Multiple Query Optimization for Data Analysis Applications on Clusters of SMPs [*]

Henrique Andrade[†], Tahsin Kurc[‡], Alan Sussman[†], Joel Saltz[†‡]

[†] Dept. of Computer Science
University of Maryland
College Park, MD 20742
{hcma,als}@cs.umd.edu

[‡] Dept. of Biomedical Informatics
The Ohio State University
Columbus, OH, 43210
{kurc-1,saltz-1}@medctr.osu.edu

Abstract

This paper is concerned with the efficient execution of multiple query workloads on a cluster of SMPs. We target applications that access and manipulate large scientific datasets. Queries in these applications involve user-defined processing operations on data and distributed data structures to hold intermediate and final results. Our goal is to implement system components to leverage previously computed query results and to effectively utilize processing power and aggregated I/O bandwidth on SMP nodes so that both single queries and multi-query batches can be efficiently executed.

1 Introduction

The availability of low-cost storage systems, built from a cluster of PCs with a disk farm, is making it possible for institutions to create data repositories and make them available for collaborative use. In a collaborative setting, a data server may need to answer queries simultaneously submitted by multiple clients. Thus, efficient handling of multiple query workloads is an important optimization in many application domains [2, 9, 13].

The query optimization and scheduling problem has been extensively investigated in past surveys [7]. Traditionally, multiple query optimization techniques for relational databases rely on caching common subexpressions [11]. Cache space is limited by nature, and it is very well possible that the space available will not be enough to store all the common subexpressions detected. Carefully scheduling queries also plays an important role, because one could craft the query execution order in a way to better exploit expressions that have been already cached. Gupta et al. [8] present an approach that tackles this problem in the context of decision support queries.

In this work we look at methods for efficient execution of multiple queries with user-defined functions and data structures. This class of queries arises in data analysis applications that make use of large scientific datasets. Data analysis applications often access a subset of all the data available in a dataset. The data of interest is then processed to transform it into a new data product. This data product is usually generated by computing an aggregation over some of the dimensions of the dataset, ranging from simple associative arithmetic and logical operators to more complicated user-defined functions such as scientific visualization operations. Oftentimes, a user-defined data structure is created to maintain intermediate results during processing. When data analysis is employed in a collaborative environment, queries from multiple clients are likely to have overlapping regions of interest and similar processing requirements (i.e. the same operations on data). Hence, several optimizations can be applied to improve system response time. These optimizations include reuse of intermediate and final results, data prefetching and caching, and scheduling to improve inter-query locality [2, 3].

This paper investigates the use of SMP clusters to improve response times and overall system performance. In particular, we look at the effective use of aggregate processing power and I/O bandwidth for executing single and multiple queries efficiently. Unlike previous work on query execution in parallel systems [5, 6, 10, 12], our system design combines parallel execution of queries with data caching and multi-threaded execution so that multiple queries can execute concurrently on multiple processors on an SMP

[*]This research was supported by the National Science Foundation under Grants #EIA-0121161, #ACI-9619020 (UC Subcontract #10152408) and #ACI-9982087, Lawrence Livermore National Laboratory under Grant #B500288 (UC Subcontract #10184497), and the Department of Defense, Advanced Research Projects Agency, USAF, AFMC through Science Applications International Corporation under Grant #F30602-00-C-0009 (SAIC Subcontract #4400025559).

node and also reuse cached results to improve performance and lower interprocess communication. Moreover, a query is executed in parallel to exploit processing power and memory space distributed across the SMP nodes in the system, as well as the available aggregate I/O bandwidth. We also investigate different strategies for accumulating the final query result, as well as strategies to perform the final *stitching* of the partial results computed by each of the processors. Finally, we describe experimental results on a cluster of 2-processor SMP nodes using an image visualization application, exploring multiple system configurations using a fixed amount of caching memory.

2 Runtime System Architecture

We have deployed the runtime system within a middleware framework we have developed for evaluating multiple, simultaneous queries on a shared-memory system [2, 3]. In the current implementation, each SMP node essentially runs a copy of the middleware with extensions to handle data exchange among SMP nodes. We briefly describe the middleware in this section. The extensions we have implemented for execution on a cluster of SMPs are presented in Section 3.

The middleware architecture consists of several service components, implemented as a C++ class library and a runtime system, which support multithreaded execution on a cluster of shared-memory multiprocessor machines. Each processor hosts a complete instance of the system, with all the service components available. In the current implementation, there is conceptually only one copy of each of the system components in the parallel machine, however, concretely, these components exist in all the nodes but they work on partitioned data, which is a function of the location of the required input data, i.e., an instance handles only data blobs computed from the input data residing on its local disks.

