Optimizing Retrieval and Processing of Multi-dimensional Scientific Datasets *

Chialin Chang†, Tahsin Kurc†, 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

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

Abstract

Exploring and analyzing large volumes of data plays an increasingly important role in many domains of scientific research. We have been developing the Active Data Repository (ADR), an infrastructure that integrates storage, retrieval, and processing of large multi-dimensional scientific datasets on distributed memory parallel machines with multiple disks attached to each node. In earlier work, we proposed three strategies for processing range queries within the ADR framework. Our experimental results show that the relative performance of the strategies changes under varying application characteristics and machine configurations. In this work we investigate approaches to guide and automate the selection of the best strategy for a given application and machine configuration. We describe analytical models to predict the relative performance of the strategies when input data elements are uniformly distributed in the attribute space of the output dataset, restricting the output dataset to be a regular d-dimensional array. We present an experimental evaluation of these models for various synthetic datasets and for several driving applications on a 128-node IBM SP.

1 Introduction

The exploration and analysis of large datasets is playing an increasingly central role in many areas of scientific research. Depending on the application area, datasets may include data produced by scientific simulations, along with measurements obtained from satellites, microscopes, seismic data or tomographic imaging techniques.

*This research was supported by the National Science Foundation under Grant #ACI-9619020 (UC Subcontract # 10152408) and the Office of Naval Research under Grant #N6600197C8534.

$O \leftarrow$ 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.

Over the past several years we have been actively working on data intensive applications that employ large-scale scientific datasets, including applications that explore, compare, and visualize results generated by large scale simulations [15], visualize and generate data products from global coverage satellite data [7], and visualize and analyze digitized microscopy images [1]. Such applications often use only a subset of all the data available in both the input and output datasets. References to data items are described by a *range query*, namely a multi-dimensional bounding box in the underlying multi-dimensional attribute space of the dataset(s). Only the data items whose associated coordinates fall within the multi-dimensional box are retrieved and processed. 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 data values 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. We extend the computational model to allow for an intermediate data structure, referred to as an *accumulator*, that can be 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).

We have been developing the Active Data Repository (ADR) [5], a software system that efficiently supports the processing loop shown in Figure 1, integrating storage, retrieval, and processing of large multi-dimensional scientific datasets on distributed memory parallel machines with multiple disks attached to each node. 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 described above), 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.

This paper addresses optimization of processing for range queries on distributed memory machines within the ADR framework. In earlier work [6, 14], we described three potential processing strategies, and evaluated the relative performance of these strategies for several application scenarios and machine configurations. Our experimental results showed that the relative performance of the strategies changes under varying application characteristics and machine configurations. In this paper we investigate approaches to guide and automate the selection of the best strategy for a given application and machine configuration. We describe analytical models to predict relative performance of the strategies when input data elements are uniformly distributed in the attribute space of the output dataset, restricting the output dataset to be a regular d-dimensional array. We present an experimental evaluation of these models for synthetic datasets and for several driving applications [1, 7, 15].

2 Query Execution Strategies

In this section we briefly describe three strategies for processing range queries in ADR. First we briefly describe how datasets are stored in ADR, and outline the main phases of query execution in ADR. More detailed descriptions of these strategies and of ADR in general can be found in [5, 6, 14].

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, communicated and computed on as a whole during query processing. Every data item is associated with a point in a multi-dimensional attribute space, so every chunk is associated with a minimum bounding rectangle (MBR) that encompasses the coordinates (in the associated attribute space) of all the data 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 placed in the same chunk. Chunks are distributed across the disks attached to ADR back-end nodes using a declustering algorithm [10, 16] 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 [11]) 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 Processing in ADR

Processing of a query in ADR is accomplished in two steps; query planning and query execution.

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 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 contains the input chunks that map to the output chunks in the tile. During query processing, each output tile is cached in main memory, and input chunks from the required 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 tiles. Hence, an input chunk may be retrieved multiple times during execution of the processing loop. In the workload partitioning step, the workload associated with each tile (i.e. aggregation of input items into 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.

The execution 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.

A query iterates through these phases repeatedly until all tiles have been processed and the entire output dataset has been computed. 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 queues are polled and, upon their completion, new asynchronous operations are initiated when there is more work to be done and memory buffer space is available. Data chunks are therefore retrieved and processed in a pipelined fashion.

2.3 Query Processing Strategies

In the following discussion, 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* [10] 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 [16], 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.

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.

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 maps to the corresponding accumulator chunk.

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, for each processor, local output chunks from that processor until the memory space allocated for the corresponding accumulator chunks in the processor 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

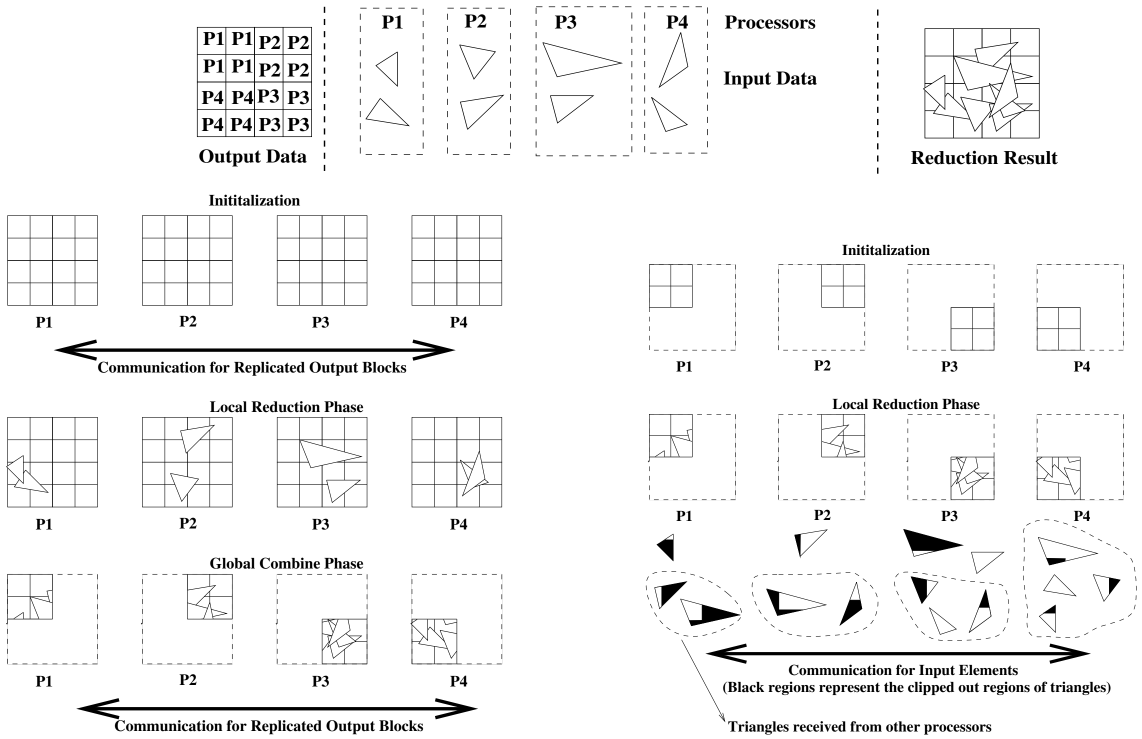


Figure 2: FRA strategy (left) and DA strategy (right).

