

Querying Very Large Multi-dimensional Datasets in ADR ^{*}

Tahsin Kurc[†], Chialin Chang[†], Renato Ferreira[†], Alan Sussman[†], Joel Saltz^{†+}

[†] 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

Abstract

Applications that make use of very large scientific datasets have become an increasingly important subset of scientific applications. In these applications, datasets are often multi-dimensional, i.e., data items are associated with points in a multi-dimensional attribute space, and access to data items is described by range queries. The basic processing involves mapping input data items to output data items, and some form of aggregation of all the input data items that project to the each output data item. We have developed an infrastructure, called the *Active Data Repository* (ADR), that integrates storage, retrieval and processing of multi-dimensional datasets on distributed-memory parallel architectures with multiple disks attached to each node. In this paper we address efficient execution of range queries on distributed memory parallel machines within ADR framework. We present three potential strategies, and evaluate them under different application scenarios and machine configurations. We present experimental results on the scalability and performance of the strategies on a 128-node IBM SP.

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 [23], output from hydrodynamics and chemical transport simulations [19], and archives of medical images [2]. For example, a dataset of coarse-grained satellite data (with 4.4 km pixels), covering the whole earth surface and captured over a relatively short period of time (10 days) is about 4.1GB; a finer-grained version (1.1 km per pixel) contains about 65 GB of sensor data. In medical imaging, the size of a single digitized composite slide image at high power from a light microscope is over 7GB (uncompressed), and a single large hospital can process thousands of slides per day.

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. That is, the aggregation operation is an *associative* and *commutative* operation—the aggregation functions allowed correspond to the *distributive* and *algebraic* aggregation functions defined by Gray et. al [15]. The mapping function, $Map(i_e)$, maps an input item to a set of

^{*}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.

```

O ← Output Dataset, I ← 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.

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. The intermediate results stored in the accumulator are post-processed to produce final results (steps 9–11).

Some typical examples of applications that make use of multi-dimensional scientific datasets are satellite data processing applications [1, 8, 30], the Virtual Microscope and analysis of microscopy data [2, 14], and simulation systems for water contamination studies [19]. In satellite data processing, for example, 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 [1, 30], 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 [36]; each pixel in the composite image is computed by selecting the “best” sensor value that maps to the associated grid point. Another example is the *Virtual Microscope* [2, 14], which supports the ability to interactively view and process digitized data arising from tissue specimens. The raw data for such a system can be captured by digitally scanning collections of full microscope slides under high power. The digitized images from a slide are effectively a three-dimensional dataset, since each slide can contain multiple two-dimensional focal planes. At the basic level, a range query selects a region on a focal plane in a slide. The processing for the Virtual Microscope requires projecting high resolution data onto a grid of suitable resolution (governed by the desired magnification) and appropriately compositing pixels mapping onto a single grid point, to avoid introducing spurious artifacts into the displayed image.

We have developed an infrastructure, called the Active Data Repository (ADR) [7], 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. We have demonstrated [7, 19] that ADR achieves good performance for a variety of applications, including satellite data processing, archiving and processing of medical imagery, and coupling multiple simulations [2, 8, 19].

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* [37] to generate various application scenarios for the applications classes that motivated the design of ADR. 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. We present experimental evaluation of the three strategies on a 128-node IBM SP.

2 Query Execution in ADR

2.1 The Active Data Repository

The system architecture of *Active Data Repository* (ADR) [6, 7] consists of a front-end and a parallel back-end. The front-end interacts with client applications and relays the range queries to the back-end, which consists of a set of processing nodes and multiple disks attached to these nodes. During query processing, the back-end nodes are responsible for retrieving the data items that intersect the range query and performing user-defined *Initialize*, *Map*, *Aggregate*, and *Output* functions over the data items retrieved to generate the output products. ADR has been developed as a set of modular services in C++. The ADR services include:

- *Attribute space service* manages the registration and use of multi-dimensional attribute spaces and user-defined mapping functions (*Map*). An attribute space is specified by the number of dimensions and the range of values in each dimension.
- *Dataset service* manages the datasets stored on the ADR back-end and provides utility functions for loading datasets into the ADR back-end. ADR expects each of its datasets to be partitioned into *data chunks*, each chunk consisting of one or more data items from the same dataset.
- *Indexing service* manages various indices (default and user-provided) for the datasets stored in the ADR back-end. An index returns the disk locations of the set of data chunks that contain data items that fall inside the given multi-dimensional range query.
- *Data aggregation service* manages the user-provided functions *Initialize* and *Aggregate* to be used in aggregation operations, and *Output* functions to generate the final outputs. It also encapsulates the data types of both the intermediate results (accumulator) used by these functions and the final outputs generated by these functions.