Query Server: The query server interacts with clients for receiving queries and returning query results, and is implemented as a fixed-size thread pool (typically the number of threads is set to the number of processors available on a SMP node). A client request contains a query type id and user-defined parameters to the query object implemented in the system. The user-defined parameters include a *dataset id* for the input dataset, *query meta-information*[1], and an *index id* for the index to be used for finding the data items that are requested by the query.

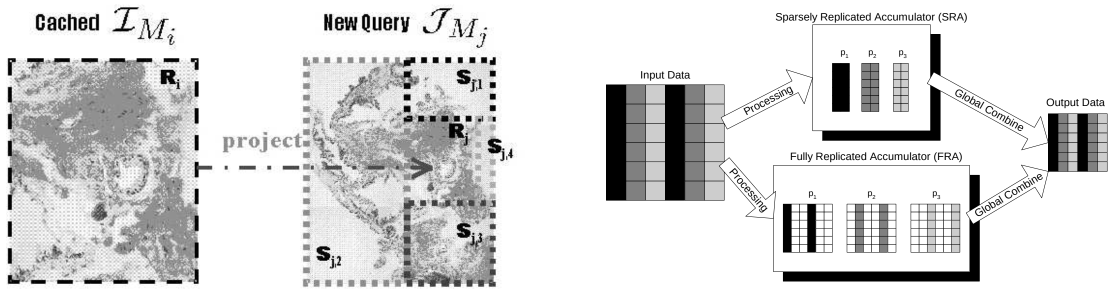
An application developer can implement one or more query objects that are responsible for application-specific subsetting and processing of datasets. When a query object is integrated into the system, it is assigned a unique *query type id*. The implementation of a new query object is done through C++ class inheritance and the implementation of virtual methods. A *Query* base class is provided for this purpose. A query object is associated with (1) an `execute` method, (2) a query meta-information object `qmi`, which stores query information, and (3) an accumulator object `qbuf`, which encapsulates user-defined data structures for storing intermediate results. The `execute` method implements the user-defined processing of data. In the current design, this method is expected to carry out index lookup operations, the initialization of intermediate data structures, and the processing of data retrieved from the dataset. Both the query meta-data object and the accumulator meta-data object are implemented by the application developer by deriving a subclass from a *QueryMI* base class provided by the system.

When a query is received, the query server instantiates the corresponding query object and spawns a *Query Thread* to execute the query. The query thread searches for cached results that can be reused to either completely or partially answer a query. The lookup operation employs a user-defined `overlap` operator to test for potential matches. The user-defined accumulator meta-data object associated with the query object is compared with the accumulator meta-data objects of the cached results for the same query type. A user-defined `project` method is then called so that the cached result can be *projected*, potentially performing a transformation on the cached data, to generate a portion of the output for the current query (see Figure 1(a)). Finally, if the current query is only partially answered by the cached results, sub-queries are created to compute the results for the portions of the query that have not been computed from cached results. The sub-queries are processed just like any other query in the system, thereby allowing more intermediate results to be reused.

Both the `overlap` and `project` method interfaces are defined in the *QueryMI* class, and have to be implemented by the application developer. These two methods allow for the identification and implementation of reuse possibilities by the runtime system for user-defined data structures, when a new query is being executed.

Data Store Manager: The data store manager (DS) is responsible for providing dynamic storage space for data structures generated as intermediate or final results for a query. The most important feature of the data store is that it records semantic information about intermediate data structures. This makes the use of intermediate results possible to answer queries later submitted to the system. A query thread interacts with the data store using a *DataStore* object, which provides functions similar to the C language function *malloc*. When a query wants to allocate space in the data

[1]The query meta-information describes which part of the dataset is relevant to satisfy a query, and is domain dependent, e.g., it can be an 3-dimensional bounding box in a visualization application or a boolean expression in relational database queries.


