# Comparing the Performance of High-Level Middleware Systems in Shared and Distributed Memory Parallel Environments*

Jik-Soo Kim†, Henrique Andrade‡, Alan Sussman†

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

‡ IBM T. J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10598
hcma@us.ibm.com

## Abstract

*The utilization of toolkits for writing parallel and/or distributed applications has been shown to greatly enhance developer's productivity. Such an approach hides many of the complexities associated with writing these applications, rather than relying purely on programming language aids and parallel library support, such as MPI or PVM. In this work, we evaluate three different middleware systems that have been used to implement a computation and I/O-intensive data analysis application from the domain of computer vision. This study shows the benefits and overheads associated with each of the middleware systems, in different computational environments and with different workloads. Our results lead the way toward being able to make better decisions for tuning the application environment, for selecting the appropriate middleware, and also for designing more powerful middleware systems to efficiently build and run modern, highly complex applications in both parallel and distributed computing environments.*

## 1 Introduction

The implementation of applications that have high requirements for resources such as computational power and disk bandwidth is challenging, because it is usually necessary to rely on parallel and/or distributed resources. On the other hand, as data from sensors (e.g., microscopes, cameras, satellites, etc.) becomes available at an ever-increasing rate, effective applications for processing and analyzing such data are required. Because of the scarcity and cost of the human resources needed to tackle the task of programming such complex data analysis applications, a number of attempts have been made to help developers become more productive, by providing software tools to aid in that task.

Although many studies have been conducted comparing the relative performance of applications using low-level parallelization and distributed computing mechanisms, not much attention has been devoted to comparing high-level frameworks, the problem we address in this paper. Our group has produced different middleware systems over the last several years, each one based on a set of different environmental and utilization assumptions. They have been shown to be powerful, flexible, and able to capture most of the common support that is necessary to write complex applications, and yet still are able to shield the developer from the intricacies of distributed and/or parallel programming. A number of real-world applications have been implemented with these tools. A comparison of the behavior of three of our middleware systems on the *same* application, with the same workload model, and executing in the same runtime environment using the same resources (machines, network, disks, etc.) will be shown to be very instructive. Such a study not only can highlight each of the systems' strengths and associated overheads, but can also be used to guide the design of better middleware systems and to tune the configuration of applications using these middleware systems in complex computational environments.

In this paper we will experimentally show several results: (1) Workload characterization is the single most important variable to account for in designing runtime middleware systems for high-performance computer vision applications; (2) Detecting and leveraging reuse plays an important role in increasing the throughput of the middleware system and decreasing the user's response time; (3) Coordination amongst competing threads in using I/O resources contributes considerably toward improving system performance; and (4) Deployment of auto-tuning capabilities into

*This research was supported by the National Science Foundation under Grants #EIA-0121161 and #ACI-9619020 (UC Subcontract #10152408), Lawrence Livermore National Laboratory under Grant #B517095, and NASA under Grant #NAG5-12652.

runtime middleware systems is paramount for ensuring high performance in varying computational environments.

The rest of this paper is organized as follows: in Section 2 we describe some of the research that has been done in developing high-level libraries and runtime systems to facilitate the development of parallel data analysis applications. In Section 3, we describe and contrast three different middleware systems developed by our group over the past decade. In Section 4, we describe a case study application from the computer vision domain used to obtain the experimental data analyzed in this paper. In Section 5 we present our comparative study and describe the lessons we have learned that can be used to guide the further development of data analysis middleware systems. And, finally, in Section 6, we summarize our results and suggest open research issues that should be addressed in order to provide self-adaptive middleware systems that enable consistently high performance for data analysis applications.

## 2 Related Work

Over the last decade, many programming paradigms and libraries have become available to ease the process of implementing and deploying complex data analysis and visualization applications. We refer to these paradigms as the *low-level* portion of the spectrum of mechanisms for supporting the implementation of data-/compute-intensive applications, as they typically require a complete understanding (and also the actual programming) of the communication patterns and component interactions within the application. Examples of such efforts are PVM (Parallel Virtual Machine) [23] and the MPI (Message Passing Interface) [35] standard. At the other end (i.e., *high-level*) of the spectrum, middleware systems and customizable toolkits that hide many of the low-level details of writing and deploying a parallel/distributed data analysis application have also been designed and built. Examples of such toolkits are parallel I/O libraries such as Passion [38] and Panda [19]; C++ tools such as pC++ [41], Chaos++ [18], POOMA [26], and Overture [16]; parallelization tools such as Chaos [27] and Multiblock Parti [1], KeLP [22], Fortran D [25], High Performance Fortran [24], and OpenMP [33]; and parallel visualization toolkits such as the Dv Project [30], VisDB [28], and OpenDX [39]. In many cases, these middleware systems and toolkits rely on low-level mechanisms for handling communication and I/O, but present application templates or operators that can be customized for individual applications. These templates allow many of the details of interprocess communication, I/O, etc. to be hidden from the application writer, essentially freeing up the developer to concentrate on writing the application, rather than focusing on the internal details of the parallel or distributed computing environment.