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. On the other hand, it 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.

Figure 2 illustrates the FRA and DA strategies for an example application. One 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.

3 Analytical Cost Models

In earlier work [6, 14], we have shown that the relative performance of the query processing strategies changes under varying application characteristics and machine configurations. For example, Figure 3 shows an example mapping between input and output chunks using two different mapping functions. Circles denote the input chunks, whereas squares denote the output chunks. The arrows from input chunks to output chunks indicate the mapping between input and output datasets. Consider the volume of communication in both cases. For the sake of simplicity, assume that the size of an input chunk equals the size of an output chunk, O_{size}. In Figure 3(a), an

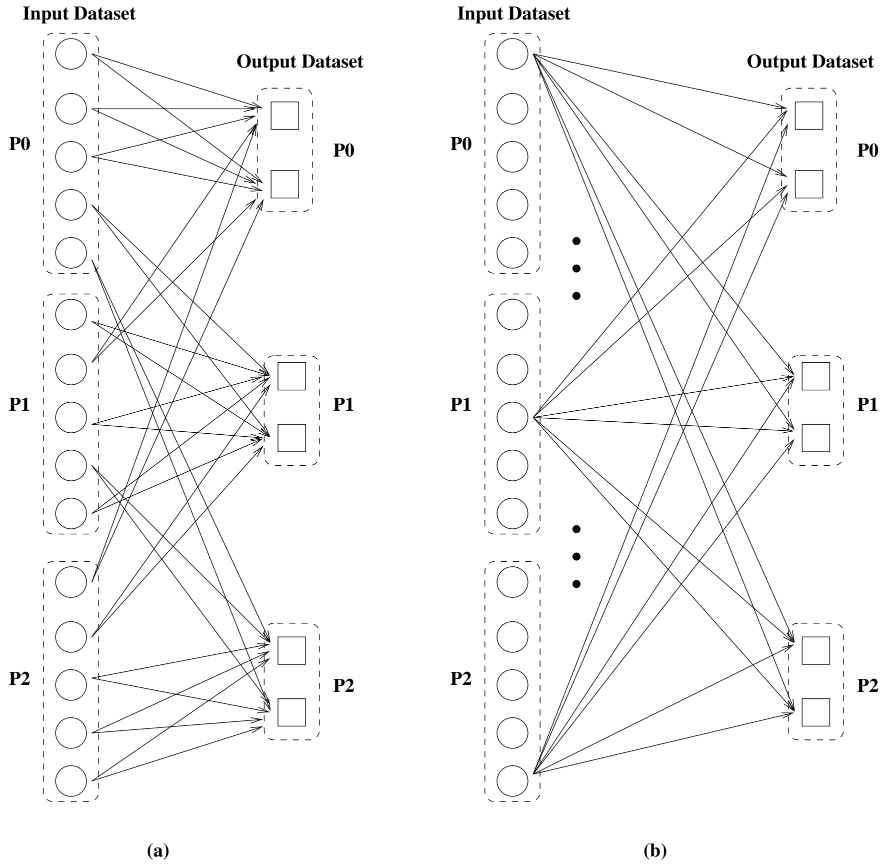


Figure 3: Mapping of input chunks to output chunks under two different mapping functions. Arrows from input chunks (circles) to output chunks (squares) denote the mapping between input and output.

input chunk maps to two output chunks, and each processor has to send 2 input chunks in the local reduction phase for the DA strategy. Therefore, the volume of communication per processor in DA is $2O_{size}$ in this case. At least one input chunk from each processor maps to an output chunk. As a result, each output chunk has to be replicated on all processors for the SRA and FRA strategies. The volume of communication per processor for the example, for FRA and SRA, is $4O_{size}$ for the initialization phase and $4O_{size}$ for the global combine phase. Therefore, DA requires less communication than SRA and FRA, potentially performing better than these strategies. In Figure 3(b), however, an input chunk maps to all of the output chunks, so each input chunk must be sent to two other processors. The volume of communication per processor for DA under this mapping is $5 * 2 * O_{size}$ (five input chunks to two other processors). The volume of communication for SRA and FRA, on the other hand, remains the same as for the first mapping, since each output chunk is replicated on each processor. Thus, in this case, SRA and FRA require less communication than DA.

In this section we present analytical models to predict the relative performance of the query processing strategies. Our goal is to predict the relative performance of the three strategies without running the query planning phase, i.e., without performing tiling and workload partitioning. Predicting the relative performance of the query processing strategies can be accomplished in two

Query Execution Phase	Strategy								
	FRA			SRA			DA		
	I/O	Comm.	Comp.	I/O	Comm.	Comp.	I/O	Comm.	Comp.
Initialization	$\frac{O_{fra}}{P}$	$\frac{O_{fra}}{P}(P-1)$	O_{fra}	$\frac{O_{sra}}{P}$	G	$\frac{O_{sra}}{P}+G$	$\frac{O_{da}}{P}$	0	$\frac{O_{da}}{P}$
Local Reduction	$\frac{I_{fra}}{P}$	0	$\frac{O_{fra}}{P}\beta$	$\frac{I_{sra}}{P}$	0	$\frac{O_{sra}}{P}\beta$	$\frac{I_{da}}{P}$	I_{msg}	$\frac{O_{da}}{P}\beta$
Global Combine	0	$\frac{O_{fra}}{P}(P-1)$	$\frac{O_{fra}}{P}(P-1)$	0	G	G	0	0	0
Output Handling	$\frac{O_{fra}}{P}$	0	$\frac{O_{fra}}{P}$	$\frac{O_{sra}}{P}$	0	$\frac{O_{sra}}{P}$	$\frac{O_{da}}{P}$	0	$\frac{O_{da}}{P}$

Table 1: The expected average number of I/O, communication, and computation operations per processor for a tile in each phase. O_{fra}, O_{sra}, and O_{da} denote the expected average number of output chunks per tile for the FRA, SRA, and DA strategies, respectively. Similarly, I_{fra}, I_{sra}, and I_{da} are the expected average number of input chunks retrieved per tile for the FRA, SRA, and DA strategies. G is the expected average number of ghost chunks per processor for a tile in SRA, and I_{msg} is the expected average number of messages per processor for input chunks in a tile for DA. The average number of output chunks that an input chunk maps to is denoted by α, and β represents the average number of input chunks that map to an output chunk. P is the number of processors executing the query.

steps. First, we must estimate the number of I/O, communication, and computation operations that must be performed for an output tile in each phase. Second, the counts must be used to produce an estimated execution time for each strategy.

