Optimizing Reduction Computations In a Distributed Environment *

Tahsin Kurc†, Feng Lee*, Gagan Agrawal*, Umit Catalyurek†, Renato Ferreira†, Joel Saltz†
†Department of Biomedical Informatics
*Department of Computer and Information Sciences
Ohio State University
Columbus OH 43210
{kurc,umit,jsaltz}@bmi.ohio-state.edu {lif,agrawal}@cis.ohio-state.edu

ABSTRACT

We investigate runtime strategies for data-intensive applications that involve generalized reductions on large, distributed datasets. Our set of strategies includes replicated filter state, partitioned filter state, and hybrid options between these two extremes. We evaluate these strategies using emulators of three real applications, different query and output sizes, and a number of configurations. We consider execution in a homogeneous cluster and in a distributed environment where only a subset of nodes host the data. Our results show replicating the filter state scales well and outperforms other schemes, if sufficient memory is available and sufficient computation is involved to offset the cost of global merge step. In other cases, hybrid is usually the best. Moreover, in almost all cases, the performance of the hybrid strategy is quite close to the best strategy. Thus, we believe that hybrid is an attractive approach when the relative performance of different schemes cannot be predicted.

1. INTRODUCTION

One of the major new developments in computer and computational sciences is the emergence of remote and distributed data repositories. This development is driven by the need and feasibility of conducting *data-driven science* [13], where sharing and analysis of large scientific datasets is used to facilitate and accelerate scientific advances. With the increasing sizes of datasets, it is generally impossible to download a dataset and process it locally. Thus, data has to be accessed and processed in a distributed environment. The emergence of grid environments [20, 19] gives us an unprecedented opportunity to flexibly use distributed resources for analyzing datasets resident in remote data repositories. In recent years, a number of research projects have focused on providing services for resource discovery, authentication and authorization, allocation of resources, and secure, efficient, and reliable access to resources [13, 18, 19, 32]. However, it still remains a challenging research problem to efficiently execute applications in a Grid environment.

In this paper, we develop and evaluate execution strategies for applications that process data resident in repositories. Our work is based upon the observation that data-driven applications from many domains including scientific data analysis, data mining, visualization, and image analysis [7, 1, 16, 27, 26, 12, 25, 2, 24], frequently involve a common type of aggregation operations referred to as *generalized reductions*.

The most important characteristic of generalized reductions is that the aggregation operations are *commutative* and *associative*, i.e., the same output value is computed irrespective of the order the input data items are aggregated. This allows for different parallelization strategies. This paper presents a set of techniques: replicated filter state (RFS), partitioned filter state (PFS), and a hybrid scheme which combines the two. Two of the strategies are based on the replicated and distributed accumulator strategies in our earlier work [11, 26]. In that work, we evaluated strategies for an SPMD style program on tightly-coupled, homogeneous systems. We investigated application of a hypergraph partitioning strategy on these two approaches in [10] for partitioning the workload among the nodes on a homogeneous, distributed memory machine. Although hypergraph partitioning strategy achieves good load balance and reduces communication, it is an expensive algorithm. This work differs from the earlier work in the following ways. 1) We discuss how these techniques can be developed in the context of a component-based framework designed for application development in distributed and heterogeneous environments. 2) We propose a hybrid strategy that combines the salient features of the two schemes by first regularly partitioning the accumulator among processor groups and replicating the accumulator pieces within a group. 3) We consider scalability in a cluster environment as well as performance in a distributed environment where data is distributed between a set of nodes, and another cluster or set of nodes is used for processing.

We report on a detailed experimental study to under-

*This research was supported by the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408), #EIA-0121177, #EIA-0203846, #ACI-0130437, #ACI-9982087, Lawrence Livermore National Laboratory under Grant #B500288 and #B517095 (UC Subcontract #10184497).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC'03 November 15-21, 2003, Phoenix, Arizona, USA
Copyright 2003 ACM 1-58113-695-1/03/0011 ...$5.00.

```
O ← Output Dataset, I ← Input Dataset
1.  [S_I] ← Intersect(I, Query)
    (* Initialization *)
2.  foreach o_e in S_O do
3.      a_e ← Initialize(o_e)
    (* Reduction *)
4.  foreach i_e in S_I do
5.      read i_e
6.      S_A ← Map(i_e) ∩ S_O
7.      foreach a_e in S_A do
8.          a_e ← Aggregate(i_e, a_e)
    (* Finalization *)
9.  foreach a_e in S_O do
10.     o_e ← Output(a_e)
```

Figure 1: The basic processing loop for generalized reduction operations.

stand the performance trade-offs between these techniques. We use application emulators based upon three real scientific data processing applications and queries that process up to several gigabytes of data. The key observations from our experiments are as follows. Replicating the filter state or accumulator scales well and outperforms other schemes if 1) sufficient memory is available to replicate the accumulator, and 2) sufficient computation is involved to offset the cost of merging the replicated accumulators. In other cases, the hybrid strategy is usually the best. Moreover, in almost all cases, the performance of the hybrid strategy is quite close to the best strategy. Thus, we believe that the hybrid scheme is an attractive approach when the relative performance of different schemes cannot be predicted.

The rest of the paper is organized as follows. We present a more detailed description of reduction operations in Section 2. The algorithms developed in this paper is implemented on a component-based system, called DataCutter. Section 3 gives an overview of this system. Different strategies for executing reductions and their implementation using DataCutter are described in Section 4. Our experimental study is presented in Section 5. We compare our work with related research efforts in Section 6 and conclude in Section 7.

2. REDUCTION OPERATIONS

Typical examples of applications that make use of large scientific datasets include satellite data processing applications, analysis of microscopy data, and simulation systems for water contamination studies. These and many other scientific data analysis applications employ *generalized reduction operations* as their core processing structure. Figure 1 shows the pseudo-code for a generalized reduction operation.

The input data items selected by a query are retrieved from storage system and mapped to the corresponding output items (steps 5 and 6). For instance, in a typical analysis of remotely-sensed satellite data, a range query defines a bounding box that covers a part or all of the surface of the earth over a period of time [12]. Data items retrieved from one or more datasets are processed to generate one or more composite images of the area under study. The mapping function, $Map(i_e)$, may map an input data item to a set of output data items. In satellite data processing, generating a composite image requires projection of the selected area of the earth onto a two-dimensional grid. The mapping step is followed by steps that aggregate all the input data items that map to the same output data items (steps 7 and 8). Each pixel in the composite image is computed by selecting the "best" sensor value that maps to the associated grid point. The aggregation operation at step 8 is an *associative* and *commutative* operation. That is, the result of the aggregation is independent of the order input data items are processed. An intermediate data structure, referred to as an *accumulator*, can be used to hold intermediate results during processing. 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). Final results are produced by post-processing the intermediate results maintained in the accumulator.

The implementation of generalized reduction operations in a distributed, and potentially heterogeneous, environment requires distribution of data and computations to efficiently utilize aggregate storage space and computing power, and to minimize I/O and communication overheads. A possible approach is to make combined use of task- and data-parallelism. The basic processing steps (i.e., Select, Read, Map, Aggregate, Output) in generalized reductions can be implemented as a set of connected components. For example, Select, Read, and Map operations could be implemented as a single component and run on the machines where the dataset is stored. The Aggregate operation could be a component too. Note that the accumulator structure stores the intermediate results during data processing in the Aggregate component. In this respect, the state of the component is maintained in the accumulator data structure. If aggregation operations involve a sequence of transformation and reduction operations. These operations could be implemented as separate components and executed on machines with powerful processors. Multiple copies of the aggregate component could be executed to achieve data parallelism. Similarly, the Output operation could be another component. In the next section, we describe a framework and runtime system that supports development of component-based data-intensive applications and combined task- and data-parallelism.