- *Query planning service* determines a query plan to efficiently process a set of queries based on the amount of available resources in the back-end. A query plan specifies how the final output of each query is computed and the order in which data chunks are retrieved for processing.
- *Query execution service* manages all the resources in the system and carries out the query plan generated by the query planning service. It is also responsible for returning the final output of a given query to its specified destination. If a new output dataset is created or an existing dataset is updated, the results can be written back to disks. The output can also be returned to the client from the back-end nodes, either through a socket interface or via Meta-Chaos [11]. The socket interface is used for sequential clients, while the Meta-Chaos interface is mainly used for parallel clients.

Through use of these services, ADR allows customization for application specific processing (i.e. *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. Figure 2 shows the architecture of an application implemented as a customized ADR instance.

We now describe how datasets are stored in ADR, and the ADR query planning and query execution services in more detail.

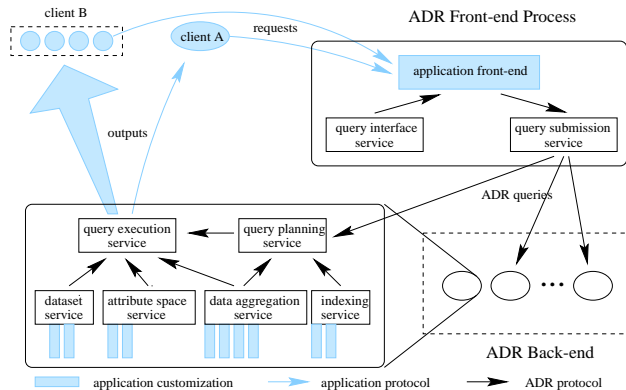


Figure 2: A complete application suite implemented as a customized ADR application. The shaded bars represent functions added to ADR by the user as part of the customization process. Client A is a sequential program while client B is a parallel program.

2.2 Storing Datasets in ADR

Loading a dataset into ADR is accomplished in four steps: (1) partition a dataset into data chunks, (2) compute placement information, (3) move data chunks to the disks according to placement information, and (4) create an index. 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. The placement information describes how data chunks are declustered and clustered across the disk farm. Chunks are distributed across the disks attached to ADR back-end nodes using a declustering algorithm [12, 21] 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.

2.3 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 corresponding input tile 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 output tiles. Hence, an input chunk may be retrieved multiple times during execution of the processing loop. Figure 3 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 data items in input and accumulator chunks) is partitioned across processors. This is accomplished by assigning each processor the responsibility for processing a subset of the input and/or accumulator chunks.

```

(* Output and Input Dataset Tiles *)
 $O_{tiles} \leftarrow \{O_t\}$  and  $I_{tiles} \leftarrow \{I_t\}$  for  $1 \leq t \leq N$ 
1. foreach [ $O_t, I_t$ ] in [ $O_{tiles}, I_{tiles}$ ] do
    (* Initialization *)
2. foreach  $o_c$  in  $O_t$  do
3.   read  $o_c$ 
4.    $a_c \leftarrow Initialize(o_c)$ 
    (* Reduction *)
5. foreach  $i_c$  in  $I_t$  do
6.   read  $i_c$ 
7.    $S_A \leftarrow Map(i_c) \cap O_t$ 
8.   foreach  $a_c$  in  $S_A$  do
9.      $a_c \leftarrow Aggregate(i_c, a_c)$ 
    (* Output *)
10. foreach  $a_c$  do
11.    $o_c \leftarrow Output(a_c)$ 
12. write  $o_c$ 

```

Figure 3: Basic processing loop with tiled input and output datasets. N is the total number of tiles after tiling step. i_c , o_c , and a_c denote input, output, and accumulator chunks, respectively.

2.4 Query Execution

Execution of a query in ADR is done through the query execution service, which manages all the resources in the system and carries out the query plan generated by the query planning service. The primary feature of the query execution service is its ability to integrate data retrieval and processing for a wide variety of applications. This is achieved by pushing processing operations into the storage manager and allowing processing operations to access the buffer used to hold data arriving from disk. As a result, the system avoids one or more levels of copying that would be needed in a layered architecture where the storage manager and the processing belonged to different layers. 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 3 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

1. $Memory = \text{Minimum of the size of memory over all processors (for holding accumulator chunks)}$
2. $Tile = 1; MemoryUsed = 0$
3. **while** (there is an unassigned output chunk)
4. Select an output chunk C
5. $ChunkSize = \text{Size of the accumulator chunk corresponding to } C$
6. **if** $((ChunkSize + MemoryUsed) > Memory)$
7. $Tile = Tile + 1$
8. $MemoryUsed = ChunkSize$
9. **else**
10. $MemoryUsed = MemoryUsed + ChunkSize$
11. Assign chunk C to tile $Tile$
12. Let k be the processor that owns the chunk C
13. Add chunk C to the set of local accumulator chunks of k for tile $Tile$
14. Add chunk C to the set of ghost chunks on all other processors for tile $Tile$
15. Add the set of local input chunks of k that map to chunk C to the set of input chunks to be retrieved by k during query execution for tile $Tile$
16. **end while**

Figure 4: Tiling and workload partitioning for the fully replicated accumulator strategy.