Cached $\mathcal{I}_{M_i}$
New Query $\mathcal{J}_{M_j}$
R_i
project
R_j
$S_{j,1}$
$S_{j,2}$
$S_{j,3}$
$S_{j,4}$
Sparsely Replicated Accumulator (SRA)
Input Data
Processing
Global Combine
Output Data
Fully Replicated Accumulator (FRA)
p_1
p_2
p_3

(a) (b)

Figure 1. (a) The query execution mechanism. Once a new query q_j with meta-information M_j is submitted, the system tries to find a complete or partial cached match that can be used to compute q_j. Once it is found (region R_i, in our example), a data transformation is applied with the user-defined `project` method to compute region R_j. Sub-queries – $S_{j,1}$, $S_{j,2}$, $S_{j,3}$, and $S_{j,4}$ – are generated to complete the query processing and produce the answer $\mathcal{J}$. (b) Strategies for allocating the accumulator. The upper path represents how a query gets answered with the Sparsely Replicated Accumulator (SRA) strategy in which each processor allocates only the relevant part of the accumulator object. The lower path shows the Fully Replicated Accumulator (FRA) strategy.

store for an intermediate data structure, the size (in bytes) of the data structure and the corresponding accumulator meta-data object are passed as parameters to the *malloc* method of the data store object. DS allocates the buffer space, internally records the pointer to the buffer space and the associated meta-data object containing a semantic description, and returns the allocated buffer to the caller.

DS also provides a method called `lookup`. This method can be used by the query server to check if a query can be answered entirely or partially using the intermediate results stored in the data store. The `lookup` method calls the `overlap` method for accumulator meta-data objects in the data store, and returns a reference to the object that has the largest overlap with the query. A hash table is used to access accumulator meta-data objects in DS.

Data Sources: A data source can be any entity used for storing datasets. In the current implementation, a dataset is assumed to have been partitioned into fixed-size pages and stored in a data source. That is, the data source abstraction presents a page-based storage medium to the runtime system, whereas the actual storage can be, for example, a file stored on a local disk or a remote database accessed over a wide-area network. When data is retrieved in pages instead of as individual data items, I/O overheads (e.g., disk seek time) can be reduced, resulting in higher application level I/O bandwidth. Using fixed-size pages also allows more efficient management of available memory space. A base class, called *DataSource*, is provided by the runtime system so that an application developer can implement support for multiple physical devices and data storage abstractions. The base class has virtual methods, with semantics similar to Unix file system operations (i.e., open, read, write, and close), that are called by the runtime system. We have implemented two data source subclasses, one for the Unix file system and a second to overcome the 2GB file size limitation in the Linux ext2 file system.

Page Space Manager: The page space manager (PS) controls the allocation and management of buffer space available for input data in terms of fixed-size pages. All interactions with data sources are done through PS. Queries access PS through a *Scan* object, which is instantiated with a data source object and a list of pages (which can be generated as a result of index lookup operations) to be retrieved from the data source. The pages retrieved from a data source are cached in memory. The current implementation uses a hash table for searching pages in the memory cache. PS also keeps track of I/O requests received from multiple queries so that overlapping I/O requests are reordered and merged, and duplicate requests are eliminated. For example, if the system receives a query into a dataset that is already being scanned for another query, the traversal of the dataset for the second query can be *piggybacked* onto the first query in order to avoid traversing the same dataset twice.

Index Manager: The index manager provides indexing support for the datasets. A query thread interacts with the index manager to access indexing data structures and search for data that intersect with the query. The integration of new indexing mechanisms is achieved by derivation from base classes defined in the core middleware.

3 Execution on a Cluster of SMPs

We now discuss how queries are executed on a cluster of SMPs, and the extensions we have incorporated into the shared-memory server for execution in a distributed-memory environment. The parallel implementation uses MPI, employed to behave correctly in a multi-threaded environment[2].

Dataset Organization: We assume that each SMP node in the system has one or more local disks. In such a system, efficient access to, and processing of data depends on how datasets are declustered across disks and processors, since workload distribution and communication costs depend on where data elements are stored. Therefore, the fixed-size data pages of a dataset are distributed across the disks in the system. If data subsets are defined by range queries, data pages that are close to each other in the underlying attribute space should be assigned to different disks.

Query Execution: Queries execute as threads, as in the original runtime system. This configuration allows multiple queries to execute concurrently on a SMP node. We consider several approaches for evaluating multiple queries when a cluster of SMP nodes is employed. One possible approach is to execute each query sequentially on a separate node in the system. The advantage of this approach is that if there are n nodes with k processors, $n \times k$ queries can be executed in the system simultaneously. However, this approach is likely to incur high interprocessor communication volume. Since datasets are declustered across the nodes in the system, if a query is executed on a single processor other processors in the system must retrieve data pages required by the query and forward them to that processor. Moreover, if fewer queries than the number of processors available are submitted to the system, some of the processors will be idle causing under-utilization of the aggregate processing power. In order to alleviate these problems, each query is executed in parallel.