3. THE RUNTIME ENVIRONMENT

We have evaluated the strategies described in this paper using a component-based framework. This framework, called DataCutter [4], supports a filter-stream programming model for developing data-intensive applications that execute in a distributed, heterogeneous environment. In this model, the application processing structure is implemented as a set of components, referred to as *filters*, that exchange data through a *stream* abstraction. The interface for a *filter*, consists of three functions: (1) an initialization function (*init*), in which any required resources such as memory for data structures are allocated and initialized, (2) a processing function (*process*), in which user-defined operations are applied on data elements, and (3) a finalization function (*finalize*), in which the resources allocated in *init* are released. Filters are connected via *logical streams*. A *stream* denotes a uni-directional data flow from one filter (i.e., the producer) to another (i.e., the consumer). A filter is required to read data from its input streams and write data to its output streams only. The current implementa-

tion of the logical stream delivers data in data buffers, and uses TCP for point-to-point stream communication.

The overall processing structure of an application is realized by a *filter group*, which is a set of filters connected through logical streams. When a filter group is instantiated to process an application query, the runtime system establishes socket connections between filters placed on different hosts before starting the execution of the application query. Filters placed on the same host execute as separate threads. An application query is handled as a *unit of work* (UOW) by the filter group. An example is a visualization of a MRI dataset from a viewing angle. The processing of a UOW can be done in a pipelined fashion; different filters can work on different data elements simultaneously.

The programming model provides several abstractions to facilitate performance optimizations. A *transparent filter copy* is a copy of a filter in a filter group. The filter copy is transparent in the sense that it shares the same *logical* input and output streams of the original filter. A transparent copy of a filter can be made if the semantics of the filter group are not affected. That is, the output of a unit of work should be the same, regardless of the number of transparent copies. The filter runtime system maintains the illusion of a single logical point-to-point stream for communication between a logical producer filter and a logical consumer filter. It is responsible for scheduling elements (or buffers) in a data stream among the transparent copies of a filter. For distribution between transparent copies, the runtime system supports a Round-Robin (RR) mechanism and a Demand Driven (DD) mechanism based on buffer consumption rate. DD aims to send buffers to the filter that will process them fastest. When a consumer filter starts processing of a buffer received from a producer filter, it sends an acknowledgment message to the producer filter to indicate that the buffer is being processed. A producer filter chooses the consumer filter with the minimum number of unacknowledged buffers to send a data buffer to, thus achieving a better balancing of the load.

DataCutter provides support for task- and data-parallelism. An application can be built from a set of components allowing task-parallelism. The transparent copies enable data-parallelism for execution of a single query, while multiple filter groups allow concurrency among multiple queries. In the next section, we describe the distributed algorithms for generalized reduction operations and discuss their implementation as DataCutter components (filters).

4. PROCESSING STRATEGIES

As we discussed in Section 2, the basic processing steps (i.e., Select, Read, Map, Aggregate) in generalized reductions can be implemented as components (filters in DataCutter). For efficient performance, the relative processing time of the components should be the same, and the time of each stage should be balanced with respect to the communication cost between components. However, because of application characteristics, or the heterogeneous nature of the environment, or a suboptimal placement of components, processing is oftentimes not well balanced. This imbalance and resulting performance penalty can be addressed using parallelism, by executing multiple copies of a single filter on a single machine or across a set of host machines. In this paper, we look at three strategies for employing parallelism in processing of reduction operations. Two of the strategies described in this paper are based on the replicated and distributed accumulator methods developed in our earlier work (see Figure 2) [11, 26]. The third strategy combines the two strategies into a hybrid scheme.

4.1 Replicated Filter State (RFS)

This approach replicates a filter and its state (i.e. data structures that maintain the accumulator) on a single machine or across multiple machines. Figure 3 illustrates the replicated filter state strategy (RFS) for an application. The application processing structure in this example is decomposed into a pipeline of three filters: (1) a data retrieval filter (**R**), which retrieves data items that intersect a query, (2) an aggregation filter (**A**), which implements the *Map* and *Aggregate* operations, and (3) an output filter (**O**), which computes the final output from the intermediate results. The aggregation filter maintains an accumulator structure, denoted by rectangles in the figure, for intermediate results. In a distributed setting, multiple copies of **R** and **A** filters can be created either explicitly or using the transparent copies abstraction of the DataCutter framework. Since we assume that operations on data are commutative and associative, a data buffer, containing input data elements, from filter **R** can be consumed by any copy of filter **A**. This leads to two main configurations. An **A** filter can be colocated with a **R** filter (Figure 3(a)). Data buffers from a **R** filter are consumed by the colocated **A** filter only. The advantage of this strategy is that no interprocessor communication is required for input data. This strategy is well suited for homogeneous systems and when data is evenly distributed across all the nodes in the system. The second strategy allows a buffer from a **R** filter to be consumed by any copy of filter **A** (Figure 3(b)). The scheduling of buffers can be done using one of the policies described in Section 3. This strategy is particularly suitable when the application is executed in a heterogeneous environment. However, it is likely to incur communication overhead because of input data buffers sent to non-local copies of filter **A** and due to the acknowledgment messages in the demand driven policy.

In RFS, when a copy of filter **A** is instantiated, the internal state of the filter, i.e., the accumulator data structure, is replicated. Thus, the filter will not operate correctly in parallel when its copies are executed, because of internal state. For example, a filter that attempts to compute the average size of all buffers will not arrive at the correct answer, because only a subset of all buffers for a given unit-of-work are seen at any one copy, hence the internal sum of buffer sizes is less than the true total. Such cases require an additional application-specific *merge* (**M**) filter be appended to the stage after the replicated filter so as to merge partial intermediate results into the final intermediate result.

In the RFS strategy, the overall processing of data is effectively divided into two phases: a *reduction* phase, in which data is processed by transformation and aggregation filters, and a *global combine* phase, in which partial results are merged into the final output. If large amounts of geographically distributed input data are processed, by performing most of the data reduction locally, the replicated filter scheme can reduce the volume of communication significantly. Nevertheless, the global combine phase is an overhead introduced due to parallelization. Therefore, this phase should be efficiently carried out, and a