Table 1 shows the expected average number of operations per processor for a tile in each phase. In the following sections we describe the methods used to compute the expected number of operations. The main assumption of the analytical models described in this paper is that the distribution of the input chunks in the output attribute space must be uniform, and the output dataset must be a regular d-dimensional dense array.

3.1 Computing Operation Counts for FRA

The number of tiles and the average number of output chunks in a tile depend on the aggregate system memory that can be effectively utilized by a query processing strategy. Since an output chunk is replicated in all processors for FRA, the effective system memory for FRA is the size of memory, M, on a single processor. Hence, the average number of output chunks per tile, O_{fra}, is $\frac{M}{O_{size}}$, and the number of tiles, T_{fra}, is $\frac{O}{O_{fra}}$, where O is the total number of output chunks, and O_{size} is the size of an output chunk. With β input chunks mapping to an output chunk, βO_{fra} computation operations are performed in the local reduction phase for each tile, and the declustering algorithm that ADR uses (see Section 2) is expected to assign an even share to each processor.

As was discussed in Section 2, an input chunk may intersect more than one output tile. To estimate the number of input chunks per output tile accurately, the number of output tiles an input chunk is expected to intersect has to be computed. Assume that the d-dimensional output grid

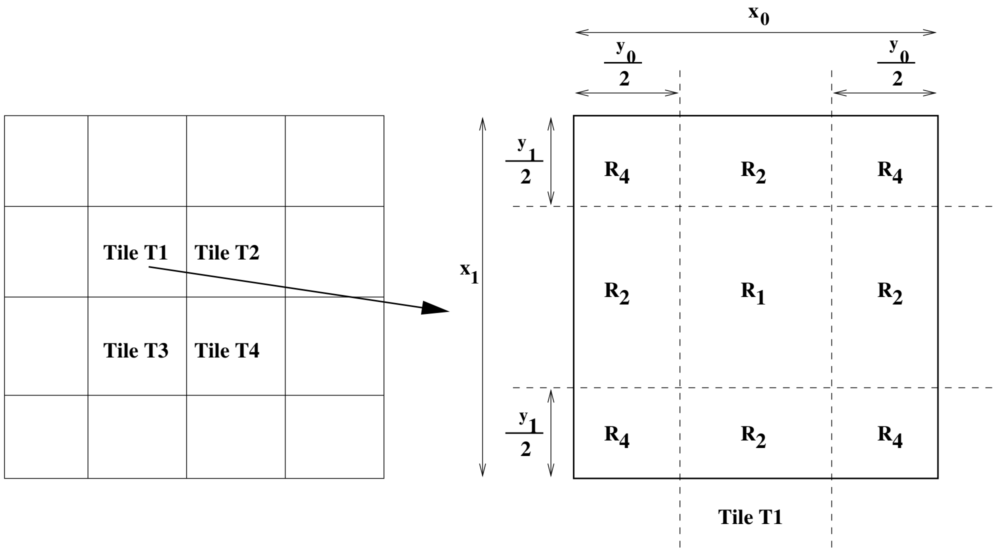


Figure 4: Partitioning of a tile ($T1$) into subregions R_1, R_2, and R_4.

is partitioned regularly into rectangular tiles and there are O_{fra} output chunks per output tile. Let each output chunk have a minimum bounding rectangle (MBR) of size z_i along dimension $i = 0, 1, \ldots, d-1$. Then, the extent of the MBR for an output tile in each dimension can be computed as $x_i = z_i n_i$ for $i = 0, 1, \ldots, d-1$, where n_i is the number of output chunks along dimension i of the tile ($n_i = \sqrt[d]{O_{fra}}$ for square tiles.) Also assume that after mapping to the output attribute space, the extent of the MBR of each input chunk is y_i along dimensions $i = 0, 1, \ldots, d-1$. In this paper d is assumed to be two, i.e., the output grid is two-dimensional, and $y_i < x_i$. The extension of the method to $d > 2$ dimensions and for $y_i \geq x_i$ can be found in [4].

A tile can be implicitly partitioned into subregions R_1, R_2, and R_4 as shown in Figure 4. If the mid-point of the MBR of an input chunk (with extents y_0 and y_1) falls into region R_k, then the input chunk intersects k adjacent output tiles. Let area(R_k) be the area of region R_k. Since we assume that the distribution of input chunks in the output space is uniform, the ratio of the area of region R_k to the total area of the tile can be used to estimate the number of input chunks that fall into that region. Thus, the expected number of output tiles that an input chunk intersects, λ, can be computed as

$$\lambda = \left(\frac{\text{area}(R_1)}{x_0 x_1}\right) + 2\left(\frac{\text{area}(R_2)}{x_0 x_1}\right) + 4\left(\frac{\text{area}(R_4)}{x_0 x_1}\right)$$

The expected number of input chunks that map to an output tile is then $I_{fra} = \frac{\lambda I}{T_{fra}}$, where I is the total number of input chunks.

3.2 Computing Operation Counts for SRA

Let e be the average percent of system memory used for local output chunks in an output tile. That is, if G is the average number of ghost chunks per processor per output tile, we have $e = \frac{O_{loc}}{O_{loc}+G}$. Here, O_{loc} is the number of local (non-ghost) output chunks in the tile. Note that we have $\frac{1}{P} \leq e \leq 1$, and when $e = \frac{1}{P}$, SRA is equivalent to FRA. Given the value of e, the number of output chunks in a tile is $O_{sra} = \frac{ePM}{O_{size}}$, the number of local output chunks per processor in the tile is $O_{loc} = \frac{O_{sra}}{P}$, and the number of tiles is $T_{sra} = \frac{O}{O_{sra}}$.