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 describe three query processing strategies that use different workload partitioning and tiling schemes. To simplify the presentation, we assume that the target range query involves only one input and one output dataset. Both the input and output datasets are assumed to be already partitioned into chunks and declustered across the disks in the system. In the following discussions we assume that an accumulator chunk is allocated in memory for each output chunk to hold the partial results, and that the total size of the accumulator exceeds the aggregate memory capacity of the parallel machine, so that tiling is needed.

We define a *local input/output chunk* on a processor as an input/output chunk stored on one of the disks attached to 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* (or *ghost cell*) is 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 query strategies described in this section, we use a *Hilbert space-filling curve* [12, 13, 21] to order the output chunks. Our goal is to minimize the total length of the boundaries of the tiles, by assigning spatially close chunks in the multi-dimensional attribute space to the same tile, to reduce the number of input chunks crossing one or more boundaries. The advantage of using Hilbert curves is that they have good clustering properties [21], 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 (e.g., step 4 in Figure 4). The current implementation, however, does not take into account the distribution of input chunks in the output attribute space, so for some distributions of the input data in its attribute space 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 is assigned the responsibility to carry out processing associated with its local input chunks. Figure 4 shows the tiling and workload partitioning step for this strategy. The accumulator is partitioned into tiles, each of which fits into the 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 each processor generates partial results using its local input

1. **for** (each processor p)
 - Memory(p) = Size of memory on p (for holding accumulator chunks)
2. Tile = 1, MemoryFull = 0
3. **while** (there is an unassigned output chunk)
4. Select an output chunk C
5. Let S_o be the set of processors having at least one input chunk that projects to chunk C
6. ChunkSize = Size of the accumulator chunk corresponding to C
7. **for** (p in S_o)
8. **if** ((Memory(p) - ChunkSize) < 0) MemoryFull = 1
9. **if** (MemoryFull == 1)
10. Tile = Tile + 1
11. **for** (p in S_o) Memory(p) = (size of memory on p) - ChunkSize
12. **for** (p not in S_o) Memory(p) = size of memory on p
13. MemoryFull = 0
14. **else**
15. **for** (p in S_o) Memory(p) = Memory(p) - ChunkSize
16. Assign chunk C to tile Tile
17. Let k be the processor that owns output chunk C
18. Add chunk C to the set of local accumulator chunks of k for tile Tile
19. Add chunk C to the set of ghost chunks on each processor in S_o for tile Tile
20. Add the set of local input chunks of k that map to output chunk C to the set of input chunks to be retrieved by k during query execution
21. **end while**

Figure 5: The tiling and workload partitioning step for the sparsely replicated accumulator strategy.

chunks. These partial results are combined into the final result in the *global combine* phase of query execution. Ghost chunks are forwarded to the processors that own the corresponding output (accumulator) chunks to produce the final output product.

Executing step 15 of the while loop requires either an efficient inverse mapping function or an efficient search method, either of which must return the input chunks that map to a given output chunk. In some cases it may be less expensive to find the projected output chunks for each input chunk. For example, the input chunks may be irregularly and sparsely distributed in the input attribute space, while the output may be a dense and regular array such as a raster image, with the output chunks as regular subregions of the array. In such cases, step 15 can be carried out in a separate loop, iterating over input chunks, finding the output chunks they map to, and adding the input chunks to the appropriate tiles. To run step 15 as a separate loop, the implementation must store the assigned tile number with each output chunk.

3.2 Sparsely Replicated Accumulator (SRA) Strategy

The fully replicated accumulator strategy eliminates interprocessor communication for input chunks, by replicating all accumulator chunks. However, this is wasteful of memory, because the 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. The tiling and workload partitioning step of the sparsely replicated accumulator strategy is shown in Figure 5.

