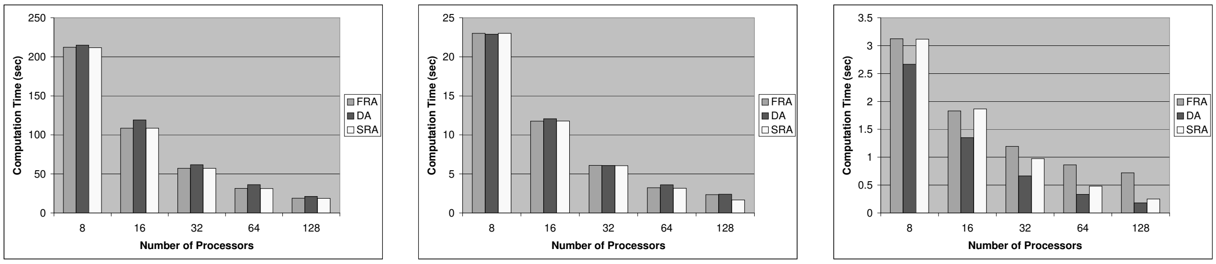
[10] T. M. Kurc, A. Sussman, and J. Saltz. Coupling multiple simulations via a high performance customizable database system. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Mar. 1999.



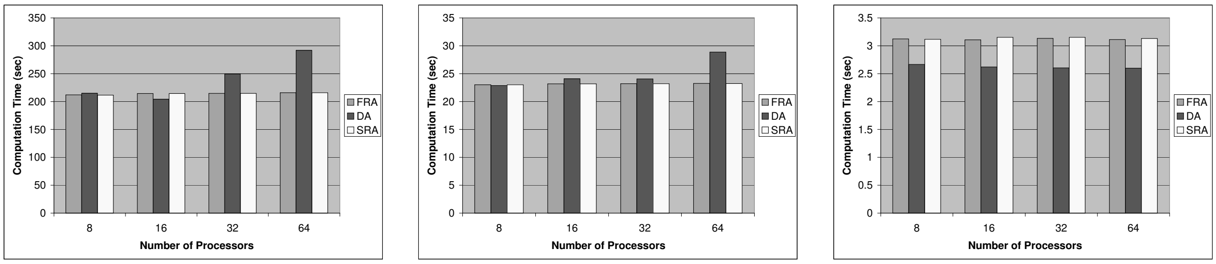
(a) Volume of communication, fixed input size. SAT (left), WCS (middle), VM (right).



(b) Volume of communication, scaled input size. SAT (left), WCS (middle), VM (right).



(c) Computation time, fixed input size. SAT (left), WCS (middle), VM (right).



(d) Computation time, scaled input size. SAT (left), WCS (middle), VM (right).

Figure 4: Communication volume and computation time for various applications, for fixed and scaled input data size.

[11] B. Moon and J. H. Saltz. Scalability analysis of declustering methods for multidimensional range queries. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):310–327, March/April 1998.

[12] NASA Goddard Distributed Active Archive Center (DAAC). Advanced Very High Resolution Radiometer Global Area Coverage (AVHRR GAC) data. Available at http://daac.gsfc.nasa.gov/CAMPAIGN_DOCS/LAND_BIO/origins.html.

[13] N. Nieuwejaar and D. Kotz. The Galley parallel file system. In *Proceedings of the 1996 International Conference on Supercomputing*, pages 374–381. ACM Press, May 1996.

[14] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings Supercomputing '95*. IEEE Computer Society Press, Dec. 1995.

[15] C. T. Shock, C. Chang, B. Moon, A. Acharya, L. Davis, J. Saltz, and A. Sussman. The design and evaluation of a high-performance earth science database. *Parallel Computing*, 24(1):65–90, Jan. 1998.

[16] 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.

[17] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. Passion: Optimized I/O for parallel applications. *IEEE Computer*, 29(6):70–78, June 1996.

[18] The USGS General Cartographic Transformation Package, version 2.0.2. *ftp://mapping.usgs.gov/pub/software/current_software/gctp/*, 1997.

[19] M. Uysal, T. M. Kurc, A. Sussman, and J. Saltz. A performance prediction framework for data intensive applications on large scale parallel machines. In *Proceedings of the Fourth Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*, number 1511 in Lecture Notes in Computer Science, pages 243–258. Springer-Verlag, May 1998.