## 3 Middleware Systems for Data-/Compute-Intensive Applications

Over the past few year, our research group has developed multiple middleware systems for efficiently supporting data analysis applications. All the systems target *multi-dimensional range queries* with user-defined aggregation operations, which often arise in various data analysis applications. A range query specifies the data to process via a bounding box in the underlying multi-dimensional attribute space of the (input and/or output) datasets, while aggregation operations are commutative and associative computations that combine multiple input data elements into a single output data element. The Active Data Repository (ADR) [10, 29] represents the result of our first design. ADR's algorithms and runtime system target architectures that range from tightly coupled shared-memory machines to distributed-memory parallel machines with attached disks in a cluster configuration.

The realization that heterogeneous environments were becoming common both in academic as well as in commercial computing environments, and the formalization of Grid computing concepts implied a fundamental modification in how complex applications could be decomposed. These multi-component applications could be distributed on top of resources with different physical characteristics. Each component could be seen as *filtering* the incoming stream of data, where each *filter* represents a different stage of the computation. This is the central idea behind the DataCutter (DC) framework [11, 36].

While DC can only process one query at a time, with other submitted queries enqueued for execution, ADR is able to process multiple queries simultaneously. ADR processes queries in batches, but must complete one query batch before starting another batch. Neither ADR nor DC implement techniques for optimizing and removing redundancies that may arise when multiple queries are presented to the system, which is one of the main features of the Multi-Query Optimization (MQO) framework. MQO not only can handle multiple simultaneous queries, but also can incorporate queries into its current query execution plan as they are submitted.

As a result of larger datasets becoming available and also as a result of the success of middleware systems such as ADR and DC, data analysis is increasingly being employed in collaborative environments. That is, multiple clients access the same datasets and perform similar processing on the data. For instance, in medical imaging [8], one scenario is that a large group of students want to simultaneously explore the same set of digitized microscopy slides or visualize the same Magnetic Resonance Imaging (MRI) and Computerized Tomography (CT) results. In these situations, the data server needs to execute multiple queries

simultaneously to minimize latencies to the clients.

Many different aspects of the multiple query optimization problem have been studied in other contexts, particularly in relational databases. In the context of scientific data analysis applications, however, the scale of the datasets, the application-specific nature of the data structures, and the computation of user-defined aggregates require new optimization techniques to ensure good system performance, especially under heavy workloads. In addition to leveraging several ideas previously developed in ADR and DC, these are the problems targeted by the MQO [4, 7].

We now describe some of the most important architectural details of the three middleware systems.

### 3.1 The Active Data Repository

The implementation of data analysis operations on a parallel machine requires distribution of data and computations among disks and processors to make efficient use of all available storage space and computing power. Careful scheduling of data retrieval, computation and network operations to keep all resources (i.e., disks, processor memory, network, and CPU) busy without overloading any of the resources is also needed. We have developed the Active Data Repository (ADR) to provide support for applications that perform multi-dimensional range queries with user-defined aggregation operations on multi-dimensional datasets, to be executed on a distributed-memory parallel machine with an attached disk farm. In this section, we briefly describe ADR, and present the algorithms and optimization techniques developed in the ADR framework.

Both input and output datasets in ADR are partitioned into and stored as sets of data chunks. A *data chunk* contains a subset of the data items in the dataset. A dataset is partitioned into data chunks by the application developer, and data chunks in a dataset can have different sizes. Since data is accessed through range queries, it is desirable to have data items that are close to each other in the multi-dimensional space placed in the same data chunk. A data chunk is the unit of data retrieval. That is, it is retrieved as a whole during processing. Retrieving data in chunks instead of as individual data items reduces I/O overheads (e.g., disk seek time), resulting in higher application level I/O bandwidth. As every data item is associated with a point in a multi-dimensional attribute space, every data chunk is associated with a minimum bounding rectangle (MBR). The MBR of a data chunk is the smallest box in the underlying multi-dimensional space that encompasses all the coordinates of all the items in the data chunk.

Data chunks are distributed across the disks in the system to fully utilize the aggregate storage space and disk bandwidth. To take advantage of the data access patterns exhibited by range queries, data chunks that are close to each other in the underlying attribute space should be assigned to different disks. In the ADR framework, we employ a Hilbert curve-based declustering algorithm [21] to distribute the chunks across the disks. Hilbert curve algorithms are fast and exhibit good clustering and declustering properties. Other declustering algorithms, such as those based on graph partitioning [31], can also be specified by an application developer using ADR. Each data chunk is assigned to a single disk, and is read and written only by the local processor to which the disk is attached. If a chunk is required for processing by one or more remote processors, it is sent to those processors as a whole by the local processor via interprocessor communication. After data chunks are assigned to disks, a multi-dimensional index is constructed using the MBRs of the chunks. The index on each processor is used to quickly locate the chunks with MBRs that intersect a given range query. Efficient spatial data structures, such as R-trees and their variants [9], can be used for indexing and accessing multi-dimensional datasets.

The processing of a range query in ADR is accomplished in two steps: a *query plan* is computed in the *query planning* step, and the actual data retrieval and processing is carried out in the *query execution* step according to the query plan.