In this strategy, a ghost chunk is allocated only on processors owning at least one input chunk that projects to the corresponding accumulator chunk. Replicating accumulator chunks sparsely in this way requires that we find the corresponding set of processors for each output chunk during the tiling step (step 5 in Figure 5). As was stated in the previous section, sometimes it may be easier to find the projected output chunks for each input chunk. For those cases, an alternative solution is to maintain a list for each output chunk to store the set of processors that require allocating

1. **for** (each processor p)
2. $\text{Memory}(p) = \text{Size of memory on } p$ (for holding accumulator chunks)
3. $\text{Tile}(p) = 1$
4. **while** (there is unassigned output chunk)
5. Select an output chunk C
6. Let p be the processor that owns chunk C
7. $\text{ChunkSize} = \text{Size of the accumulator chunk corresponding to } C$
8. **if** $((\text{Memory}(p) - \text{ChunkSize}) < 0)$
9. $\text{Tile}(p) = \text{Tile}(p) + 1$
10. $\text{Memory}(p) = (\text{size of memory on } p) - \text{ChunkSize}$
11. **else**
12. $\text{Memory}(p) = \text{Memory}(p) - \text{ChunkSize}$
13. Assign chunk C to tile $\text{Tile}(p)$
14. Add chunk C to the set of local accumulator chunks of p for tile $\text{Tile}(p)$
15. Add all the local and remote input chunks that map to chunk C
to the set of input chunks to be retrieved and processed by p for $\text{Tile}(p)$
16. **end while**
17. $\text{Tile} = \text{maximum over } p \text{ of } \text{Tile}(p)$

Figure 6: The tiling and workload partitioning step for the distributed accumulator strategy.

an accumulator chunk. The list is created prior to the tiling step by iterating over the input chunks, projecting them to output chunks, and storing the result (processor id) with each output chunk.

3.3 Distributed Accumulator (DA) Strategy

In this scheme the output (accumulator) chunks in each tile are partitioned into disjoint sets, referred to as *working sets*. Each processor is given the responsibility to carry out the operations associated with the output chunks in a working set. The tiling and workload partitioning step is shown in Figure 6. Output chunks are selected (step 5) in Hilbert curve order, as for the other two schemes. In the algorithm shown in Figure 6, the working set of a processor for a tile is composed of only the local output chunks in that processor. Local chunks for each processor are assigned to the same tile until the memory space allocated for the accumulator on that processor is filled.

In the DA strategy accumulator chunks are not replicated on other processors, thus no ghost chunks are allocated. This allows the DA strategy to make more effective use of the overall system memory by assigning more chunks to a tile, as a result produce fewer tiles than the other two schemes. Therefore, fewer input chunks are likely to be retrieved for multiple tiles in this scheme than the other two schemes. 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 mapping function may map an input chunk to multiple output chunks, an input chunk may be forwarded to multiple processors.

Figure 7 illustrates FRA and DA strategies for an example application. A possible distribution of input and output chunks to the processors is illustrated at the top. Input chunks are denoted by triangles while output chunks are denoted by rectangles. The final result to be computed by reduction (aggregation) operations is also shown.

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

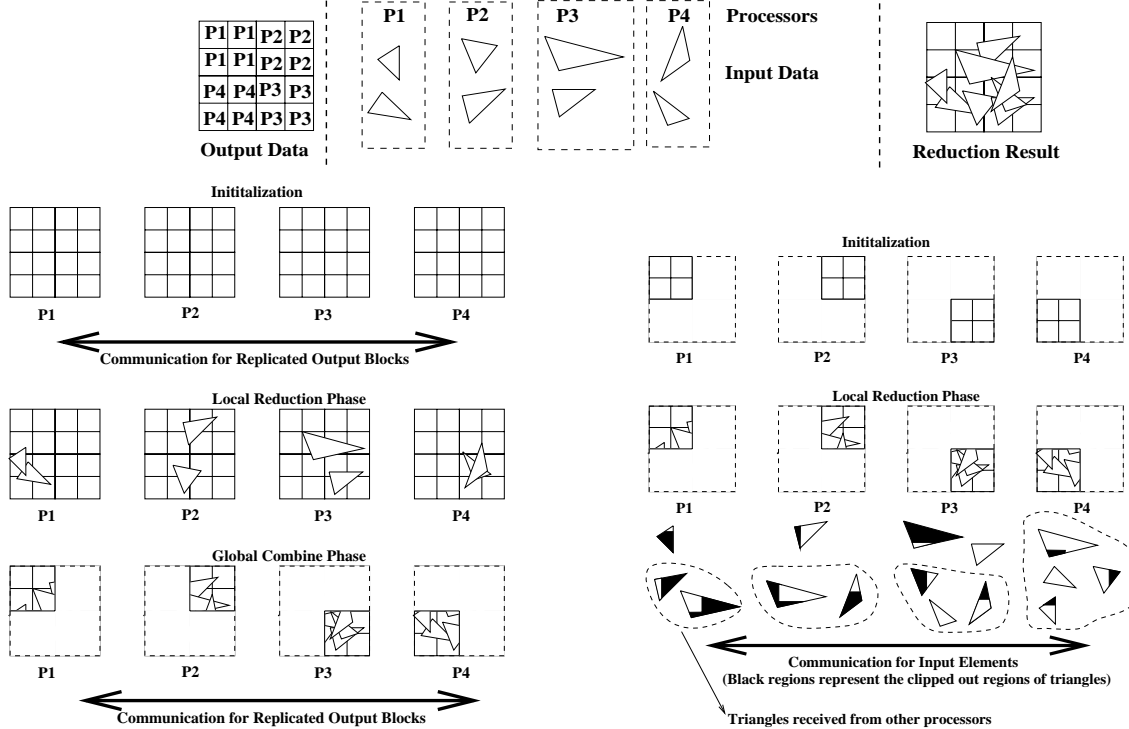


Figure 7: FRA strategy (left) and DA strategy (right).