We have implemented two strategies based on the *replicated accumulator* scheme developed in [10]. In the *Fully Replicated Accumulator* (FRA) scheme, a query is assigned to all the SMP nodes in the system for evaluation. The entire accumulator structure associated with the query is allocated on all the nodes. Each SMP node is responsible for retrieving and carrying out the aggregation operations on its local input data. In the *Sparsely Replicated Accumulator* (SRA) scheme, a query is also assigned to all SMP nodes in the system for evaluation. However, for this scheme each SMP node only allocates memory for the portions of the accumulator for which it has local input data and/or cached results. This scheme can effectively result in a partitioning of the accumulator data structure across the nodes. Both schemes are shown schematically in Figure 1(b).

[2]We have used the MPICH implementation of MPI, which is not thread-safe.

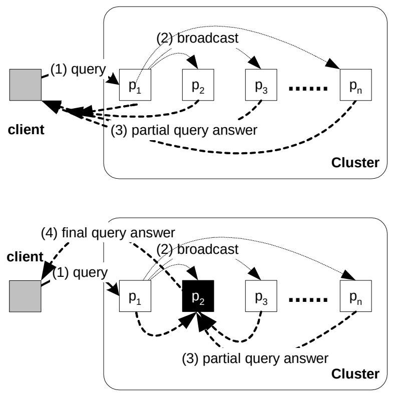
Other strategies are also possible. Our earlier work [6, 10] and the work of Shatdal and Naughton [12] have shown that other strategies, such as *distributed accumulator*, may outperform the replicated accumulator strategies, depending on machine configuration (e.g., number of nodes) and application characteristics. The previous work evaluated various strategies, but only when one query was executed in the system at a time and no results were cached. We plan to implement additional query strategies and evaluate them in the near future on multiple query workloads.

When a new query arrives in the system, the query is broadcast to all the nodes. We have implemented a *query launcher* module as an extension to the query server (Section 2) for this purpose. The launcher module is executed by one thread on one SMP node. That thread polls for queries, and upon receiving one, broadcasts it to all the nodes. The query is executed in four main steps:

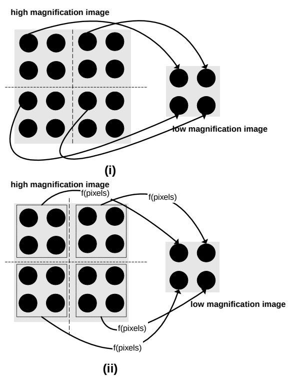
(1) **Initialization.** Accumulator elements for the query are allocated and initialized on each SMP node.
(2) **Local Processing.** Local input data that intersects the query window is retrieved from disk and aggregated into the accumulator elements allocated in step 1.
(3) **Global Combine.** Results computed in each node in step 2 are combined across all nodes to calculate the final intermediate results and final output.
(4) **Output.** Output is sent back to the client.

The query evaluation structure of the replicated accumulator schemes is similar to the execution of a query on a shared-memory system using the original middleware [2]. In the initialization phase, each SMP node allocates and initializes the accumulator structure for a given query. The data store is searched to find the cached results that can be reused to answer the query. The accumulator is divided into two types of regions; one that requires input data, and the other that uses intermediate results from previous queries. Only the *local* input data that intersects the first type of region is read from the disk(s) attached to the SMP node. At the end of step 2, each node has computed intermediate results using its local cached results or input data. As a result, the accumulator on each node contains partial intermediate results, and a *global combine* step is necessary to compute final intermediate results, and eventually the output.

The global combine step can be executed in different ways using the FRA and SRA strategies, and the strategy can be selected by the application developer based on application characteristics, as well as the cluster network configuration. The first strategy, Global Combine at Server (GCS), performs the global combine at the server, as seen in Figure 2(a), and therefore leverages the computational resources and network bandwidth available within the SMP cluster. In this strategy, once a query is received a master node is assigned to that query that is responsible for collect-



(a)



(b)

Figure 2. (a) The Global Combine strategies. The upper diagram shows the Global Combine at Client (GCC) strategy and the lower one shows Global Combine at Server (GCS). When GCS is used, a master node (shown in solid black) is needed for each query. (b) The Virtual Microscope zooming operators - (i) subsampling and (ii) pixel averaging.

ing the query results from other nodes and returning the output to the client. In the current implementation, nodes are assigned as master nodes in round-robin order. For the Global Combine at Client (GCC) strategy, each processor ships its results to the client, as seen in Figure 2(a), which performs the global combine of partial results. This strategy is possible because both the client and server have access to the query object over the entire lifetime of a query (the client instantiates the query object and hands it off to the query server). We will see that GCC is potentially beneficial for queries in which SRA is used and the combine operation is inexpensive. This strategy off-loads some of the computation from the server to the client so that the server can process other queries.