Query planning is carried out in three phases: *index lookup*, *tiling*, and *workload partitioning*. In the index lookup phase, indices associated with the datasets are used to identify all the chunks that intersect with the query. If the output data structure is too large to fit entirely in memory, it must be partitioned into *tiles*, each of which contains a disjoint subset of output elements. Partitioning is done in the tiling phase so that the size of a tile is less than the amount of memory available for the output. A tiling of the output implicitly results in a tiling of the input dataset. Each *input tile* contains the input chunks that map to the corresponding output tile. Since an input element may map to multiple output data elements, the corresponding input chunk may appear in more than one input tile if the output chunks are assigned to different tiles. During query execution, the input chunks placed in multiple input tiles are retrieved multiple times, once per output tile. Therefore, care must be taken to minimize the boundaries of output tiles so as to reduce the number of such input chunks. In the workload partitioning phase, the workload associated with a tile is partitioned among processors. In the query execution step, the processing of an output tile is carried out according to the query plan. A tile is processed in four phases– a query iterates through these phases repeatedly until all tiles have been processed and the entire output has been computed: (1) **Initialization**: Output elements for the current tile are allocated space in memory and initialized; (2) **Reduction**: Each process retrieves data chunks stored on local disks. Data items in a data chunk are aggregated into the output elements allocated in the memory of each process during

phase 1; (3) **Global Combine**: If necessary, partial results computed by each process in phase 2 are combined across the processes via inter-process communication to compute final results for the output; (4) **Output Handling**: The final output for the current tile may optionally be transformed via a user-defined operation on the values computed in phase 3. The output is either sent back to a client or stored back to the disks.

We have developed an implementation of the ADR framework as a set of modular services, implemented as a C++ class library, and a runtime system. Several of the services allow customization for user-defined processing. A unified interface is provided for customizing these services via C++ class inheritance and virtual functions. An application developer has to provide output data structures and functions that operate on *in-core* data, to implement application-specific processing of *out-of-core* data with ADR.

An ADR application consists of one or more *clients*, a *front-end process*, and a customized *back-end*. The front-end interacts with clients, translates client requests into queries and sends one or more queries to the parallel back-end. Since the clients can connect and generate queries in an asynchronous manner, the existence of a front-end relieves the back-end from being interrupted by clients during processing of queries. The back-end is responsible for storing datasets and carrying out application-specific processing of the data on the parallel machine.

The back-end runtime system provides support for common operations such as index lookup, management of system memory, and scheduling of data retrieval and the processing operations described above across a parallel machine. During the processing of a query, the runtime system tries to overlap disk operations, network operations, and processing as much as possible. Overlap is achieved by maintaining explicit queues for each kind of operation (data retrieval, message sends and receives, data processing) and switching between queued operations as required. Pending asynchronous I/O and communication operations in the operation queues are polled and, upon their completion, new asynchronous operations are initiated when more work is required and memory buffer space is available. Data chunks are therefore retrieved and processed in a pipelined fashion.

### 3.2 DataCutter

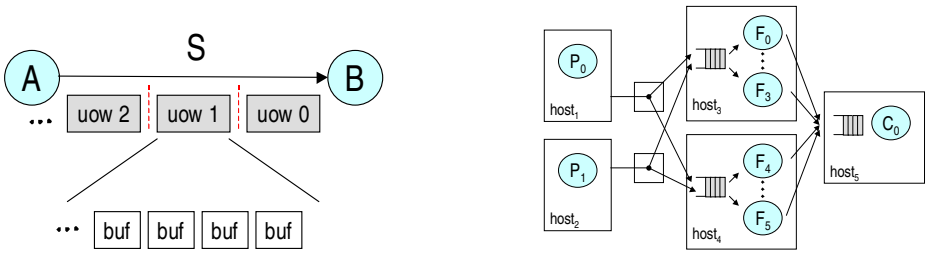
The data processing structure of an application implemented with DataCutter (DC) is represented as a set of application filters. A *filter* is a user-defined object that performs application-specific processing on data. A filter object must have three functions[1]; **init**, **process**, and **finalize**, that are implemented by the application developer. A *stream* is an abstraction used for all filter communication, and specifies how filters are logically connected. A stream also denotes a supply of data to and from the storage media, or a flow of data between two separate filters, or between a filter and a client. Bi-directional data exchange can be achieved by creating two pipe streams in opposite directions between two filters. All transfers to and from streams are through a provided buffer abstraction (Figure 1). A buffer represents a contiguous memory region containing useful data. The **init** function is where any required resources such as memory can be pre-allocated for a filter. The **process** function of the filter is called by the runtime system to read from any input streams, work on data buffers received, and write to any output streams. The **finalize** function is called after all processing is finished, to allow release of allocated resources such as scratch space.

A set of filters that collectively carry out application-specific data processing is referred to as a *filter group*. The DC runtime system provides a multi-threaded, distributed execution environment. Multiple instances of a filter group can be instantiated and executed concurrently; work (i.e. queries) can be assigned to any group. Within each filter group instance, multiple copies of individual filters can also be created transparently to the application. Filters co-located on the same machine are executed as separate threads by the runtime system. Data exchange between two co-located filters is carried out by simple pointer copy operations, thereby minimizing communication overhead. Communication between filters on different hosts is done through TCP/IP sockets.