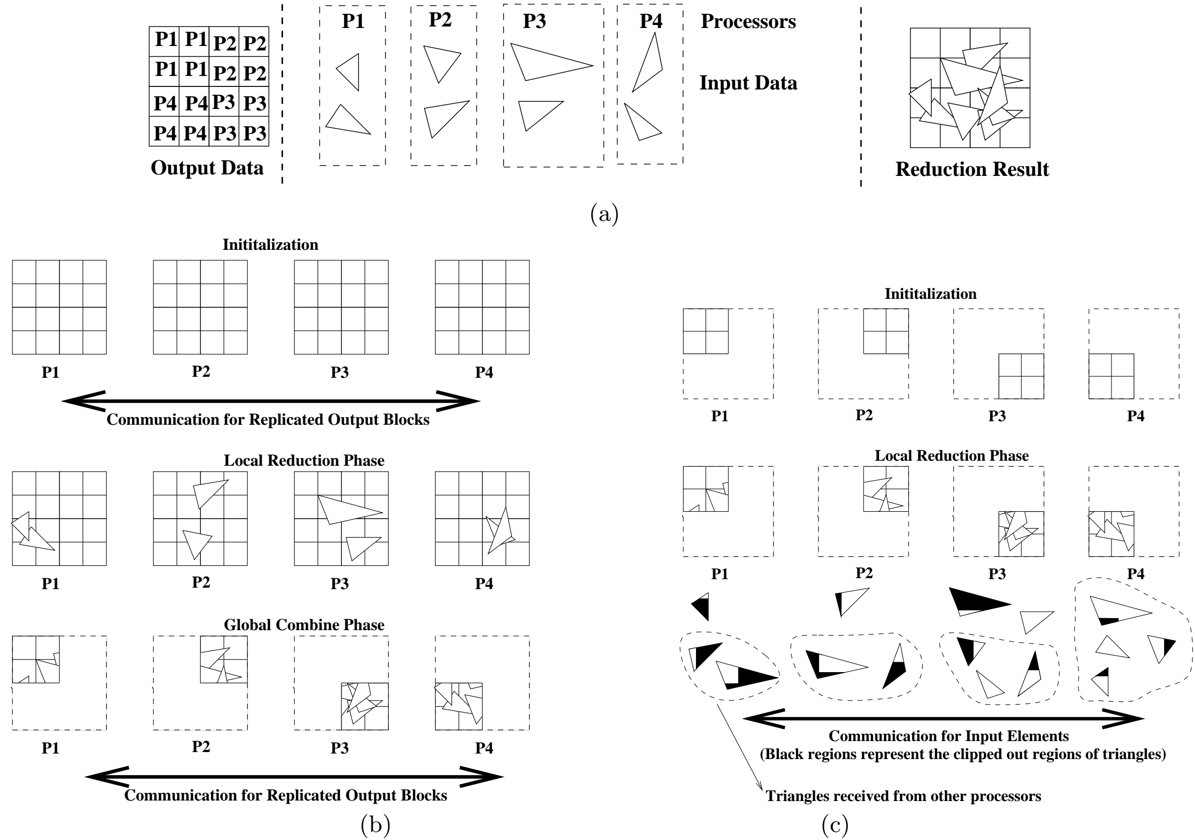

Figure 2: (a) Distribution of input and output data elements to processors, and the final result to be obtained after reduction. (b) Replicated Accumulator strategy. In this paper, this strategy is referred to as replicated filter state strategy. (c) Distributed Accumulator strategy. This strategy is referred to as partitioned filter state strategy in this paper.

single merge filter will likely become a bottleneck as the number of copies of filter **A** is increased. Figure 4(a) shows a hierarchy of merge filters organized into a tree, where each pair of **A** filters is connected to the merge filters at the leaves of the tree. In this configuration, accumulator data can be implicitly partitioned into smaller buffers in each **A** filter copy and buffers are sent through the tree, from the leaf nodes to the root node, thus allowing merge operations at different levels of the tree to be pipelined. This scheme is referred to as the *Pipeline Merge* strategy. An alternative scheme is to implicitly divide the accumulator among a set of merge filters. In the global combine phase, each copy of filter **A** sends portions of the accumulator to the corresponding merge filter. This scheme, shown in Figure 4(b), is called the *Distributed Merge* strategy. We have experimentally evaluated the performance of the two strategies for the global combine phase using both application emulators and datasets generated using Uniform and Gaussian distributions. Our results showed that the Distributed Merge strategy achieved slightly better performance than the Pipeline strategy because it makes better use of parallelism for the global combine operations. On a machine with P processors, $P-1$ processors are employed in the global combine phase with the Pipeline strategy, while the Distributed Merge uses all the processors. Hence, in this paper we have employed the Distributed Merge strategy. However, there are several cases in which the Pipeline strategy can be beneficial. For example, if there is a series of queries being processed, the set of processors can be partitioned among the merge and aggregation filters and the results from a previous query can be pipelined through the merge filters while the aggregation filters carry out reduction operations on the current query. We plan to address a detailed performance comparison of different global merge strategies in a future work.

4.2 Partitioned Filter State (PFS)

The replicated filter strategy has two main disadvantages. First, since accumulator elements are replicated across machines, the aggregate memory space is not most effectively used. The size of the accumulator structure is limited by the machine with the smallest main memory[1]. Second, merging of partial results may create a performance bottleneck– as the number of copies of aggregation filters increases, both the communication and computation cost of the global combine phase will rise. As an alternative strategy, we can partition the accumulator maintained by a filter when multiple copies of the filter are executed. This scheme eliminates the merge filter.

[1]This limitation can be alleviated to some degree by *tiling* the accumulator so that each tile can fit in memory. However, in that case, the overall processing should be executed in phases, each of which handles one accumulator tile, and input elements that map to multiple accumulator tiles have to be retrieved multiple times from (potentially distributed and remote) data sources.

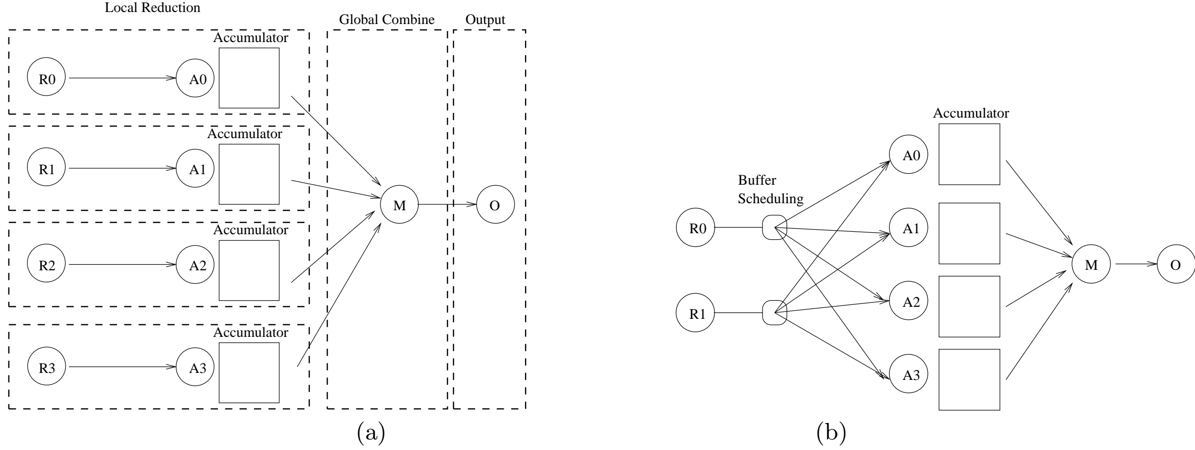

Figure 3: The replicated filter strategy. The application is composed of Read (R), Aggregation (A), and Output (O) filters. The merge filter (M) is added to ensure correct execution when multiple copies are executed. (a) each R filter sends data to only one A filter, (b) buffers from a R filter are dynamically scheduled among A filters using a demand-driven policy.