We compute G and e as follows. The goal of the declustering algorithms used in ADR [10, 16] is to achieve good I/O parallelism when retrieving input and output chunks from disks. To achieve this goal, the algorithms distribute spatially close chunks evenly across as many processors as possible. Therefore, β input chunks that map to an output chunk v on processor p can be expected to be distributed across as many processors as possible. Let G' be the average number of ghost chunks that are created for one output chunk. Then, on average a processor creates a total of $G = G'O_{loc}$ ghost chunks per output tile, and P processors create $G'O_{loc}P$ ghost chunks per output tile.

Under the assumption that input chunks that map to the same output chunk are distributed across as many processors as possible, SRA becomes FRA if $\beta \geq P$. When $\beta < P$, we have G' = (Probability$\{p$ is one of β processors$\}$(β - 1) + Probability$\{p$ is not one of β processors$\}\beta$) $= \beta\frac{P-1}{P}$, and hence,

$$e = \frac{O_{loc}}{O_{loc} + G'O_{loc}} = \frac{1}{1+G'} = \frac{P}{P + \beta(P-1)}$$

$$G = G'O_{loc} = \frac{\beta(P-1)}{P} \frac{PM}{O_{size}[P+\beta(P-1)]} = \frac{\beta M(P-1)}{O_{size}[P+\beta(P-1)]}$$

The number of input chunks that map to a tile, I_{sra}, and the number of computation operations are calculated the same way as for FRA, as shown at the end of Section 3.1.

3.3 Computing Operation Counts for DA

For DA, output chunks are not replicated, so the effective overall system memory is $P * M$, where M is the size of the memory on each processor. Therefore, the average number of output chunks per tile can be computed as $O_{da} = \frac{PM}{O_{size}}$, and the number of tiles is $T_{da} = \frac{O}{O_{da}}$. The estimated number of input chunks that map to a tile, I_{da}, is again computed as for FRA in Section 3.1.

In the local reduction phase for the DA strategy, local input chunks that map to output chunks on other processors must be sent to those processors. As a result, DA requires interprocessor communication for input chunks. We estimate the number of messages for input chunks in each processor, I_{msg}, as follows. We partition a tile into regions R_1, R_2, and R_4 as is described in Section 3.1 (see Figure 4). Let $\mathcal{C}(\alpha, P)$ be defined as

$$\mathcal{C}(\alpha, P) = \begin{cases} P-1 & \text{if } \alpha \geq P \\ \alpha\frac{P-1}{P} & \text{otherwise} \end{cases}$$

If the mid-point of the MBR of an input chunk falls inside region R_1, the input chunk is entirely contained by that output tile. Remember that the α output chunks that the input chunk maps to are declustered across as many processors as possible, as are the input chunks. That is, if $\alpha \geq P$, then the output chunks are stored on $P-1$ processors; otherwise, they are stored on α processors. Therefore,

$$\text{expected number of messages for an input chunk in region } R_1 = \mathcal{C}(\alpha, P)$$

If the mid-point of the MBR for an input chunk falls inside one of the four R_2 regions, it maps to two output tiles, say $T1$ and $T2$ (see Figure 4). Since we assume a uniform distribution of input chunks in the output space, the number of output chunks that the input chunk maps to in the two output tiles is proportional to the area of the input chunk inside each output tile. The average area inside $T2$ of all the input chunks in region R_2 is expected to be $\frac{1}{4}$ of the total area of the input chunks in that region. Therefore,

$$\text{number of output chunks in } T2 \text{ that an input chunk maps to} = \alpha \frac{\frac{y_0}{4} y_1}{y_0 y_1} = \frac{\alpha}{4}$$

Similarly, the number of output chunks in $T1$ that the input chunk maps to is $\frac{3}{4}\alpha$. For input chunks that fall inside one of the four R_4 regions in $T1$, each input chunk maps to three other output tiles, say $T2$ (to the right of $T1$), $T3$ (below $T1$), and $T4$ (diagonally across from $T1$) in Figure 4. An analysis similar to one for region R_2 shows that the input chunk maps to $\frac{9}{16}\alpha$, $\frac{3}{16}\alpha$, $\frac{3}{16}\alpha$, and $\frac{1}{16}\alpha$ output chunks in output tiles $T1$, $T2$, $T3$, and $T4$, respectively. We can therefore compute the expected number of input chunk messages in each region as

$$\begin{aligned}
\text{number of messages due to region } R_1 &= \mathcal{C}(\alpha, P) \\
\text{number of messages due to region } R_2 &= \mathcal{C}(\frac{3}{4}\alpha, P) + \mathcal{C}(\frac{1}{4}\alpha, P) \\
\text{number of messages due to region } R_4 &= \mathcal{C}(\frac{1}{16}\alpha, P) + 2 * \mathcal{C}(\frac{3}{16}\alpha, P) + \mathcal{C}(\frac{9}{16}\alpha, P)
\end{aligned}$$

Given that each processor owns in average $\frac{I_{da}}{P}$ input chunks for a tile, the expected number of input chunk messages for a processor per tile is then

$$\begin{aligned}
I_{msg} &= \frac{I_{da}}{P}(\frac{\text{area}(R_1)}{x_0 x_1}\mathcal{C}(\alpha, P) + \frac{\text{area}(R_2)}{x_0 x_1}[\mathcal{C}(\frac{3}{4}\alpha, P) + \mathcal{C}(\frac{1}{4}\alpha, P)] \\
&\quad + \frac{\text{area}(R_4)}{x_0 x_1}[\mathcal{C}(\frac{1}{16}\alpha, P) + 2 * \mathcal{C}(\frac{3}{16}\alpha, P) + \mathcal{C}(\frac{9}{16}\alpha, P)])
\end{aligned}$$

3.4 Estimating Execution Times

After the counts for each operation and each phase of query execution are calculated, the counts must be used to produce estimated execution times to predict the relative performance of the various strategies. We briefly describe our initial approach for estimating the relative execution time of the strategies.

We do not require estimating the absolute execution time of each strategy accurately. Our goal is to estimate the relative performance of the strategies so that the best strategy for an application and machine configuration is chosen, especially when one strategy performs significantly better than the others, and to compute this estimation without much overhead. For this reason, we use a simple method to compute execution times of the strategies. I/O and communication counts per processor in Table 1 are first converted into I/O and communication volumes by multiplying them by the average input and output chunk sizes. The communication and I/O times per processor per phase are computed by dividing the respective communication and I/O volumes by the measured communication and I/O bandwidths of the target machine. The computation cost in each phase is the computation count per processor times the cost of processing an input or output chunk in each phase. The total execution time is then the sum of the estimated times for communication, I/O and computation in each phase of query execution.