Decomposition of application processing into filters and filter copies effectively provides a combination of task and data parallelism (Figure 1(b)). Filter groups and individual filters can be placed on a distributed collection of machines to minimize computation and communication overheads. Filters that exchange large volumes of data can be placed on the same machine, while compute intensive filters can be executed on more powerful or less loaded machines.

An application implemented with DC is structured in a similar way to an ADR implementation. The application consists of one or more *clients*, a *console process*, and at least one filter group. The console process interacts with clients, and forwards each client request to a filter group instance for data retrieval and processing (more than one instance can be created for a filter group, for example on different sets of machines). The filter group instances are responsible for accessing stored datasets and carrying out

[1] The DC framework supplies a base class with three virtual functions (init, process, finalize). An application specific filter is derived from this base class, and application operations are implemented in the three functions.




(a) (b)

**Figure 1. DataCutter stream abstraction and support for copies. (a) Data buffers on a stream. (b) Filter group with three filters, labeled P,F,C, instantiated using transparent filter copies.**

the application-specific processing of the data on the available resources, typically a heterogeneous distributed set of machines. In the current DC implementation, all filters assigned to a given machine are run as threads within a single process, called an *application daemon*, or *appd*, because the process provides DC services on that host for the lifetime of the application.

### 3.3 The Multi-Query Optimization Framework

Previously existing techniques for taking advantage of reuse in relational databases suffer from limitations in handling either unstructured operators or operators that are not known *a priori*. In fact, most of the techniques assume a well-defined set of operators – relational database operators – and also assume specific algorithms implementing the operators. Such a restricted set of operators allows for defining efficient data structures for representing data and computation reuse scenarios. However, in more general data analysis applications, there is no pre-defined set of operators and algorithms that can be leveraged to construct an efficient query optimizer. Thus optimizations must rely on abstract operators that lend themselves to customizations. In this way, new operators and algorithms can be added dynamically to a data analysis application. This makes it necessary to provide ways to describe operators and the aggregates generated by such operators so that a generic query optimizer can automatically detect optimization opportunities.

The Multi-Query Optimization (MQO) framework targets a data processing model that is common for these types of queries. MQO provides the following set of abstract operators used to identify and exploit data reuse opportunities during query planning and execution:

$$\mathbf{compare}(M_i, M_j) = \textit{true} \text{ or } \textit{false} \quad (1)$$

$$\mathbf{overlap_{project}}(M_i, M_j) = k, \text{where } 0 \leq k \leq 1 \quad (2)$$

$$\mathbf{project}(M_i, M_j, \mathcal{J}) = \mathcal{K}_{M_k}, \text{where } M_k \subseteq M_i \text{ and } \mathcal{K} \subseteq \mathcal{I} \quad (3)$$

Equation 1 describes the $compare$ function that returns *true* or *false* depending on whether aggregate $\mathcal{I}$, described by meta-data $M_i$, is the same as aggregate $\mathcal{J}$, described by $M_j$. When the application of this function returns true, the framework has identified a common subexpression elimination opportunity. Hence, in a query plan where $\mathcal{I}$ is supposed to be computed, the query executor can replace uses of $\mathcal{I}$ with a reference to aggregate $\mathcal{J}$.

In many situations, aggregates $\mathcal{I}$ and $\mathcal{J}$ partially overlap, which means that $compare$ is false, but partial reuse is still possible. Equation 2 describes the $overlap$ function that returns a value between 0 and 1 that represents the amount of overlap between aggregates $\mathcal{I}$ and $\mathcal{J}$. This function is computed by inspecting the domain of a cached aggregate (described by $M_j$) and the domain of the query being processed ($M_i$) in two steps. First, the amount of multidimensional overlap with the query domain is computed. Second, another factor of the overlap is computed with respect to a set of data transformation functions (called $project$ functions), which are responsible for identifying each *relevant* element from the cached aggregate and *converting* it (e.g., decreasing resolution, translating or rotating the object) into an output data element for the aggregate being computed. More precisely, the $project$ function, shown in Equation 3, takes one data aggregate $\mathcal{J}$ whose meta-data is $M_j$ and *projects* it over $M_i$ by extracting and transforming the parts of $\mathcal{J}$ that are relevant to $\mathcal{I}$, either computing $\mathcal{I}$ completely or generating $\mathcal{K}$, which partially computes $\mathcal{I}$. Each projection function manipulates the input aggregate in different ways to convert it into the aggregate required by the query being computed.

Upon identifying a reusable aggregate, the query executor must complete the computation of the desired aggregate by evaluating the portions that need to be computed from input data or from other aggregates. The following equation defines an operator that computes the set of meta-data descriptors for the incomplete regions:

$$\mathbf{difference}(M_i, M_j) = C \quad (4)$$

where $C$ is the set of subqueries needed to compute the remaining parts of the query.

Dealing with the application-specific nature of data analysis queries requires customization of each of the operators in the context of a particular application, which is achieved in our implementation through C++ inheritance [3].