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* [37] 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 [12].

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 largest input datasets. The last column shows the computation time per chunk for the different phases of query execution (see Section 2.4); 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 [23]; 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 8 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 at the minimum values shown in Table 1,

App.	Input Dataset		Output Dataset		Average Fan-in	Average Fan-out	Computation (in milliseconds) I-LR-GC-OH
	Num. of Chunks	Total Size	Num. of Chunks	Total Size			
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.

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 8 display query execution time when the input dataset is scaled with the number of processors. 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.

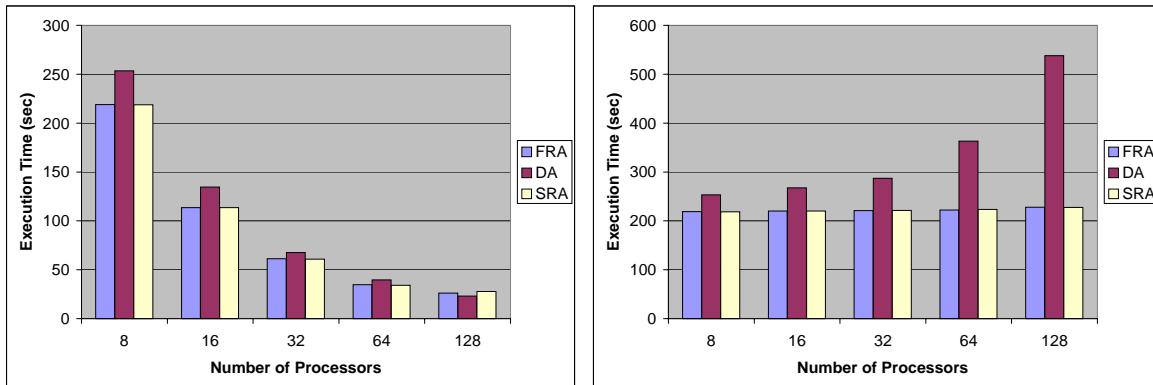
Figures 9(a)-(d) display the change in volume of communication and computation time per processor for fixed and scaled input datasets on varying number of processors. Note that all strategies read the same amount of data from disks. However, the volume of communication for DA is proportional to the number of input chunks on each processor and the average fan-out of each input chunk, while for FRA it is proportional to the total number of output chunks. As is seen in Figure 9(a), as the number of processors 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 9(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 9(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.

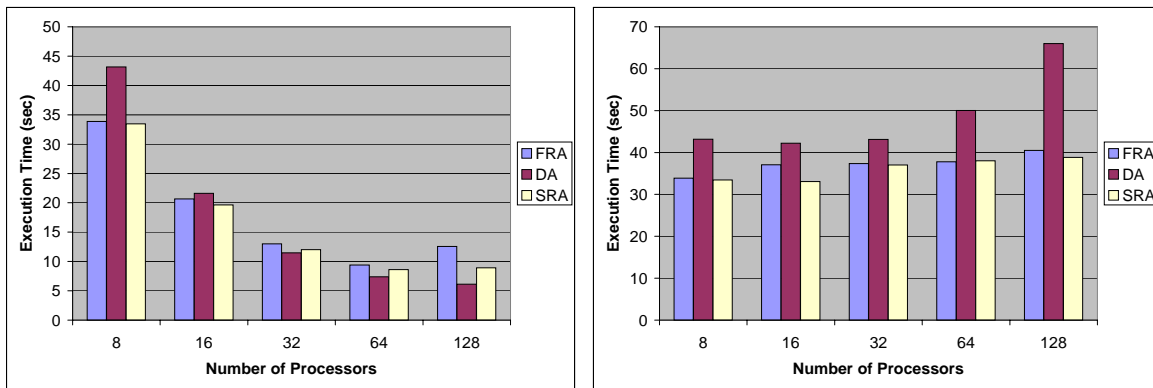
5 Related Work

Several runtime support libraries and file systems have been developed to support efficient I/O in a parallel environment [4, 9, 16, 18, 24, 28, 34, 35]. These systems mainly focus on supporting regular strided access to uniformly distributed datasets, such as images, maps, and dense multi-dimensional arrays. They also usually provide a collective I/O interface, in which all processing nodes cooperate to make a single large I/O request. 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.