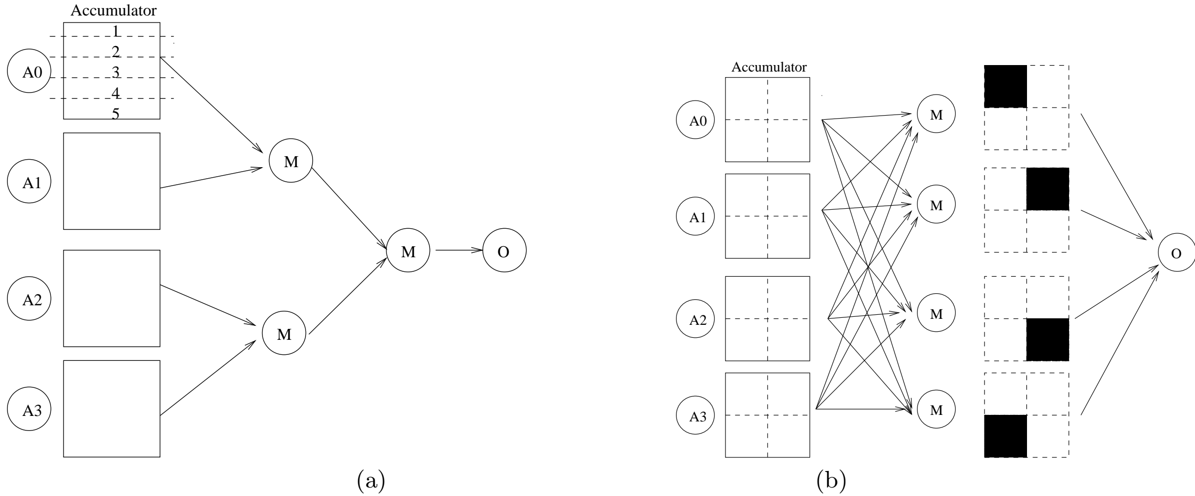

Figure 4: Alternative configurations for merge operations. (a) Merge (M) filters are organized into a tree, where merge operations at different levels can be pipelined. (b) Accumulator is implicitly partitioned among merge filters, and each A filter sends portions of the local accumulator to the corresponding M filters.

Figure 5 shows an example of the partitioned filter state strategy. In this example, the accumulator associated with the aggregation filter **A** is divided into four separate regions. Each copy of filter **A** is assigned one of the accumulator regions. Unlike the RFS strategy, a copy of the producer (reader) filter should explicitly send each buffer to the corresponding aggregation filter(s). Note that a mapping function may map an input element to multiple output elements. If the input elements of a buffer from filter **R** map to the accumulator elements assigned to different accumulator regions, the buffer needs to be sent to all the copies of filter **A** that are assigned the corresponding accumulator regions.

The main advantages of PFS are aggregate memory space is better utilized because the accumulator is partitioned, and a potential performance bottleneck due to merging of partial results is mitigated. However, this strategy incurs communication overhead because buffers from a producer filter are sent to one or more copies of an aggregation filter. Moreover, as the number of the copies of filter **A** increases, it is likely that a buffer will map to more copies, resulting in more communication overhead. Note that a subset of the copies of filter **A** can be colocated with one of the **R** filters, as in RFS. However, the other copies of filter **R** may still have to send some of the local buffers to the copies of filter **A**, colocated with a copy of **R**. In addition to communication overhead, the PFS strategy may result in computational load imbalance. If the mapping between input and output is skewed (i.e. input elements are irregularly distributed in the output space), some of the copies of filter **A** may process more data than other copies.

4.3 Hybrid Strategy (HS)

This strategy combines the RFS and PFS strategies so as to alleviate performance bottlenecks caused by each strategy. As for the partitioned filter state strategy, the

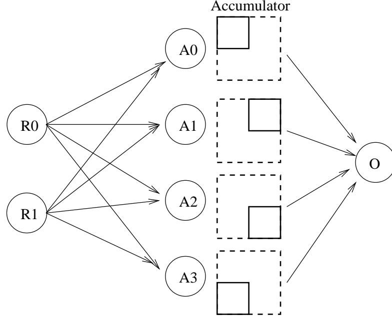

Figure 5: The partitioned filter state strategy. The accumulator structure is partitioned into four disjoint regions, and each region is assigned to one of the copies of filter A. Since the internal state is partitioned, a merger filter is not needed. The output filter O simply receives results from each copy of filter A and stitches them together to form the final output.

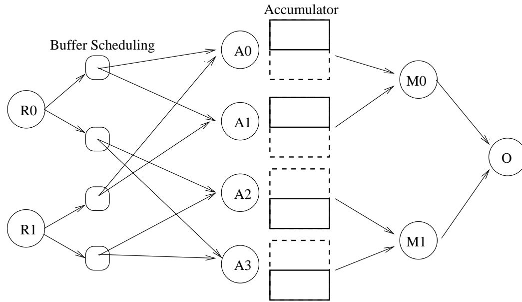

Figure 6: The hybrid strategy. The accumulator is partitioned horizontally into two regions denoted by solid rectangles. The first region (the top partition of the accumulator) is replicated on filter copies A0 and A1, while the second region (the bottom partition of the accumulator) is replicated on filter copies A2 and A3.

accumulator data structure is partitioned among a set of filters, and some of the subregions are replicated as for the replicated filter state strategy. In general, if P copies of an aggregation filter are created, where P is the number of processors, we can organize them into N sets, each of which consists of M transparent copies of the filter, where $N \times M$ is equal to P. The accumulator is first partitioned into N regions. A region is assigned to one of the sets, and replicated across all the copies in that set. Note that in this case a merge filter is required only among the copies of a filter that are in the same set. Figure 6 illustrates an example of the hybrid strategy. In this example, the accumulator is partitioned into two regions and each region is replicated across two copies of filter **A**. There are two merge filters because of the two sets of filter copies. We should note that filters **A0** and **A2** can be colocated with filter **R0**, and filters **A1** and **A3** can be colocated with filter **R1**. In that case, the processing can be viewed as consisting of a *reduction* phase and a *global combine* phase as in the RFS strategy.

When employing the hybrid strategy, we need to determine which subregions should be replicated, and how many transparent copies of the corresponding filters should be created.

In this paper, we develop a hybrid strategy with the goal of reducing the volume of communication among filters and to achieve good use of aggregate memory space in the environment. Such a strategy can be targeted for systems with relatively slow network bandwidth compared to CPU power and with small memory space per processor. Effective use of the aggregate memory space is particularly important when the accumulator is very large and each processor has relatively small memory. In that case, the full accumulator replicated on a processor may use virtual memory and result in paging to disk. We start with partitioning the accumulator into N pieces. Each piece contains M copies so that $N \times M$ is equal to P. Note that a single piece corresponds to the replicated filter state strategy, whereas with P pieces we obtain the partitioned filter strategy.

The choice of N depends on the query and the configuration of the machine. First, the accumulator for a given query should be partitioned such that each piece fits in the local memory of a processor. Second, if the amount of input data is much larger than that of the output data (or the size of the accumulator), the number of pieces should be minimized so that the volume of communication is reduced due to input data elements. On the other hand, if the size of the output is comparable to that of the input data, the accumulator can be partitioned into more pieces (thus approaching the partitioned accumulator strategy) in order to reduce the volume of communication due to the global combine phase. In our experiments we found that in regards to these issues, typically choosing $N = 2$ gave the best results for the configurations and applications used in the experiments. Once N is determined, we look at the volume of communication that will be generated by each reader filter to each of the pieces. These values can be computed by performing the mapping between input and output elements, without running the entire query. In most of the applications targeted in this work, input data elements are grouped into data chunks for better I/O performance. Each data chunk is associated with a bounding box in the underlying multi-dimensional space of the input dataset. The mapping of bounding boxes of input data chunks to the subregions of the accumulator can be used to determine the volume of communication. In our experiments, we observed that this phase of the algorithm takes negligible time. We order the read filters based on the total volume of communication generated by each filter. Starting from the most communication intensive filter, a filter is assigned a copy of the piece, for which it generates the most volume of communication. After all the read filters are assigned a copy of an accumulator piece, the remaining copies are distributed among the other nodes in the environment so that each node gets one copy.