In MQO query planning and execution are carried out in two steps that are only sketched here; the detailed algorithm is described in [2]. In the first step, a decision is made as to which query from the waiting queue is selected for execution [5]. The second step determines the query plan to be employed in order to compute the query [6]. For each query, the data cache is searched for aggregates that overlap the primitive meta-data (using the *compare* and *overlap* operators). If there is complete overlap, the output is computed from the cached aggregate by applying the appropriate projection primitive(s) (using the *project* operator). If cached aggregates can only partially be used to compute the query under evaluation, subqueries are recursively scheduled for computation of the incomplete regions (using the *difference* operator). On the other hand, if no overlap was detected, the query needs to be executed from scratch by computing its result from input data. As a result each output and temporary dataset is tagged and cached for potential future reuse.

## 4 Multi-Perspective Volumetric Shape Analysis

In this section we describe the application we used for the experimental case study described in Section 5.

Modern image analysis and computer vision systems often use *multi-perspective* imaging, which employs multiple cameras shooting the scene of interest from various perspectives. The basic idea is that more views deliver more information about the scene, and potentially allow recovery of interesting 3-dimensional features with high accuracy and minimal intrusion into the scene (e.g., no markers for tracking objects or people through the scene are required). An exciting broad range of applications for such systems includes virtual view rendering, complex shape and movement analysis, multi-person tracking, virtualized reality, and smart environments [15, 17, 20, 32, 34, 37, 40].

The availability of affordable high-speed digital cameras, along with the ever increasing computing capabilities of desktop computers, have made such applications more attractive for use in various computer art and animation studios, medical facilities, business environments and people's everyday lives. Small scale systems (involving 3 cameras or less) can be handled by a modern desktop computer, while larger scale applications could be quite challenging even for high performance computing systems. Performance issues arise because multi-perspective systems with large numbers of cameras can produce and process vast amounts of image data and video streams that can be very difficult to manage without an efficient way to store, catalog, and process the data.

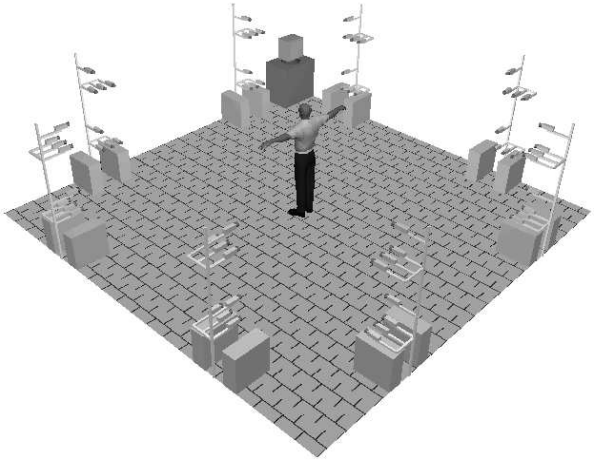


**Figure 2. Keck Lab model**

Figure 2 shows a VRML model of the multi-perspective Keck laboratory at the University of Maryland [20]. One minute of multi-perspective video, shot in this facility, may require up to 95 GB of storage. If the desired processing of the video data cannot be performed in real time, as described in [12], the image data must persist in long-term storage for further off-line processing.

It is still not feasible to manage and process such large quantities of image data on a single PC or workstation, because of performance issues stemming both from accessing the stored data and the computational requirements of the subsequent processing. Therefore it is imperative to store and process those sequences using a middleware system that is able to service data- and compute-intensive applications and leverage storage and processing capabilities from a parallel machine or a cluster of workstations. Much of the processing can be done in parallel, and there is an opportunity to distribute data across a disk farm to achieve high performance.

Although such an application could be implemented *from scratch* using message passing with MPI or PVM as the underlying parallelization model, a data analysis middleware system can offer considerable savings in developer time and effort. Using such a system requires the developer to only customize the middleware with several methods and/or operators that are specific to the application, thereby isolating her from ensuring correctness in dealing with the complexities of a parallel or distributed computing environment, such as interprocessor communication and I/O scheduling, data distribution, managing shared data structures, and other tasks that are usually time-intensive for the developer and require considerable technical expertise.

### 4.1 Algorithms for Reconstructing 3D Volume

The multi-perspective volumetric reconstruction procedure is based on *visual cone intersection* and the details are described in [13, 14]. A visual cone is the portion of 3D space that a camera can see from its particular vantage point. The main idea is to efficiently build a 3D *Volume* (represented by an *occupancy map*) of the foreground object(s) in a scene by using 2D silhouette image data from all available cameras. The process assumes that the images from all the cameras are synchronized in time, as is the case in the Maryland Keck lab. The algorithm is applied to the images from all the cameras at a given point in time (a *frame*), although for performance reasons multiple volumes for different frames can be produced simultaneously. The reconstruction algorithm uses the 2D image data to determine the occupancy of the space bounded by a cube at a given resolution. At each step in the algorithm, if the occupancy of the cube is undetermined, meaning that it has not yet been determined whether it is completely occupied or completely unoccupied, the algorithm is invoked for each of the cube's eight sub-cubes recursively. This procedure finally produces an occupancy map, stored compactly as an *octree*, that approximates the space occupied by the object(s) in the full 3D space. However, since we are dealing with 3D space, we only obtain a *partial occupancy map* from each 2D image. Therefore, the overall algorithm must intersect all the partial octrees from all the camera images (i.e., perform visual cone intersection) to produce the complete reconstruction for a frame.