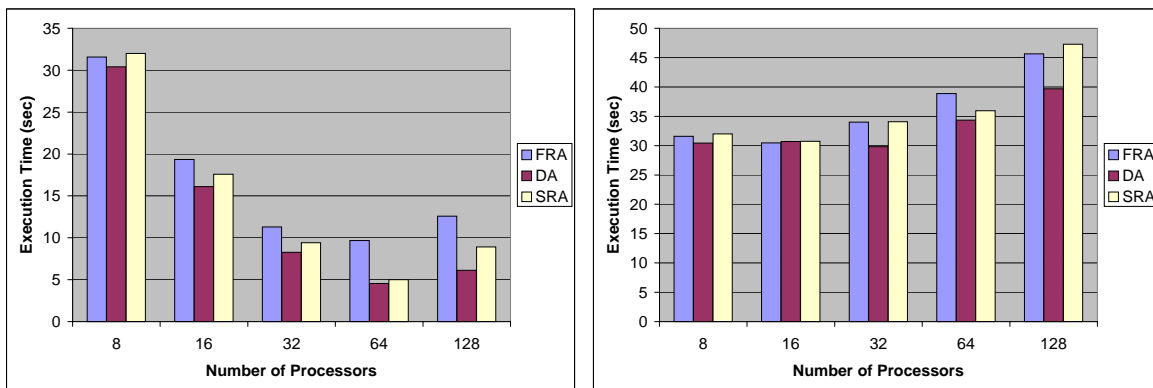
The increasing importance of multi-dimensional datasets has been recognized by several database research groups and multiple systems have been developed for managing and/or visualizing them [3, 17, 20, 26, 32]. In addition, com-



(a) SAT application.

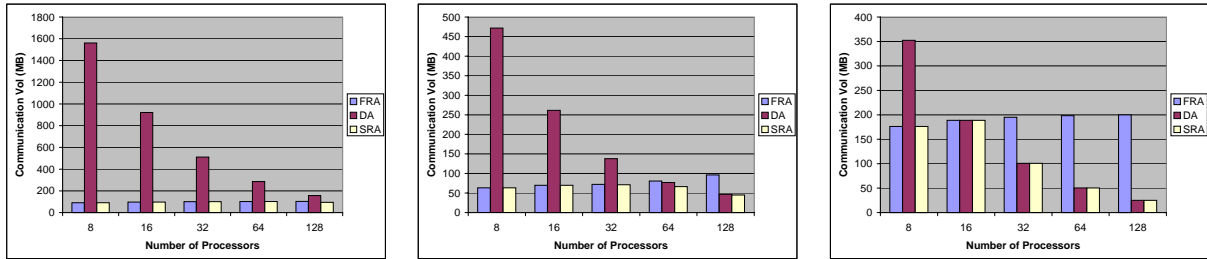


(b) WCS application.

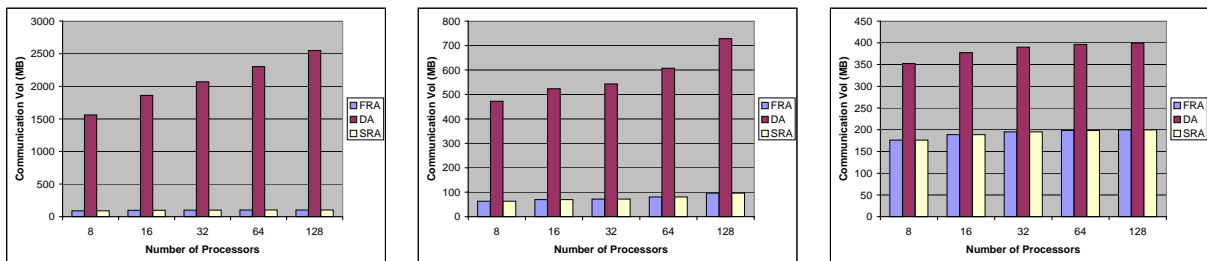


(c) VM application.