We have added two methods to the *Query* base class in the original system to implement the global combine phase for the GCS scheme. The `send` method takes a pointer to the local accumulator structure, query meta-data and accumulator meta-data, and returns a list of nodes, and for each node a pointer to a buffer. On each SMP node, the buffer pointer for a remote node points to the portion of the local accumulator that will be sent to the corresponding node. The `combine` method takes a pointer to the local accumulator buffer and a pointer to the buffer received from a remote node. The `combine` method is called by the runtime system when a node receives a message, to merge the local accumulator values with the received accumulator values. After a node has received all the remote accumulator elements and combined them with local accumulator elements, the `project` method is called on the final intermediate results to compute the final output. The `combine` method is expected to be implemented by the application developer for application-specific global combine operations. The middleware provides a default implementation for the `send` method. The current default implementation sends the local accumulator elements in each node to the master node assigned for the query. However, this method can also be customized by the application developer for application or hardware specific optimizations[3].

After a query has been executed, each node has partial intermediate results for the query. These results are stored in the local accumulator on each node and maintained in the local data store for possible reuse by future queries. When a new query is received, and if it can be answered by cached results, only the global combine and output phases are executed. That is, the cached partial results that can be used to answer the query are extracted from the data store and used to produce new intermediate results. This is the characteristic that makes our middleware particularly suitable to handle multiple query workloads, especially when intra- and inter-query locality is present.

Note that the class of queries targeted in this work involves aggregation/reduction operations on input data, and the size of the output is often much smaller than the size of the input dataset. As a result, the replicated accumulator strategy is likely to incur less communication overhead than executing multiple queries such that each query executes sequentially on an SMP node. In addition, the RA schemes achieve better load balance and better utilization of distributed processing power, when there are fewer queries than the number of processors.

[3]We are in the process of implementing different versions of the send method to minimize communication overheads.

4 Example Application: The Virtual Microscope

The Virtual Microscope (VM) application [1] implements a realistic digital emulation of a high power light microscope. VM can be used in a training environment, where a group of fellows or students may examine and manipulate the same set of slides. In such a setting, the data server has to process multiple queries simultaneously.

The input datasets for VM are digitized, 2-dimensional full microscope slides. Each digitized slide can be up to several gigabytes in size, and is stored on disk at the highest magnification level. In order to achieve high I/O bandwidth during data retrieval, each slide is regularly partitioned into data chunks, each of which is a rectangular subregion of the 2D image and corresponds to fixed-size pages in our framework. In this paper, the data chunks are row-wise ordered and distributed to the disks in the SMP cluster in a round-robin fashion. Each pixel in a chunk is associated with a coordinate (in x- and y-dimensions) in the entire image. Since the image is regularly partitioned into rectangular regions, a simple lookup table consisting of a 2-dimensional array corresponding to the bounding boxes of data chunks serves as an index.

During query processing, the chunks that intersect the query region, which is a two-dimensional rectangle within the input image, are retrieved from disk. Each retrieved chunk is first clipped to the query window. Each clipped chunk is then processed to compute the output image at the desired magnification. We have implemented two functions to process high resolution clipped chunks to produce lower resolution images, each of which results in a different version of VM as can be seen in Figure 2(b). The first function employs a simple subsampling operation, and the second implements an averaging operation over a window. For a magnification level of N given in a query, the subsampling function returns every N^{th} pixel from the region of the input image that intersects the query window, in both dimensions. The averaging function, on the other hand, computes the value of an output pixel by averaging the values of $N \times N$ pixels in the input image. The *averaging* function can be viewed as an image processing algorithm in the sense that it has to aggregate several input pixels in order to compute an output pixel. Algorithms such as image enhancement and automatic feature extraction would have similar relative computing and I/O requirements. The accumulator for these functions is a 2-dimensional pixel array, each entry of which stores values for a pixel in the lower resolution output image.