Another hybrid strategy could be targeted at reducing the load imbalance among the filters that process the accumulator. This strategy could be beneficial especially when the distribution of input data over the output space is highly skewed, resulting in large computational imbalance, and complex load balancing algorithms (e.g., hypergraph based partitioning [10]) are too expensive to apply. Filter(s) that are most heavily loaded are obvious candidates for replication. We can consider two ways to detect which filters are more loaded than other filters. The first ap-

proach uses the history information. That is, during the execution of a unit-of-work, performance information can be collected for each aggregation filter. This information can be used to determine the set of filters to be replicated for the next unit-of-work. This approach can be beneficial if there is a correlation between successive units of work. For example, if a visualization of a three-dimensional volume is carried out over several time steps, the rendering of each time step can be considered a unit-of-work. In that case, it is likely that load distribution will be more or less the same between two consecutive renderings. The second approach uses the mapping between input and output elements. Although the second approach requires an extra step before execution of a unit-of-work, it does not require the instrumentation of filter codes and does not assume a correlation between units-of-work. We are planning to look at the relative performance benefits of the two approaches in future. Once the subset of filters that should be replicated has been determined, the next question is how many copies of filters should be instantiated. A possible strategy keeps the number of filters equal to the number of processors. Assume the accumulator is initially partitioned into P subregions, where P is the number of processors. The subregions assigned to the least loaded filters are combined into one subregion. The new subregion is assigned to one filter, and the subregion that is assigned to the most loaded filter is replicated so as to maintain the total number of filters at P. This process can be iteratively executed for the next most loaded filter until the load imbalance between filters is below a user-defined threshold.

Ideally, a hybrid strategy should target reducing both the volume of communication and the load imbalance. In this paper, we use a hybrid strategy that aims to reduce volume of communication and achieve good use of aggregate system memory. This strategy is beneficial for systems with slow network bandwidth and small memory space per processor. Large disk-based storage systems oftentimes trade network bandwidth and memory space for more disk storage space. An example is the storage cluster used in our experiments (see Section 5.1.3), which has 300GB disk space, but only 512MB main memory, per node. The nodes are connected via a Fast Ethernet Switch. On a system with large memory per node and high-end communication network, a hybrid strategy addressing the load imbalance problems could be more beneficial.

Analytical cost models can be useful for choosing between different strategies and determining the values of N and M for the hybrid strategy. In an earlier work, we examined models for partitioned and replicated accumulator strategies [9] when the distribution of input data elements over the output space is uniform. We were able to show that under uniform distribution assumptions a simple cost model could be used to choose between the two strategies on a given parallel machine. However, when data distributions are irregular and the target systems are heterogeneous, the analytical models can become quite complicated. We plan to investigate this issue in a future work.

5. EXPERIMENTAL RESULTS

We have carried out a detailed evaluation of the strategies presented in the paper. Due to limited space, we briefly describe the overall experimental design and present some of the experimental results. More detailed description of the experimental setup and additional experimental results can be found in the technical report version of this paper [28]. We have evaluated the execution strategies for different applications, varying the number of processors, their configuration, and the input dataset sizes.

5.1 Experimental Design

5.1.1 Case Study Applications

Our studies used application emulators, which are based on three data-intensive applications that involve reductions. An application emulator preserves the important *computational characteristics* of the application, but allows the input and output sizes to be scaled [34]. By computational characteristics, we mean the ratios between I/O, computation, and communication, and regularity or sparsity in the processing structure. The three applications we used were the following:

Satellite Data Processing: A typical analysis processes data acquired by satellite-based sensors for ten days to a year and generates one or more composite images of the area under study [12]. Generating a composite image requires projection of the globe onto a two dimensional grid; each pixel in the composite image is computed by selecting the "best" sensor value that maps to the associated grid point. This application will be denoted by `Sat` in our description.

Virtual Microscope: The Virtual Microscope [17] is an application to support the need 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 virtual microscope application emulates the usual behavior of a physical microscope, including continuously moving the stage and changing magnification. This application will be denoted by `VM` in our description.

Water contamination studies: Environmental scientists study the water quality of bays and estuaries using long running hydrodynamics and chemical transport simulations [27]. The chemical transport simulation models reactions and transport of contaminants, using the fluid velocity data generated by the hydrodynamics simulation. The chemical transport simulation is performed on a different spatial grid than the hydrodynamics simulation, and also often uses significantly coarser time steps. To facilitate coupling between these two simulation, there is a need for mapping the fluid velocity information from the hydrodynamics grid, averaged over multiple fine-grain time steps, to the chemical transport grid and computing smoothed fluid velocities for the points in the chemical transport grid. This application will be referred to as `WCS` in our description.

Although these three applications involve reduction computations on large datasets, they are quite different in other characteristics. In `Sat`, the distribution of individual data items (and of the input data chunks) is irregular. This is because of the polar orbit of the satellite [12], the data chunks near the poles are more elongated on the surface of the earth than those near the equator and there are more overlapping chunks near the poles. The other two applications are regular in this respect. The input datasets for those applications are regular dense arrays, which are

partitioned into equal-sized rectangular chunks. In VM, the execution time is dominated by I/O, as the amount of computation is small. In comparison, WCS involves significant amount of computation.

5.1.2 Datasets

In our experiments, we processed a number of queries for each of these applications. A query typically specifies a region in space (part of the image or grid), and a subset of the time domain covered by the dataset (time-steps or number of days). However, the queries for each application were processed over fixed-sized datasets. We chose dataset sizes that cannot be trivially downloaded over the Internet. The characteristics of the datasets are listed in Table 1.

In order to achieve parallelism when retrieving data from the storage system, input data chunks should be distributed across the disks in the system. In this work, the assignment of input chunks to the disks was done using a Hilbert curve based declustering algorithm [15]. Hilbert curve algorithms have been shown to achieve good I/O parallelism for multi-dimensional datasets.

5.1.3 Hardware Configuration

For efficient and distributed processing of datasets available in a remote data repository, we need high bandwidth networks and a certain level of quality of service support. Recent trends are clearly pointing in this direction, for example, the five sites that are part of the NSF funded Teragrid project expect to be connected with a 40 Gb/second network [33]. However, for our study, we did not have access to a wide-area network that gave high bandwidth and allowed repeatable experiments. Therefore, all our experiments were conducted within a single cluster. The cluster, referred to here as *storage cluster*, used for our experiment has 16 nodes, each with one Pentium III 933MHz CPU, 512 MB memory, and three 100 GB hard disks. The nodes are connected to each other through a 100 Mb Fast Ethernet switch.

We assumed that input data was available on a subset of the nodes, and other nodes in the cluster were available for processing of this data. For the experiments on scalability analysis, we had all data distributed on 1, 2, 4, 8, and 16 nodes of the cluster, and used the same set of nodes for all processing. For heterogeneous execution environment experiments, the data was partitioned among X nodes, and Y nodes were used as extra processing nodes. We used four different configurations, denoted by X-Y in our experiments. For example, in the 4-12 configuration, data is distributed on 4 nodes, and additional 12 nodes are used for processing.

5.2 Performance Results

In this section we present an experimental evaluation of the three strategies. We had three goals in our experiments. We wanted to evaluate

- the scalability of our techniques in a cluster environment.
- the relative performance of our techniques in a distributed environment, where data is hosted on one set of machines, and another cluster is used for processing the data.
- the performance of the three strategies as the volume of input data and the size of accumulator are scaled proportionately.