4 Experimental Results

In this section we present experimental evaluation of the cost models 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 220MB of that space for the input dataset and 50MB 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 230MB on the disk to clean the file cache before each experiment to obtain reliable performance results.

In the experiments we first use synthetic datasets to evaluate the cost models under controlled scenarios. The output dataset is a 2D rectangular array. The entire output attribute space is regularly partitioned into non-overlapping rectangles, with each rectangle representing an accumulator chunk in the output dataset. The input dataset has a 3D attribute space, and input chunks were placed in the input space randomly with a uniform distribution. The assignment of input and output chunks to the disks was done using a Hilbert curve-based declustering algorithm [10]. In these experiments the size of the input and output datasets were fixed. The output dataset size is set at 400MB, with 1600 output chunks. The input dataset size is set at 1.6GBytes. We varied several of the input dataset parameters, to show that DA performs better than the other two strategies in some cases, while SRA performs better for other cases. There are several parameters (e.g., size of datasets, number of chunks in a dataset, extent of a chunk in the dataset) that can be varied to create different scenarios. As was previously discussed, communication for SRA and FRA results from output chunks that are replicated on multiple processors, whereas input chunks are communicated for DA. The value of β, the average number of input chunks that map to an output chunk, determines the volume of communication in SRA, whereas the volume of communication in DA is affected by the value of α, the average number of output chunks that an input chunk maps to. We varied the number and extent of input chunks to create different β and α values. A large β value means that each output chunk is likely to be replicated on all processors, thereby increasing the volume of communication for SRA. On the other hand, a small β value may not require an output chunk to be replicated on all processors. Similarly, a large α value means that an input chunk is likely to be sent to many processors for DA, thus increasing the volume of communication.

In these experiments we varied the number and extent of input chunks to produce (α, β) pairs of $(9, 72)$ and $(16, 16)$. We set the computation time to 1 millisecond for processing an output chunk in the initialization, global combine, and output handling phases, and to 5 milliseconds for processing each intersecting (input,output) chunk pair in the local reduction phase. In selecting the cost for the local reduction phase computation, we chose a value that is larger than the computation costs in the initialization and global combine phases, but also is small enough so that the computation cost relative to communication and I/O for the queries is not too large. Otherwise, the queries become very computationally intensive, and the performance of the strategies is affected only by computational load balance across the processors.

Note that the average input and output chunk size, α and β values, and I/O and communication

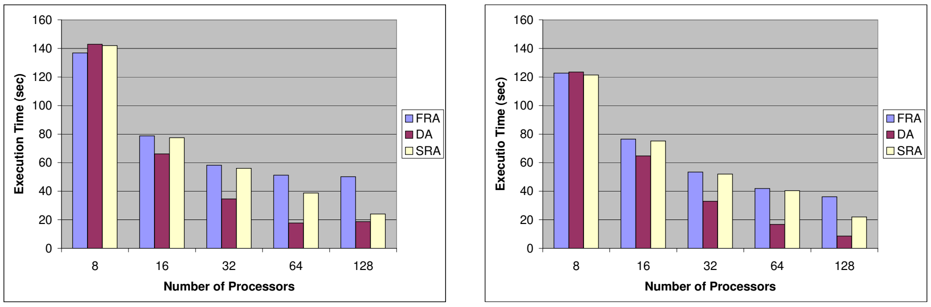


Figure 5: Measured (left) and estimated (right) total execution time for queries with $(\alpha,\ \beta) = (9, 72)$.

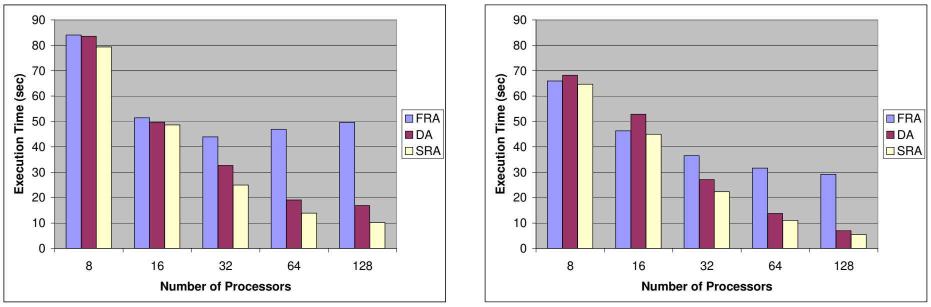


Figure 6: Measured (left) and estimated (right) total execution time for queries with $(\alpha,\ \beta) = (16, 16)$.

bandwidths in each phase are needed to evaluate the cost models described in Section 3, to compute estimated execution times. In this work I/O and communication bandwidths were measured by running a set of queries (including the queries in Figures 5 and 6) on the target machine and taking the average value across these queries. These values were used to estimate the execution times of the query strategies across all queries. For a real application the same strategy may be applied; the user may run several sample queries to compute the average application level I/O and communication bandwidths. Since the values of α and β depend on the mapping function specified for a range query (see Figure 1), the values must be computed for each query. This can be accomplished using the minimum bounding rectangle (MBR) of each input and output chunk that intersects the query. The MBR of each input chunk is mapped to output chunks via the mapping function, and the value of α for the input chunk is computed by counting the number of output chunks the input chunk maps to. The average α is calculated as the average of α values over all input chunks. The average β value can be computed from the equation $I\alpha = O\beta$, where I is the number of input chunks, and O is the number of output chunks (i.e. the total number of chunks sent is the same as the total number of chunks received, across all processors).

Figure 5 shows the measured and estimated query execution times of the query strategies for cases in which DA performs better than the other strategies. Figure 6 shows the measured and estimated query execution times for cases in which SRA performs best. As is seen from the figures the cost models can estimate the relative performance of the strategies under varying scenarios. Figures 7(a)-(d) show breakdowns of the measured and estimated values into computation time, I/O volume, and communication volume. As is seen from the figures, the cost models are able

App.	Input Dataset Num. of Chunks	Input Dataset Total Size	Output Dataset Num. of Chunks	Output Dataset Total Size	Average β	Average α	Computation (in milliseconds) I–LR–GC–OH
SAT	9K	1.6GB	256	25MB	161	4.6	1–40–20–1
WCS	7.5K	1.7GB	150	17MB	60	1.2	1–20–1–1
VM	16K	1.5GB	256	192MB	64	1.0	1–5–1–1