We have added a query object to the runtime system for each of the processing functions. The magnification level, the processing function, and the bounding box of the output image in the entire dataset are stored as meta-data. An `overlap` function was implemented to intersect two regions and return an overlap index, which is computed as

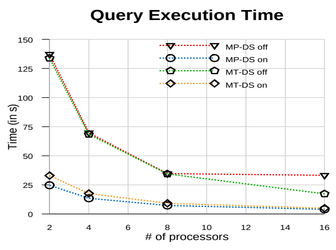
$$overlap\ index = \frac{I_A}{O_A} \times \frac{I_S}{O_S} \qquad (1)$$

In this equation, I_A is the area of intersection between the intermediate result in the data store and the query region, O_A is the area of the query region, I_S is the zooming factor used for generating the intermediate result, and O_S is the zooming factor specified by the current query. O_S should be a multiple of I_S so that the query can use the intermediate result. Otherwise, the value of the overlap index is 0.

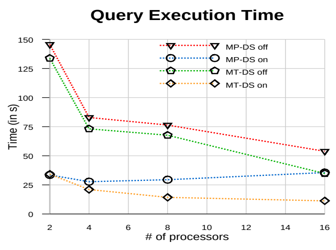
For execution on a cluster of SMPs, we have implemented two different accumulator strategies. The first implementation creates a copy of the full accumulator structure on each node. In the local processing phase, output pixels generated by processing input data chunks, or cached results, are stored in the full accumulator. In the global combine phase, each node forwards the local full accumulator to the master node. This implementation will likely incur high interprocessor communication volume, and memory on each node is not efficiently utilized. The second implementation allocates only the accumulator elements in each node for which there are local cached results or local input elements. Once the local input elements and cached results that contribute to the accumulator elements have been determined (by a lookup into the index and searching the data store), the accumulator is partitioned into rectangular regions. Each region corresponds to a portion of the accumulator that is entirely covered by a subset of the input elements and/or cached results. If the regions are allocated separately, during the global combine phase either multiple messages must be generated to send the regions to the master node, or the regions should be packed into a compact buffer, requiring each region to be copied into the buffer. In order to avoid these overheads, in the initialization phase a buffer large enough to hold all of the regions is allocated and each region is assigned a place in this buffer. In order to do this, we have extended the Data Store manager (see Section 2) to include a method that allocates a buffer without registering any meta-data information.

5 Experimental Evaluation

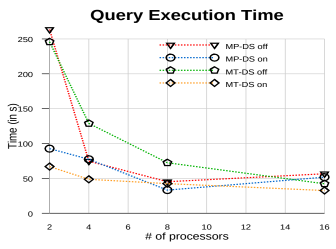
We show experimental results with several configurations, varying the version of VM used, the global combine strategies, the accumulator handling strategies, as well as employing multi-threaded vs. multi-process execution. The experiments ran on a cluster of eight dual-processor 550MHz Pentium III nodes, each with 512KB cache, 1GB of memory and 36GB of disk storage. The nodes are interconnected via a Gigabit Ethernet switch.



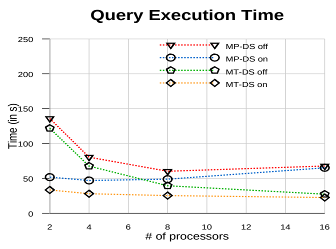
(a) GCC-SRA



(b) GCS-SRA



(c) GCC-FRA



(d) GCS-FRA

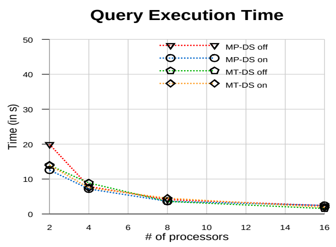
Figure 3. Processor scalability for the pixel averaging implementation of VM using different global combine and accumulator handling strategies. GCC is Global Combine at Client and GCS is Global Combine at the Server. FRA is Fully Replicated Accumulator and SRA is Sparsely Replicated Accumulator. MP denotes Multiple Processes, meaning that 2 *processes* are executed on each node as opposed to 2 *threads* for MT (Multiple Threads). DS $\{\text{on}|\text{off}\}$ specifies whether the partial results stored at the Data Store manager are used during query execution.