Our first set of experiments examines the scalability and relative performance of the strategies in a cluster environment. The second set of experiments extends this investigation to a distributed environment, in which data is hosted on a set of nodes and processed on another set of nodes. These set of experiments show that the relative performance of the PFS and RFS strategies varies as the number of machines as well as the relative size of input data and accumulator structure are changed. We observe that being more flexible, the hybrid strategy can follow the performance trend of the better of PFS and RFS and also perform better than both strategies most of the time. The first two sets of experiments indicate that the sizes of input and output and the distribution of data have an impact on the relative performance of the strategies. In order to further investigate the effect of the size of input and output data structures on performance, we carry out a third set of experiments. These experiments illustrate that the hybrid strategy is a more viable approach, performing better than both RFS and PFS.

5.2.1 Scalability Studies on a Homogeneous Cluster

The goal of this set of experiments is to evaluate the scalability of the three strategies in a cluster environment. For each of the three applications, we considered several query and output (or accumulator) sizes. To focus on the more interesting cases, we are only including results from two of the cases for each of the applications. These cases are 1) small accumulator and large query, and 2) large accumulator and small query. Results from the other two cases are available in a longer version of this paper [28]. The characteristics of the queries used in the experiments are shown in Table 2. In the table, Query Window denotes the multi-dimensional bounding box of the query.

The results from these two cases of execution of Sat are presented in Figures 7 and 8. With large query and small accumulator (Figure 7), the RFS scheme consistently performs the best. This is because the time required for performing the local reductions dominates the overall execution time. PFS does not scale well. This is because of two reasons, high volume of communication between the nodes reading the data and the nodes performing the reduction, and load imbalance among the nodes. The hybrid scheme is able to reduce some of these overheads, and particularly the hybrid scheme comes very close to RFS on 16 nodes. The results change significantly with small query and large accumulator (Figure 8). In this case, the RFS scheme does not scale well. This is because the time required for merging the accumulators dominates the reduction time. The performance of PFS and the hybrid scheme is quite close to each other. None of the schemes, however, give very high speedups, as the total amount of work in processing the query is quite small.

The results from the two cases of execution of VM are presented in Figures 9 and 10. The results are quite similar to what was observed for Sat. With large query and small accumulator, RFS scales well and outperforms other schemes. However, with small query, it does not scale well. In these cases, the hybrid scheme gives the best performance. The results from the two cases of execution of WCS are presented in Figures 11 and 12. The trends are again quite similar. However, because this application is regu-

Application	Dataset Size	Explanation
Sat	51GB	300 days of global coverage data (a $46,080 \times 20,480$ region).
VM	53GB	64 slides, each covering a $46,080 \times 46,080$ pixel region.
WCS	50GB	64 time-steps and a $46,080 \times 46,080$ grid.

Table 1: The characteristics of the datasets for different case study applications.

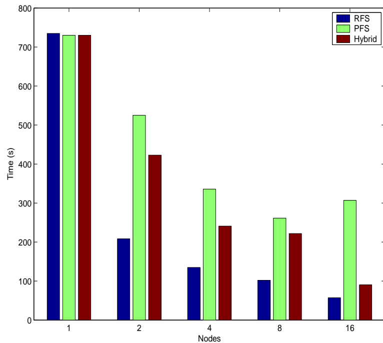

Figure 7: `Sat` with large query and small accumulator (Scalability)

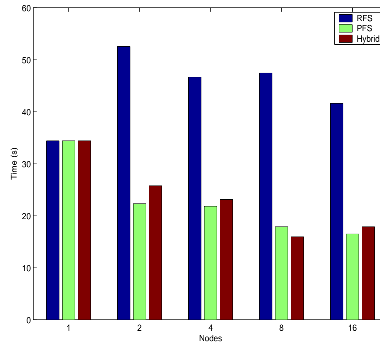

Figure 8: `Sat` with small query and large accumulator (Scalability)

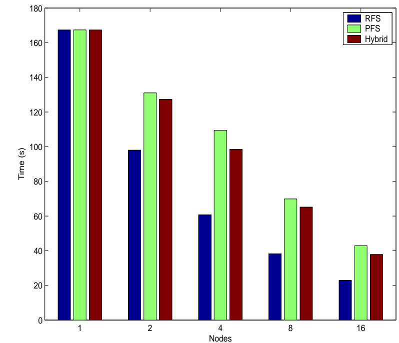

Figure 9: `VM` with large query and small accumulator (Scalability)

lar and involves a significant amount of computation, the relative performance of the RFS is even better. The hybrid scheme is able to improve upon the PFS scheme, by reducing load imbalance and communication.

Overall, our experiments on studying the scalability of the techniques show the following. First, the RFS scheme outperforms other schemes if sufficient memory is available, and if the amount of processing offsets the merge costs. In other cases, the hybrid scheme either gives the best performance or is close to the best scheme. This shows that if it is not possible to predict the best strategy for a given query and hardware configuration, using the hybrid strategy is generally a good option.

5.2.2 Execution in a Distributed Environment

Our second set of experiments evaluated the different processing strategies in a distributed environment, where a set of nodes host the data, and another set of nodes are available for processing of this data. The x-axis in Figures 13 – 18 is the different X–Y configurations employed in the experiments, where X processors are used to host the data and Y additional nodes are used for processing.

We used the same three applications, each with the four different combinations of input and output sizes. We used the same query and accumulator sizes we had used for our previous set of experiments, with the exception of `Sat`. The two queries used here corresponded to 3.3 GB and 114 MB of input data. For the RFS strategy, when additional nodes are available for processing, we used the *demand-driven* scheme, i.e., the chunks read on one node are forwarded to different nodes based upon the rate they are able to process them. Again, because of limited space, we will only present results from two of the four cases, i.e., large query and small accumulator and small query and large accumulator.

The results for the two cases for `Sat` are presented in Figures 13 and 14. Similar to the previous set of experiments, the RFS scheme outperforms the others with the large query. Here, the PFS scheme performs very poorly, but the hybrid scheme is quite close to RFS. With small query and large accumulator, RFS performs poorly, and the performance of PFS and hybrid is quite similar. The results for the two cases for `VM` are presented in Figures 15 and 16. The RFS scheme outperforms others when the query is large and the accumulator is small. In the other case, the relative performance of PFS and hybrid also varies. PFS usually performs the best. However, the hybrid scheme is close to the best scheme in all cases. The results for the two cases for `WCS` are presented in Figures 17 and 18. Again, RFS is a clear winner when the query is large and the accumulator is small. In the other case, the hybrid scheme is either quite close to the best scheme.

5.2.3 Effect of Size of Input Data and Accumulator

As we discussed in Section 4 and as is seen in the first two sets of experiments, the amount and distribution of input data as well as the size of the accumulator can have significant impact on the performance of an application. The performance of the PFS strategy may suffer from high communication overhead and load imbalance. RFS can reduce the overhead that is due to communication of input data elements, but its performance may suffer because of the overheads in maintaining and combining the replicated accumulator. We can expect that the ratio between the amount of data read and the size of output can affect the relative performance of various schemes. If this ratio is high, RFS can be expected to perform better. However, RFS does not scale well even if this ratio is kept constant. To demonstrate this, we conducted an experiment, in which the accumulator size was increased, but the ratio between the query size and the accumulator size was kept constant. Corresponding to accumulator sizes of 100 MB, 300 MB, and 600 MB, the size of input data was 1 GB, 3 GB, and 6 GB, respectively. We used `Sat` with a `4-12` configuration, where data is stored on 4 nodes and 12 additional nodes are used as compute nodes.