The steps of the volumetric reconstruction algorithm can be performed sequentially on a single machine. However, it is easy to see that there are many possibilities for the algorithm to be parallelized to improve performance, by retrieving all available image data and producing multiple partial occupancy maps simultaneously. That is the overall strategy for parallelizing the algorithm taken for all the middleware systems described in Section 3.

### 4.2 Implementing the Volumetric Reconstruction Application

In order to implement the volumetric reconstruction application with our middleware systems, we started with a single common, source code and customized and configured the middleware systems appropriately. We now highlight the most important aspects of the customization process.

For the volumetric reconstruction application, we have customized the parallel back-end of ADR. During the Initialization and Reduction steps for computing an output frame, ADR retrieves required image data from all available cameras and produces partial occupancy maps in parallel via the user-provided indexing and aggregation functions. Each process is responsible for computing its partial occupancy maps, represented as octrees, based on image data stored locally. Finally, ADR produces the complete 3D Volume for a frame through the Global Combine step by intersecting all partial occupancy maps across all processes.

For implementing the volumetric reconstruction application with the DC framework, as for the ADR implementation, we can produce a 3D volume for a frame in parallel. The implementation requires three filters – ImageReader, LocalCombiner and GlobalCombiner. The ImageReader filter reads the required image data from the disk(s) where the data is stored and sends the data to the LocalCombiner filter through the filter output buffer. The LocalCombiner filter gets input image data from ImageReader filter and computes partial occupancy maps and sends them to GlobalCombiner. Finally, the GlobalCombiner filter accumulates all required partial occupancy maps from the LocalCombiner filter and generates the final complete 3D space. The sequence of filters is specified in DC as a single logical pipeline, with the ImageReader output connected to the LocalCombiner input with one *stream* and the LocalCombiner output connected to the GlobalCombiner input with another *stream*.

DC supports replication of individual filters to enable data parallelism via transparent copies, while still supporting the abstraction of a single logical stream connecting a pair of filters. Data written onto the same logical stream from multiple producer filter copies is multiplexed into the stream, while data read from the stream is directed to different copies of a consumer filter either in a *round-robin* fashion or via a token-based, *demand driven* policy based on how fast filter copies are consuming the data on the stream (similar to TCP-based flow control). This DC feature enables effective dynamic scheduling of filters onto the multiple machines to enable parallel execution. However, if multiple filter copies are deployed onto a single SMP machine, there can be some overhead caused by contention among the multiple consumer filter copies attempting to read from the same input stream. In the SMP environment, because reading data from the input stream is a *destructive* operation, the DC runtime system must lock the stream buffer before reading and returning the data requested by a filter copy from the stream. Such locking can incur substantial overhead if many filter copies are reading from the same stream. For the volumetric reconstruction application, the ImageReader and LocalCombiner filters can be, for example, replicated onto each host that stores image data, while only a single copy of the GlobalCombiner filter is needed to combine all the partial occupancy maps for a frame into the complete map for the frame, to be returned to the requesting client. More details on the configuration of DC filters for our experiments will be presented in Section 5.

As do the ADR and DC implementations, the MQO im-

plementation processes volumetric reconstruction queries in parallel through its *operators*, but performs the required operations in a rather different way. To produce a 3D volume with ADR or DC, all the image data specified by the range query must always be retrieved to produce the desired partial occupancy maps. However, if there are overlaps in the data and/or processing required across different volumetric reconstruction queries, MQO can reduce the amount of work it must perform to satisfy the queries, since it will only retrieve and process data for computations whose results have not been already produced and stored in the MQO cache. In addition, MQO can use cached intermediate results to compute query results if additional processing can be performed to produce the desired results. In an SMP environment, MQO's query server can either run as a multi-threaded application (denoted as MQO-SMP) or as an MPI-based application (denoted as MQO-MPI). In the SMP version, MQO can execute multiple queries simultaneously by allocating a thread or a group of threads to each query (i.e., queries can be *a priori* partitioned). In the MPI version, all MPI processes collaborate to execute a single query. The simultaneous execution of multiple queries contributes to decreasing the amount of time a query will wait before being scheduled for execution at the expense of requiring more time for processing (since less computational power is available for each individual query).

# 5 A Comparative Analysis

In this section, we compare and analyze the performance of the three frameworks, supporting the execution of multiple Volumetric Reconstruction (VR) queries. The VR application has been implemented using each of the middleware systems, using much of the same source code and operators. To provide additional insights from this comparison, we have employed two quite different environments for running the middleware systems, a *cluster* environment and a *shared memory* environment.

The cluster environment is a Linux cluster containing 17 Pentium III 650 MHz nodes, each of which has 768 MB of RAM, and 320 GB of disk storage. The nodes are interconnected via channel-bonded Fast Ethernet (200 Mb/s per node). The shared memory environment is a Solaris Sunfire 6800 with 24 processors, 72 GB of RAM and 4 $\times$ A1000 RAID disk systems. In the shared memory environment, we also use 17 processors for all three frameworks, 16 for the query server and one that hosts the workload generator producing the queries to be processed.