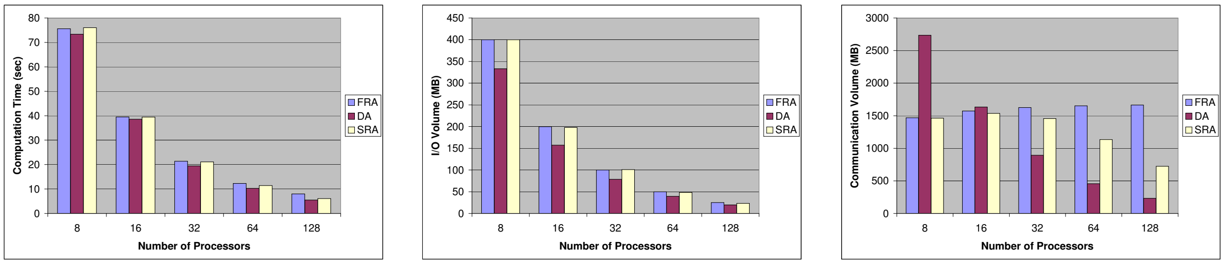
Table 2: Application characteristics.

to estimate relative computation time, I/O volume, and communication volume for the query processing strategies for different α and β values for varying number of processors. The cost model for DA does not accurately estimate the communication volume for 16 processors, as seen in Figure 7(d). This is because the cost model assumes perfect declustering of the output chunks that an input chunk maps to. Thus, with $\alpha = 16$, an input chunk on one processor is expected to be sent to fifteen other processors. In practice, however, perfect declustering is not achieved, and an input chunk is sent to fewer than fifteen processors. As a result, the actual communication volume is less than what the cost model predicts.

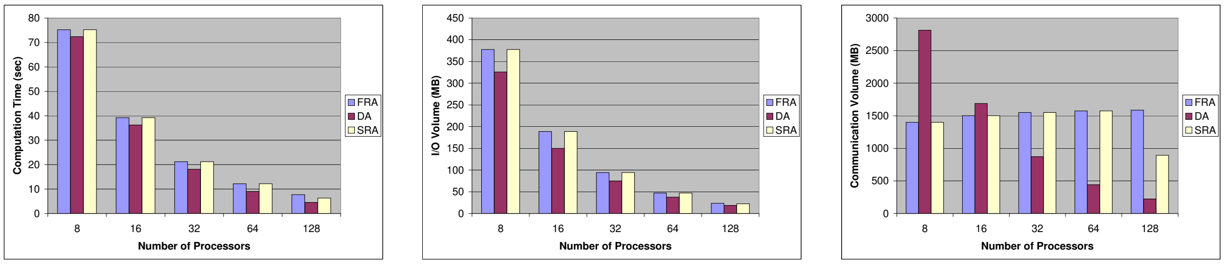
We have also evaluated the cost models for different application scenarios, varying the number of processors. We used *application emulators* [26] to generate various application scenarios for the applications classes that motivated the design of ADR (see Section 1). An application emulator provides 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.

Table 2 summarizes dataset sizes and application characteristics for three application classes; *satellite data processing* (SAT) [7], analysis of microscopy data with the *Virtual Microscope* (VM) [1], and *water contamination studies* (WCS) [15]. The output dataset size was a fixed size for each application. The last column shows the computation time per chunk for the different phases of query execution (see Section 2); 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 for SAT is irregular. This is because of the polar orbit of the satellite [18]; 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 the poles. The input datasets for WCS and VM are regular dense arrays that are partitioned into equal-sized rectangular chunks. We selected the values for the various parameters to represent some typical scenarios for these application classes on the SP machine, based on our experience with the complete ADR applications.

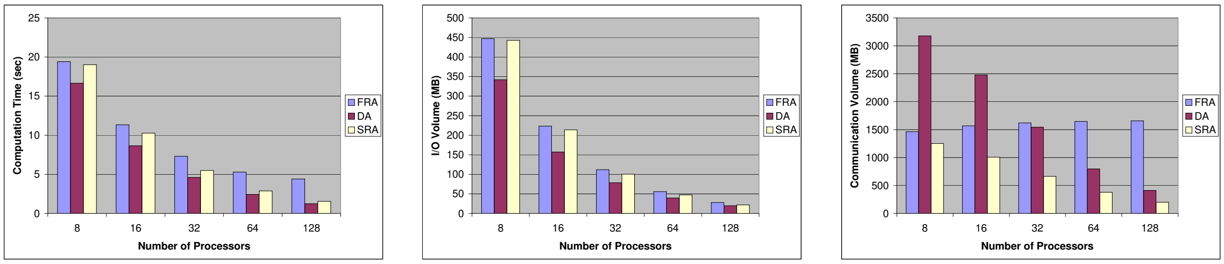
Measured and estimated values for computation time, I/O volume and communication volume for each application are shown in Figures 8–10. As is seen from the figures, the cost models are able



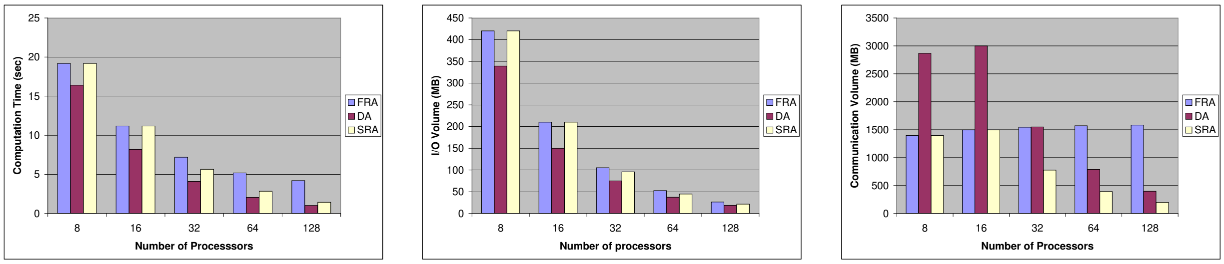
(a) Measured, $(\alpha,\ \beta) = (9, 72)$.



(b) Estimated, $(\alpha,\ \beta) = (9, 72)$.



(c) Measured, $(\alpha,\ \beta) = (16, 16)$.



(d) Estimated, $(\alpha,\ \beta) = (16, 16)$.