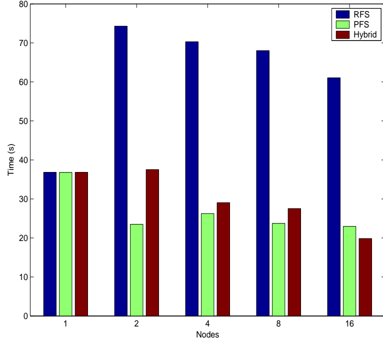

Figure 10: `VM` with small query and large accumulator (Scalability)

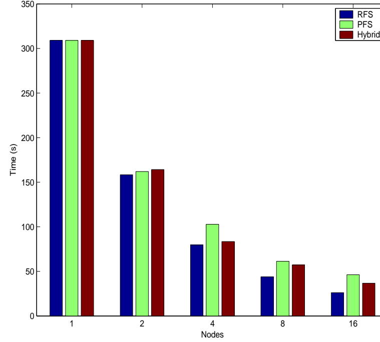

Figure 11: `WCS` with large query and small accumulator (Scalability)

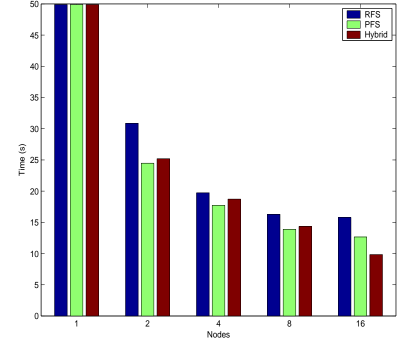

Figure 12: `WCS` with small query and large accumulator (Scalability)

Large Query/Small Accumulator			
	`Sat`	`VM`	`WCS`
Query Window	[(0,0)x(8191,4095)] subregion over 30 days.	[(0,0)x(8191,8191)] subimage from 30 slides.	[(0,0)x(8191,8191)] subarray from 50 time steps.
Input Read	1.9GB	1.9GB	1.9GB
Output (Accumulator) Size	25MB	50MB	12MB
Small Query/Large Accumulator			
	`Sat`	`VM`	`WCS`
Query Window	[(0,0)x(8191,4095)] subregion over 1 day.	[(0,0)x(8191,8191)] subimage from 3 slides.	[(0,0)x(8191,8191)] subarray from 6 time steps.
Input Read	82MB	190MB	227MB
Output (Accumulator) Size	100MB	200MB	48MB

Table 2: The characteristics of queries used in the experiments.

The results are presented in Figure 19. Recall that the available main memory on each machine is 512 MB. When the accumulator size is 100 MB, RFS clearly outperforms PFS, but hybrid is slightly better. As the accumulator size is increased from 100 MB to 300 MB, the time taken by each scheme increases consistently by a factor of 3. The relative performance between the three strategies is the same. However, as the accumulator size is increased to 600 MB, the overhead of replicating the accumulator becomes significant. Because of the effect of virtual memory, RFS is now the slowest. PFS now has better performance than RFS. We also observe that if the accumulator size is increased to 1.2GB, RFS will not run as the processors do not have sufficient main and virtual memory space to hold the accumulator. Thus, compared to RFS, PFS is a more viable strategy for configurations with large accumulator size and relatively small memory space per processor, as it uses the aggregate memory space more effectively. However, our results show that the hybrid strategy is the best among these schemes. It can make good use of available aggregate memory by partitioning the accumulator, while reducing communication overheads and achieving better load balance by replicating some of the accumulator regions.

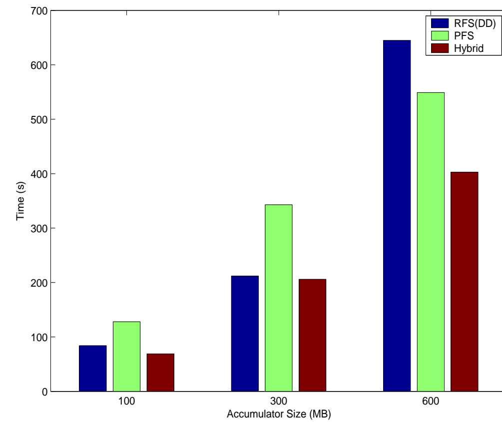

Figure 19: Effect of Scaling Accumulator Size: `Sat` in a `4-12` configuration. In this configuration, data is distributed on 4 nodes, and 12 additional nodes are used for processing.

6. RELATED WORK

Parallelization of reductions has been widely studied. Here, we list only a subset of such efforts. Several researchers have focused on improving the performance of irregular applications through compiler analysis and/or runtime support [14, 23, 35]. Shatdal and Naughton [31] examine two algorithms for relational group-by queries on

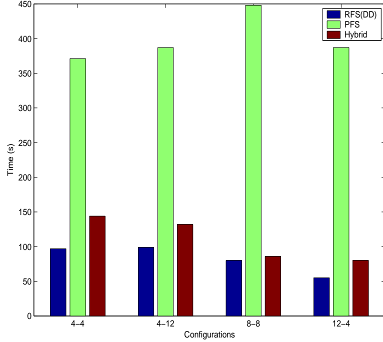

Figure 13: `Sat` with large query and small accumulator (Distributed Environment)

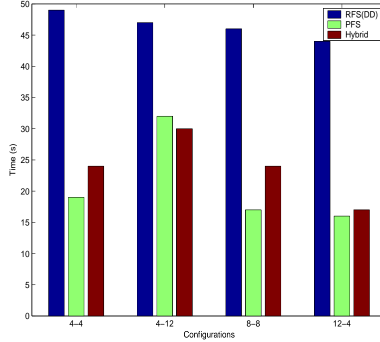

Figure 14: `Sat` with small query and large accumulator (Distributed Environment)

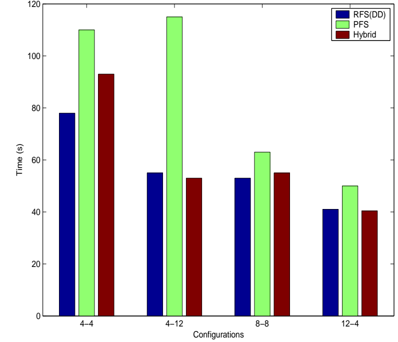

Figure 15: `VM` with large query and small accumulator (Distributed Environment)

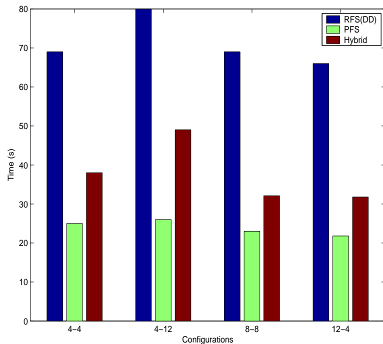

Figure 16: `VM` with small query and large accumulator (Distributed Environment)

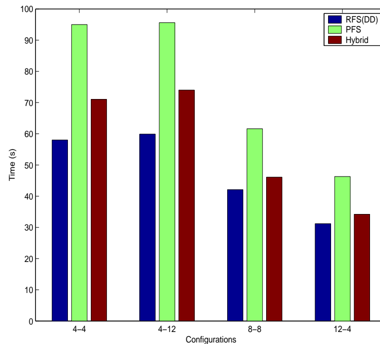

Figure 17: `WCS` with large query and small accumulator (Distributed Environment)

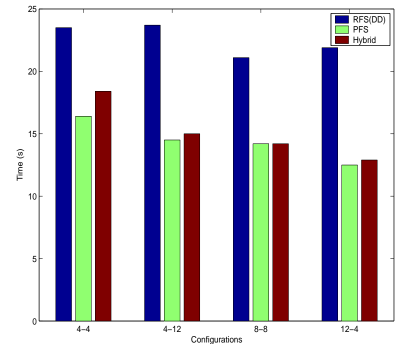

Figure 18: `WCS` with small query and large accumulator (Distributed Environment)

distributed-memory systems. Bitton *et al.* [5] describe two sorting based algorithms, similar to the two algorithms developed by Shatdal and Naughton, for a shared disk cache architecture. Chang et. al. [11, 26] investigate strategies for performing aggregation operations on multi-dimensional datasets on distributed-memory systems with a local disk farms.