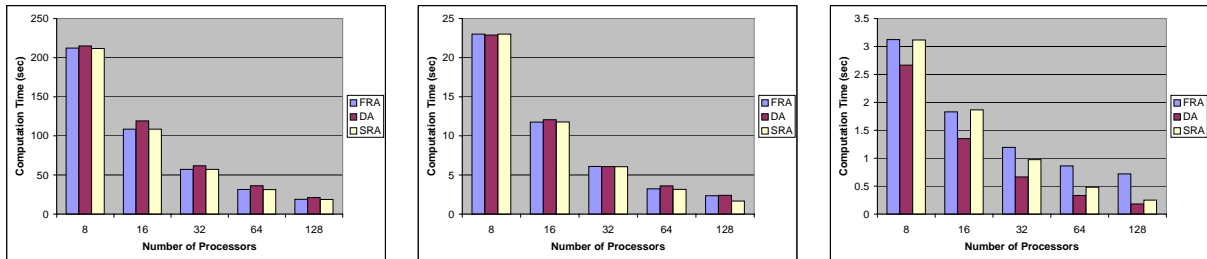
Figure 8: Query execution time for various applications with fixed input size (left), and scaled input size (right).



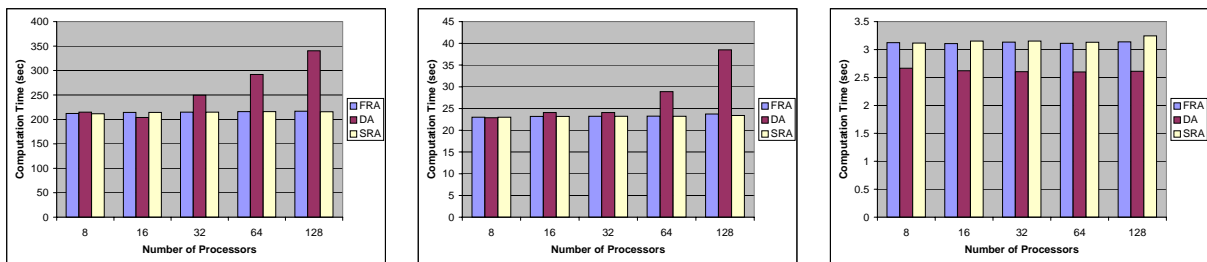
(a) Volume of communication, fixed input size. SAT (left), WCS (middle), VM (right).



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

mercial object-relational database systems provide some support for managing multi-dimensional datasets (e.g., the *SpatialWare DataBlade Module* [31] for the Informix Universal Server and the Oracle 8 *Spatial data cartridge* [25]). These systems, however, focus on lineage management, retrieval and visualization of multi-dimensional datasets. They provide little or no support for analyzing or processing these datasets – the assumption is that this is too application-specific to warrant common support. As a result, applications that process these datasets are usually decoupled from data storage and management, resulting in inefficiency due to copying and loss of locality. Furthermore, every application developer has to implement complex support for managing and scheduling the processing. The main performance advantage of ADR comes from integrating complex processing with the retrieval of stored data.

Parallel database systems have been a major topic in the database community [10] for a long time, and much attention has been devoted to the implementation and scheduling of parallel joins [22, 27]. As in many parallel join algorithms, our query strategies exploit parallelism by effectively partitioning the data and workload among the processors. However, the characteristics of the distributive and algebraic aggregation functions allowed in our queries enable deployment of more flexible workload partitioning schemes through the use of ghost chunks. Several extensible database systems have been proposed to provide support for user-defined functions [5, 33]. The incorporation of user-defined functions into a computation model as general as the relational model, can make query optimization very difficult, and has recently attracted much attention [29]. ADR, on the other hand, implements a more restrictive processing structure that mirrors the processing of our target applications. Good performance is achieved through effective workload partitioning and careful scheduling of the operations to obtain good utilization of the system resources, not by rearranging the algebraic operators in a relational query tree, as is done in relational database systems.

6 Conclusions and Future Work

In this paper we addressed efficient execution of range queries (i.e. the basic processing loop in Figure 1) on distributed memory parallel machines within the ADR framework. We have presented three query processing strategies that employ different tiling and workload partitioning schemes. We have evaluated scalability of these strategies for different application scenarios, varying both the number of processors and input dataset size.