Our test dataset is a multi-perspective sequence of 2600 frames generated by 13 synchronized color cameras, each producing 640 $\times$ 480 pixel images at 30 Hz [14]. The test dataset is partitioned into 32 silhouette image files, each of which is 329 MB in size, and the files are spread across two disks per node on 16 nodes in the cluster environment and stored in 32 different directories on the RAIDs in the shared memory environment. Each of the 32 image files contains a collection of data chunks. A *chunk* of data is a single image whose attributes include a *camera index* and a *timestamp*. Therefore an image can be identified and accessed via its camera index and timestamp. A VR query specifies the 3D region within the overall image space to be reconstructed, a timestamp range (which represents the set of frames for which volumetric models are computed), and a reconstruction resolution (higher resolution results in a reconstruction with finer detail, up to the resolution of the images). A query result is a reconstruction of the foreground object region lying within the query region, encoded as an octree.

## 5.1 The Workload Model

In our experiments, we generated 16 sets of queries, or *batches*. Each batch contains 50 VR queries, with different batches modeling different distributions of query inter-arrival times (exponential distributions with means varying from 4 to 64 in 4-second increments) and other VR query attributes, simulating multiple simultaneous users/clients generating queries. The workload characteristics are shown in Table 1.

The queries in a batch were constructed according to a synthetic workload model (since at this time we do not have enough real user traces for the application). The workload generator emulates a hypothetical situation in which users want to view a 2 to 4-second (at a rate of 10 frames per second) 3D instant replay for *hot* events in, for example, a basketball game. The workload generator takes as input parameters a set of "hot frames" (e.g., slam dunks during the game) that marks the *interesting* scenes, and the length of a "hot interval" (i.e., the duration of the scene), characterized by a mean and a standard deviation.

A query in a batch requests a set of reconstructions associated with frames selected according to the following algorithm. The center of the interval is drawn randomly with a uniform distribution from the set of hot frames (10 hot frames were used). The length of the interval is selected from a normal distribution (each hot frame is associated with a mean video segment length, statistically varying from 34 to 62 frames). Between the first and last frame requested by a particular query, intermediate frames can be skipped, i.e., a query may process every frame, every 2nd frame, or every 4th frame. The skip factor is randomly selected. The output volume resolution and the 3-dimensional query box were fixed (queries reconstructed the entire available volume, and the resolution corresponds to an octree of maximum depth 6), as was the dataset and we have used data from all the available cameras (the system allows for

| *Batch* | *Frames* | *UF* | *FPQ* | *AvgIAT* | *F1* | *F2* | *F3* | *F> 3* |
|---|---|---|---|---|---|---|---|---|
| 0 | 1540 | 400 | 30.80 | 3.67 | 18.0% | 13.2% | 16.8% | 52.0% |
| 1 | 1578 | 395 | 31.56 | 9.73 | 25.8% | 25.8% | 3.0% | 45.3% |
| 2 | 1598 | 400 | 31.96 | 11.17 | 10.8% | 10.8% | 24.8% | 53.8% |
| 3 | 1366 | 401 | 27.32 | 14.77 | 8.2% | 29.7% | 20.4% | 41.6% |
| 4 | 1691 | 386 | 33.82 | 20.62 | 9.3% | 23.1% | 17.9% | 49.7% |
| 5 | 1384 | 374 | 27.68 | 25.59 | 23.3% | 18.7% | 17.1% | 40.9% |
| 6 | 1526 | 397 | 30.52 | 27.92 | 16.6% | 10.6% | 27.2% | 45.6% |
| 7 | 1366 | 365 | 27.32 | 32.56 | 27.7% | 12.3% | 7.1% | 52.9% |
| 8 | 1392 | 372 | 27.84 | 30.37 | 17.2% | 26.3% | 24.5% | 32.0% |
| 9 | 1580 | 416 | 31.60 | 38.15 | 9.6% | 12.3% | 21.9% | 56.2% |
| 10 | 1506 | 392 | 30.12 | 33.79 | 12.2% | 13.0% | 20.9% | 53.8% |
| 11 | 1645 | 400 | 32.90 | 48.74 | 11.2% | 13.8% | 15.2% | 59.8% |
| 12 | 1640 | 387 | 32.80 | 36.04 | 21.2% | 19.6% | 14.5% | 44.7% |
| 13 | 1421 | 396 | 28.42 | 62.06 | 5.8% | 17.2% | 30.1% | 47.0% |
| 14 | 1395 | 359 | 27.90 | 65.58 | 26.2% | 17.5% | 11.7% | 44.6% |
| 15 | 1374 | 362 | 27.48 | 59.82 | 26.8% | 18.2% | 19.1% | 35.9% |

**Table 1. Workload characteristics. For each batch, the *Frames* column shows the total number of frames in the batch, *UF* shows the total number of *unique* frames in the batch, *FPQ* shows the average number of frames per query, *AvgIAT* shows the average inter-arrival time in seconds between queries, *F1* shows the percentage of the number of frames reconstructed only once in the batch, *F2* shows the percentage of the number of frames reconstructed twice in the batch, *F3* shows the percentage of the number of frames reconstructed three times, and *F>3* shows the percentage of the number of frames reconstructed more than three times.**

queries that perform the reconstruction from a subset of the cameras that shot a frame).