As compared to the above (and other similar efforts), our work is distinct in several ways. Our focus is on data-intensive applications, and in processing of data accessed from remote data repositories. Inspector/executor, which has been the key approach for optimizing irregular reductions, is generally not applicable in scenarios we are targeting. This is because only a limited information may be available about datasets in advance, and therefore, sophisticated partitioning or runtime preprocessing cannot be applied. We are also not aware of any previous work in which the distributed environment of the type we have considered was used for execution of reduction computations.

A number of research projects have focused on providing services for resource discovery, authentication and authorization, resource allocation, and secure, efficient and reliable access to resources [3, 6, 18, 19, 22, 32]. None of these applications have, however, examined execution of reduction computations over large datasets.

Several other projects have certain similarities with DataCutter. Many projects have used component-based frameworks as approaches for application development in distributed, heterogeneous environments [8, 21, 30, 29]. The techniques described and evaluated in this paper can also be implemented in the context of these systems.

7. CONCLUSIONS

In this paper, we have investigated and evaluated techniques and runtime support for processing of generalized reductions over datasets stored in repositories. The three techniques we have examined are, replicated filter state (RFS), partitioned filter state (PFS), and a hybrid scheme combining the two.

The key observations from our experiments are as follows. The relative performance of different techniques depends upon the application, query and output sizes, and the configuration. However, certain general patterns can be seen from the results. Replicating the filter state or accumulator scales well and outperforms other schemes if 1) sufficient memory is available to replicate the accumulator, and 2) sufficient computation is involved to offset the cost of merging the replicated accumulators. In other cases, the hybrid strategy is usually the best. Moreover, in almost all cases, the performance of hybrid strategy is quite close to the best strategy.

Based upon these observations, we can make certain inferences about the design of software for our target applications. First, it is clear that a number of different schemes must be supported as part of the system, to allow efficient processing for different applications, queries, and configurations. Further, based upon the trends listed above, we

believe that it is possible to automatically choose between different schemes at runtime. Because the performance of the hybrid scheme is always competitive, it is an attractive approach when the relative performance of different schemes cannot be predicted.

8. 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] G. Agrawal, R. Jin, and X. Li. Middleware and compiler support for scalable data mining. In *Proceedings of Languages and Compilers for Parallel Computing (LCPC)*, 2001.

[3] B. Allcock, I. Foster, V. Nefedova, A. Chervenak, E. Deelman, C. Kesselman, J. Lee, A. Sim, A. Shoshani, B. Drach, and D. Williams. High-performance remote access to climate simulation data: A challenge problem for data grid technologies. In *Proceedings of the 2001 ACM/IEEE SC01 Conference*. ACM Press, Nov. 2001.

[4] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, Oct. 2001.

[5] D. Bitton, H. Boral, D. J. DeWitt, and W. K. Wilkinson. Parallel algorithms for the execution of relational database operations. *ACM Transactions on Database Systems*, 8(3):324–353, 1983.

[6] R. Butler, V. Welch, D. Engert, I. Foster, S. Tuecke, J. Volmer, and C. Kesselman. A national-scale authentication infrastructure. *IEEE Computer*, 33(12):60–66, Dec. 2000.

[7] U. Catalyurek, M. D. Beynon, C. Chang, T. Kurc, A. Sussman, and J. Saltz. The virtual microscope. *IEEE Transactions on Information Technology in Biomedicine*, 2003. To appear.

[8] Common Component Architecture Forum. *http://www.cca-forum.org*.

[9] C. Chang. Cost models for query processing strategies in the active data repository. Technical Report CS-TR-4060 and UMIACS-TR-99-54, University of Maryland, Department of Computer Science and UMIACS, Sept. 1999.

[10] C. Chang, T. Kurc, A. Sussman, U. Catalyurek, and J. Saltz. A hypergraph-based workload partitioning strategy for parallel data aggregation. In *Proceedings of the Eleventh SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Mar. 2001.

[11] C. Chang, T. Kurc, A. Sussman, and J. Saltz. Optimizing retrieval and processing of multi-dimensional scientific datasets. In *Proceedings of the Third Merged IPPS/SPDP (14th International Parallel Processing Symposium & 11th Symposium on Parallel and Distributed Processing)*. IEEE Computer Society Press, May 2000.

[12] 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.

[13] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications: Special Issue on Network-Based Storage Services*, 23(3):187–200, July 2000.

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

[15] 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.

[16] R. Ferreira, T. Kurc, M. Beynon, C. Chang, A. Sussman, and J. Saltz. Object-relational queries into multi-dimensional databases with the Active Data Repository. *Parallel Processing Letters*, 9(2):173–195, 1999.

[17] 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.

[18] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[19] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufman, 1999.

[20] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications and High Performance Computing*, 15(3), 2001.

[21] Global Grid Forum. *http://www.gridforum.org*.

[22] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Wide-area computing: Resource sharing on a large scale. *IEEE Computer*, 32(5):29–37, May 1999.

[23] H. Han and C.-W. Tseng. Improving compiler and run-time support for irregular reductions. In *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, Aug. 1998.

[24] R. Jin and G. Agrawal. A middleware for developing parallel data mining implementations. In *Proceedings of the first SIAM conference on Data Mining*, Apr. 2001.

[25] T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Visualization of large datasets with the Active Data Repository. *IEEE Computer Graphics and Applications*, 21(4):24–33, July/August 2001.

[26] T. Kurc, C. Chang, R. Ferreira, A. Sussman, and J. Saltz. Querying very large multi-dimensional datasets in ADR. In *Proceedings of the 1999 ACM/IEEE SC99 Conference*. ACM Press, Nov.

1999.
[27] 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.
[28] F. Lee, T. Kurc, G. Agrawal, U. Catalyurek, R. Ferreira, and J. Saltz. Optimizing reduction computations in a distributed environment. Technical Report OSU-CISRC-4/02-TR20, Department of Computer and Information Sciences, The Ohio State University, April 2003.
[29] R. Oldfield and D. Kotz. Armada: A parallel file system for computational. In *Proceedings of CCGrid2001: IEEE International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, May 2001. IEEE Computer Society Press.
[30] B. Plale and K. Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *IEEE International High Performance Distributed Computing (HPDC)*, August 2000.
[31] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD95)*, pages 104–114, San Jose, CA, May 1995.
[32] SRB: The Storage Resource Broker. *http://www.npaci.edu/DICE/SRB/index.html*.
[33] The TeraGrid: A Primer, September 2002. Available at www.teragrid.org.
[34] 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.
[35] H. Yu and L. Rauchwerger. Adaptive reduction parallelization techniques. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 66–75. ACM Press, May 2000.