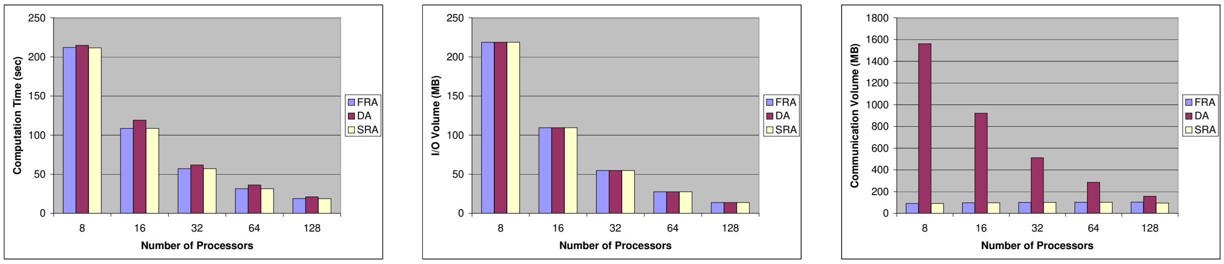
Figure 7: Measured and estimated computation time (left), I/O volume (middle), and communication volume (right), for different queries on different number of processors.

to estimate the relative volume of I/O and communication in each application for varying number of processors. However, the cost models fail to estimate the relative computation times of the strategies for the SAT and WCS applications. Our experiments show that in these two applications there is a load imbalance in the computation assigned to the various processors. There are two main reasons for the load imbalance in these applications. First, the distribution of data elements in the output attribute space is not uniform for SAT. Second, the Hilbert curve-based declustering algorithms do not achieve optimal distribution of the input and output chunks across the processors, causing load imbalance in some cases. Since the cost models assume perfect declustering and a uniform distribution of the computations across the processors, the models may fail to predict the relative computation times of the strategies in those cases. Figure 11 shows the measured and estimated total execution time for each application. As is seen from the figure, the cost models can successfully predict the relative performance of the strategies for the VM application, which has a uniform distribution of input and output chunks. For the SAT and WCS applications, however, the cost models fail to predict the relative performance of the strategies in some cases. One of the reasons is the load imbalance in computation as is mentioned previously. Another reason is that I/O and communication bandwidths may vary across applications and across different number of processors. We used the I/O and communication bandwidths computed from synthetic datasets for estimating the total execution time of the applications. We observed that there can be a large difference between the bandwidths measured from the synthetic datasets and the bandwidths measured in some of the runs of the WCS application. We plan to further investigate these cases to understand why I/O and communication bandwidths may change by large amounts across different applications and different number of processors, and look into approaches to estimate such changes to make our cost models more accurate.

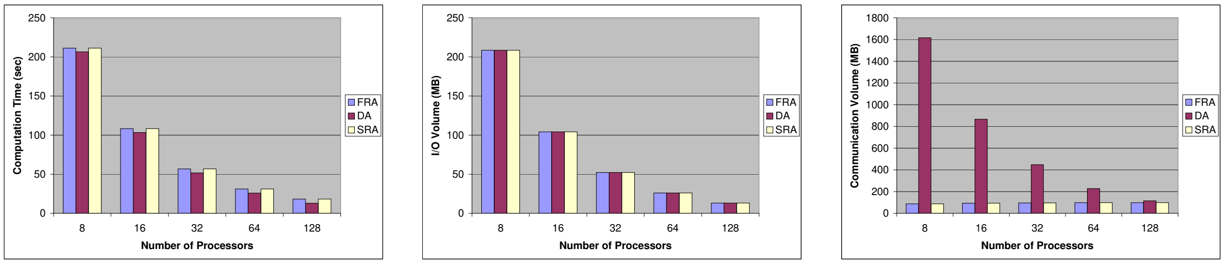
5 Related Work

Several runtime support libraries and file systems have been developed to support efficient I/O in a parallel environment [2, 8, 12, 13, 19, 21, 24, 25]. These systems mainly focus on supporting regular strided access to uniformly distributed datasets, such as images, maps, and dense multi-dimensional arrays. 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.

Parallel database systems have been a major topic in the database community [9] for a long time, and much attention has been devoted to the implementation and scheduling of parallel joins [17, 20]. 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 [3, 23]. 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 [22]. ADR,

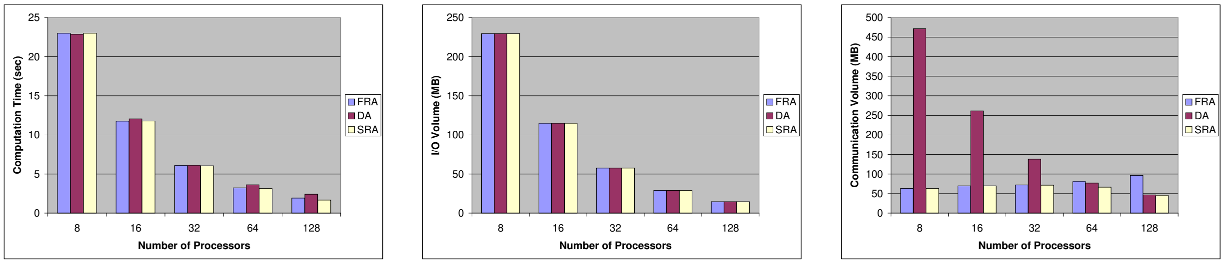


(a) Measured.

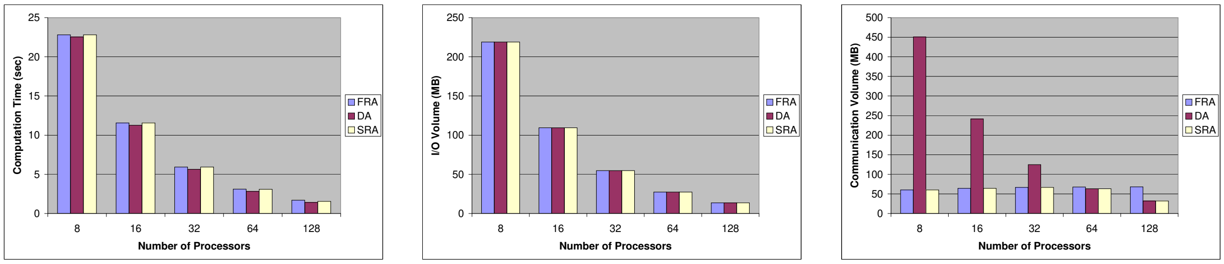


(b) Estimated.

Figure 8: Measured and estimated computation time (left), I/O volume (middle), and communication volume (right) for SAT application.