Our results indicate that no one scheme is always best. The relative performance of the various query planning strategies changes with the application characteristics and machine configuration. The strategies presented in this paper represent two extreme approaches. The full and sparse replicated accumulator strategies lie one end of the spectrum of possible strategies; processing is performed on the processors where input chunks are stored. The distributed accumulator strategy, on the other hand, lies at the other end; processing is carried out on the processors where output chunks are stored. Our experimental results suggest that a hybrid strategy may provide better performance. For example, the tiling and workload partitioning steps can be formulated as a multi-graph partitioning problem, with input and output chunks representing the graph vertices, and the mapping between input and output chunks provided by the mapping function representing the graph edges. A planning algorithm based on graph partitioning could provide an effective means for implementing a general hybrid strategy.

One of the long-term goals of our work on query planning strategies is to develop simple but reasonably accurate cost models to guide and automate the selection of an appropriate strategy. Answering the questions “under what circumstances do the simple cost models provide accurate or inaccurate results?”, and “how can we refine the cost model in situations where it does not provide reasonably accurate results?” require extensive evaluation of these strategies. We plan to evaluate our strategies on additional applications and on other parallel architectures.

References

- [1] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz, and A. Sussman. Tuning the performance of I/O-intensive parallel applications. In *Proceedings of the Fourth ACM Workshop on I/O in Parallel and Distributed Systems*, May 1996.
- [2] 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.
- [3] P. Baumann, P. Furtado, and R. Ritsch. Geo/environmental and medical data management in the RasDaMan system. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB97)*, pages 548–552, Aug. 1997.

- [4] 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.
- [5] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. R. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The EXODUS extensible DBMS project: An overview. In D. Zdonik, editor, *Readings on Object-Oriented Database Systems*, pages 474–499. Morgan Kaufman, San Mateo, CA, 1990.
- [6] C. Chang, A. Acharya, A. Sussman, and J. Saltz. T2: A customizable parallel database for multi-dimensional data. *ACM SIGMOD Record*, 27(1):58–66, Mar. 1998.
- [7] 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.
- [8] 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.
- [9] P. F. Corbett and D. G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, Aug. 1996.
- [10] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.
- [11] G. Edjlali, A. Sussman, and J. Saltz. Interoperability of data parallel runtime libraries. In *Proceedings of the Eleventh International Parallel Processing Symposium*. IEEE Computer Society Press, Apr. 1997.
- [12] 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.
- [13] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Mar. 1989.
- [14] R. Ferreira, B. Moon, J. Humphries, A. Sussman, J. Saltz, R. Miller, and A. Demarzo. The Virtual Microscope. In *Proceedings of the 1997 AMIA Annual Fall Symposium*, pages 449–453. American Medical Informatics Association, Hanley and Belfus, Inc., Oct. 1997.
- [15] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proceedings of the 1996 International Conference on Data Engineering*, pages 152–159. IEEE Computer Society Press, 1996.
- [16] 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.
- [17] Y. Ioannidis, M. Livny, S. Gupta, and N. Ponnkanti. ZOO: A desktop experiment management environment. In *Proc. 22nd International VLDB Conference*, pages 274–85, 1996.
- [18] 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.
- [19] 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.
- [20] M. Livny, R. Ramakrishnan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and K. Wenger. DEVise: integrated querying and visual exploration of large datasets. In *Proceedings of the 1997 ACM-SIGMOD Conference*, pages 301–12, 1997.
- [21] 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.
- [22] M. C. Murphay and D. Rotem. Multiprocessor join scheduling. *IEEE Transactions on Knowledge and Data Engineering*, 5(2):322–338, Apr. 1993.
- [23] 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.
- [24] 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.
- [25] The Oracle 8 spatial data cartridge, 1997. <http://www.oracle.com/st/cartridges/spatial/>.

- [26] J. Patel et al. Building a scaleable geo-spatial DBMS: technology, implementation, and evaluation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD97)*, pages 336–47, 1997.
- [27] D. Schneider and D. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of the 16th VLDB Conference*, pages 469–480, Melbourne, Australia, Aug. 1990.
- [28] 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.
- [29] P. Seshadri, M. Livny, and R. Ramakrishnan. The case for enhanced abstract data types. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB97)*, pages 66–75. Morgan Kaufmann, 1997.
- [30] 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.
- [31] SpatialWare DataBlade Module (from MapInfo Corp.), 1997. <http://www.informix.com/informix/bussol/iusdb/databld/dbtech/sheets/spware.htm>.
- [32] M. Stonebraker. An overview of the Sequoia 2000 project. In *Proceedings of the 1992 COMPCON Conference*, pages 383–388, San Francisco, CA, 1992.
- [33] M. Stonebraker, L. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, Mar. 1990.
- [34] 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.
- [35] 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.
- [36] The USGS General Cartographic Transformation Package, version 2.0.2. ftp://mapping.usgs.gov/pub/software/current_software/gctp/, 1997.
- [37] 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.