For the experiments, we have employed two datasets, each of which is an image of size 30000×30000 3-byte pixels, requiring a total of 7.5GB storage space. Each dataset was partitioned into 64KB pages, each representing a square region in the entire image. These pages were declustered in round-robin fashion across the nodes, and stored on the local disk attached to each node. We have emulated 16 concurrent clients. Each client generated a workload of 16 queries (8 queries for the more computationally expensive pixel averaging version), producing 1024×1024 RGB images (3MB) at various magnification levels. Of the 16 clients, 8 issued queries to the first dataset, and 8 submitted queries to the second dataset. We have used the driver program described in [4] to emulate the behavior of multiple simultaneous clients. The implementation of the driver is based on a workload model that was statistically generated from traces collected from experienced VM users. We have chosen to use the driver for two reasons. First, extensive real user traces are very difficult to acquire. Second, such an emulator allows us to create different scenarios and vary the workload behavior (both the number of clients and the number of queries) in a controlled way.

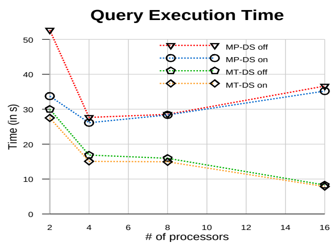
The experiments using the subsampling implementation of VM show the system behavior when the queries are essentially I/O intensive, and the pixel averaging algorithm shows the system performance, when queries are computationally more expensive, so more balanced between the time spent on computation and on I/O. We have determined in [3] that the CPU time to I/O time ratio is between 0.04 and 0.06 for the subsampling implementation (i.e., for each 100 seconds, between 4 and 6 seconds are spent on computation, and between 94 and 96 seconds are spent doing I/O). The averaging implementation is more balanced, with the CPU and I/O times nearly equal. We show scalability results for different strategies when the number of processors is varied. Hence, we fixed the total aggregate amount of memory available for the Page Store Manager (PS) at 64MB and for the Data Store Manager (DS) at 128MB. That is, for a configuration with P nodes, each node allocates $\frac{64}{P}$MB for PS, and $\frac{128}{P}$MB for DS.

Figures 3 and 4 show experimental results for processor scalability for the pixel averaging and subsampling implementations, respectively. We now evaluate the results according to several criteria.

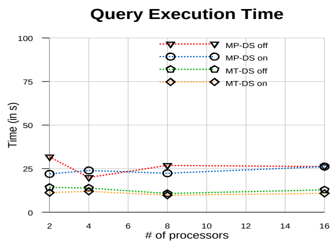
Using multiple query optimization support (DS is on). The major strength of our middleware lies in its ability to leverage intermediate results previously computed. As is seen from the figures, performance improves when results



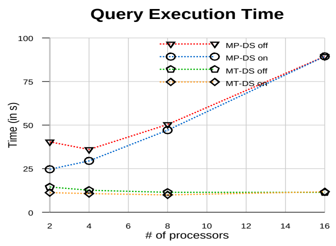
(a) GCC-SRA



(b) GCS-SRA



(c) GCC-FRA



(d) GCS-FRA

Figure 4. Processor scalability for the subsampling implementation of VM with different global combine and accumulator handling strategies.

are cached in the data store. We see in Figure 3(a) that this improvement can be up to a three-fold decrease in the query execution time, although the improvements are not that high in general. Note that even for configurations that do not show good scalability, using DS usually decreases execution times.

Multi-process (MP) *versus* Multi-threaded (MT) Execution. Each of the nodes in our cluster is has two processors, hence it is possible to run two processes or only one process with two threads on a node. In the majority of cases, MT performs better than MP. As is seen from Figures 4(b) and (d), multi-threading can greatly improve scalability. Multi-process execution allows a greater degree of intra-query parallelism, since both processes can work on the same query simultaneously. Multi-threaded execution, on the other hand, enables a higher degree of inter-query parallelism because two queries can be executed simultaneously, resulting in better utilization of cached results. In addition, communication cost per query is lower than for MP.

Fully Replicated Accumulator *versus* Sparsely Replicated Accumulator. The Virtual Microscope queries can be handled well with the Sparsely Replicated Accumulator strategy, because VM is regular, in the sense that the input data on each processor always maps to the local portion of the accumulator allocated on that processor. We show experimental results for the Fully Replicated Accumulator strategy because many other data analysis applications map local input data or cached results into the entire accumulator. As can be seen by comparing the results in Figure 4(a) vs. 4 (c), and in Figure 4(b) vs. 4 (d), using SRA results in almost perfect scalability in 4(a) and reasonably scalable behavior when the DS optimizations are turned on (Figure 4(b)). For the pixel averaging results shown in Figure 3, FRA does not have such high overhead because the computation cost is much higher than for the subsampling implementation. Therefore the extra communication does not have as big an impact on overall performance.