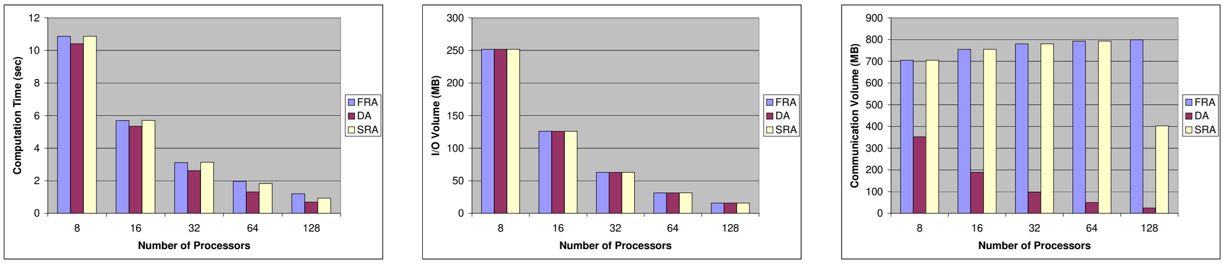


(a) Measured.

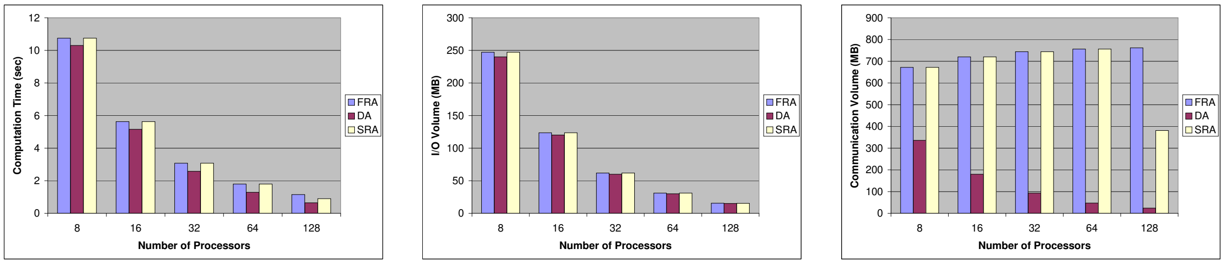


(b) Estimated.

Figure 9: Measured and estimated computation time (left), I/O volume (middle), and communication volume (right) for WCS application.

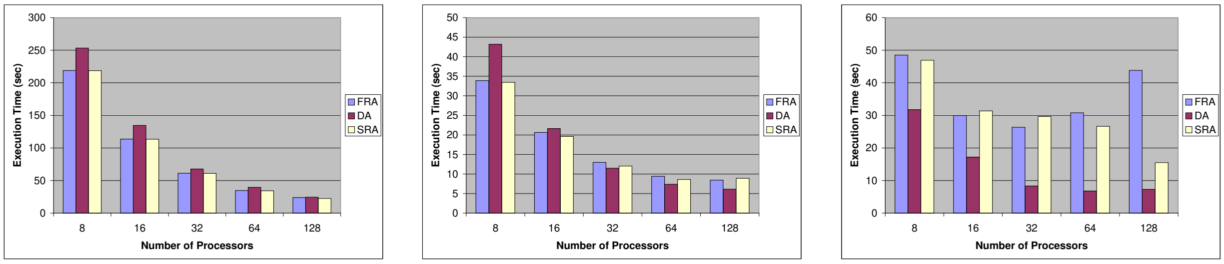


(a) Measured.

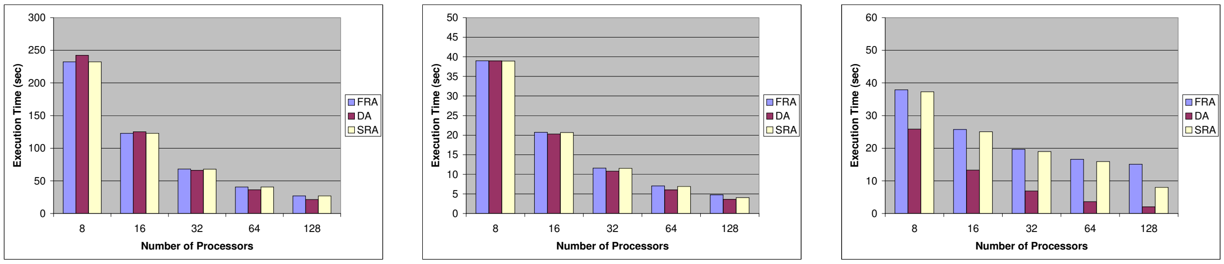


(b) Estimated.

Figure 10: Measured and estimated computation time (left), I/O volume (middle), and communication volume (right) for VM application.



(a) Measured.



(b) Estimated.

Figure 11: Measured and estimated total execution times for SAT (left), WCS (middle) and VM (right) applications.

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

We have described an approach to guide and automate selection of the best query processing strategy for a given query within the ADR framework. We have presented analytical cost models to predict the relative performance of three query processing strategies for different application scenarios and machine configurations. Our results on a 128-node IBM SP show that in most cases the cost models are able to predict the relative performance of the strategies, both for synthetic datasets and for our driving applications. However, our cost models can fail when there is a significant computational load imbalance or when there is a large variance in measured I/O and communication costs on the parallel machine, because the current models assume both a computational load balance and fixed, predictable I/O and communication bandwidth from the machine. We plan to further investigate these limitations and devise approaches to accurately model the costs of the query execution strategies for a wider range of applications and machine configurations.

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] 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.

[4] C. Chang. Cost models for query processing strategies in Active Data Repository. Technical Report CS-TR-4060 and UMIACS-TR-99-54, University of Maryland, Department of Computer Science and UMIACS, 1999.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

[11] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM-SIGMOD Conference*, pages 47–57, Boston, MA, June 1984.

[12] 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.

[13] 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.

[14] T. Kurc, C. Chang, R. Ferreira, A. Sussman, and J. Saltz. Querying very large multi-dimensional datasets in ADR. Technical Report CS-TR-4022 and UMIACS-TR-99-29, University of Maryland, Department of Computer Science and UMIACS, May 1999. To appear in SC'99.

[15] 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.

[16] 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.

[17] M. C. Murphay and D. Rotem. Multiprocessor join scheduling. *IEEE Transactions on Knowledge and Data Engineering*, 5(2):322–338, Apr. 1993.

[18] 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.

[19] N. Nieuwejaar and D. Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[20] 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.

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

[22] P. Seshadri, M. Livny, and R. Ramakrishnan. The case for enhanced abstract data types. In *Proceedings of the 23th VLDB Conference*, Athens, Greece, Aug. 1997.

[23] M. Stonebraker, L. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, Mar. 1990.

[24] 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.

[25] 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.

[26] 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.