Global Combine at the Client *versus* Global Combine at the Server. The final phase of query execution is the global combine, where the multiple pieces of the accumulator are combined into a final result. As we previously described, our middleware is able to perform the combine at the client or at the server. Each of these strategies has both benefits and drawbacks. Offloading the Global Combine to the client removes the computational and communication burden from the server, which in high server workload situations may improve overall system performance. This is especially true when the SRA strategy is used for query evaluation, because the total amount of communication will be the same as if the Global Combine were executed at the server. This is exactly the behavior we observe in Figures 3(a) and Figure 4(a). In the GCC-SRA configuration, both VM implementations show almost perfect scalability up to 16 processors.

Our overall results show that for both implementations of VM, the ideal system configuration is SRA with GCC.

SRA achieves good performance, because each input element (pixel) corresponds to a single output element, thus all the operations and data structures can be evenly partitioned across the processors. GCC achieves good performance, because the global combine operation for VM is simply to *stitch* together the individual image pieces computed at each processor. There is no extra computation in the global combine where the parallel server could be beneficial. Therefore shipping the results directly to the client lowers the overall communication cost compared to GCS. On the other hand, if the Global Combine were an expensive operation, leveraging the parallel and multithreaded capability of the server should lead to better performance. When FRA is used, performing the Global Combine at the server would be beneficial, because of the amount of communication involved. The advantage is actually two-fold, both from getting the results from each of the processing nodes faster, assuming a fast network between nodes in the server, and because only one copy of the accumulator is shipped to the client (as opposed to n - where n is the total number of processes).

6 Conclusions

We have presented a parallel and multithreaded middleware system suitable for the implementation of data analysis applications dealing with large distributed datasets. Its major and novel strength lies in its ability to leverage previously computed results in order to speed up the processing of new queries. It integrates several different strategies for handling partial results using replicated accumulators, and also provides multiple methods for performing the global combine to produce final results.

We have presented experimental scalability results for two implementations of the Virtual Microscope application that show that one particular configuration that employs one way of performing the global combine operation and handling of the accumulator object is much better than all other configurations. Additionally, we have shown that making use of the intermediate results available at the Data Store Manager, to avoid recomputing partial or complete common aggregates, usually improves query response time significantly. In fact, for some cases, reuse is the major factor in explaining the difference between good scalable behavior and poor scalability for the various potential configurations for a particular application.

We are now in the process of implementing new data intensive applications with our middleware to further study and explore the flexibility the system allows with the various execution strategies. We are also starting to experiment with scheduling techniques in the context of parallel execution, much in the way we have done for the single node, multithreaded version we have shown results for in [3].

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 *AMIA98*. American Medical Informatics Association, Nov 1998.

[2] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Efficient execution of multiple workloads in data analysis applications. In *Proceedings of the 2001 ACM/IEEE SC'01 Conference (to appear)*, Denver, CO, Nov. 2001.

[3] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Scheduling multiple data visualization query workloads on a shared memory machine. Technical Report CS-TR-4290 and UMIACS-TR-2001-68, University of Maryland, Department of Computer Science and UMIACS, Oct. 2001. Submitted to IPDPS 2002.

[4] M. D. Beynon, A. Sussman, and J. Saltz. Performance impact of proxies in data intensive client-server applications. In *Proceedings of the 1999 International Conference on Supercomputing*. ACM Press, June 1999.

[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] 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.

[7] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the ACM Symposium on Principles of Database Systems on Principles of Database Systems*, pages 34–43, Seattle, WA, 1998.

[8] A. Gupta, S. Sudarshan, and S. Vishwanathan. Query scheduling in multi query optimization. In *International Database Engineering and Applications Symposium, IDEAS'01*, pages 11–19, Grenoble, France, 2001.

[9] M. H. Kang, H. G. Dietz, and B. K. Bhargava. Multiple-query optimization at algorithm-level. *Data and Knowledge Engineering*, 14(1):57–75, 1994.

[10] 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 SC'99 Conference*. ACM Press, Nov. 1999.

[11] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[12] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *Proceedings of the 1995 ACM-SIGMOD Conference*, pages 104–114, San Jose, CA, May 1995.

[13] Y. Zhao, P. M. Deshpande, J. F. Naughton, and A. Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In *Proceedings of the 1998 ACM-SIGMOD Conference*, pages 271–282, Seattle, WA, 1998.