From Table 1, we see that all of the batches have a considerable amount of locality. Between 70% and 95% of the frames in any given batch are specified by more than one query in the batch, which indicates the possibility of leveraging considerable amounts of reuse in the MQO implementation. The table also shows that the amount of *load* on the query server varies from very high for Batch 0, where the average query inter-arrival time is around 3.7 seconds, to very low for Batches 13, 14, and 15, where the average query inter-arrival time is as high as 60 seconds. This workload enables us to study the three middleware systems from the perspective of how well they handle workload with varying characteristics, in terms of coordinating the use of I/O and computational resources, as well as how the systems are able to leverage the locality seen in the queries that must be processed.

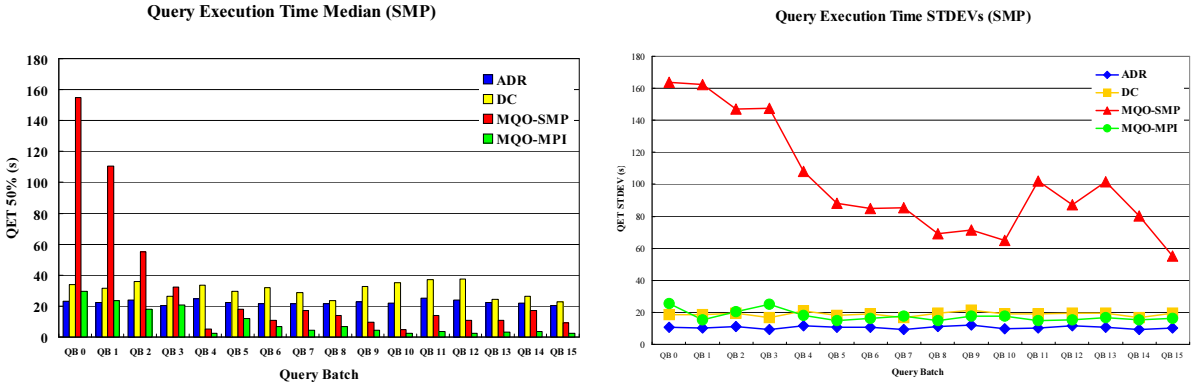
## 5.2 Experimental Metrics

To measure the performances of our three middleware frameworks, we considered the following metrics: *Query Waiting Time* (QWT), *Query Execution Time* (QET), and *Total Query Batch Time* (TotalQBT). QWT is the amount of time from the moment a query was submitted to the system until it gets scheduled for execution. That is, QWT is the query delay before actual processing begins, if the query server is busy. QET captures the elapsed time for a query to complete from the moment it gets scheduled for execution. Finally, TotalQBT captures the total execution time for one query batch. From a user standpoint, lower QET and lower QWT implies faster query turnaround time. Similarly, from the query server perspective, lower TotalQBT implies higher query server throughput.

## 5.3 Shared Memory Environment

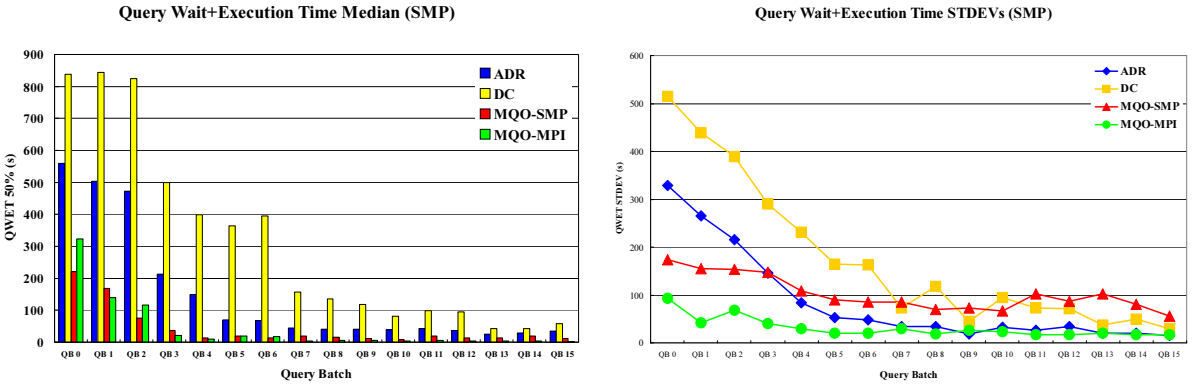
Each of the middleware systems can be hand-tuned by a variety of parameters. In the following experiments, we employed the best configuration found during a set of trial runs. Among other issues, we were particularly careful in ensuring that all three systems used exactly the same amount of resources, in particular, the same number of processors and disks were compiled with the same compiler, and used the same libraries (i.e., ADR and MQO-MPI both were linked against the same MPI library).

In the shared memory environment, there are 32 data files that are stored in 32 different directories. ADR uses 16 different processes/processors to compute the queries. The




(a) Median (b) Standard Deviation

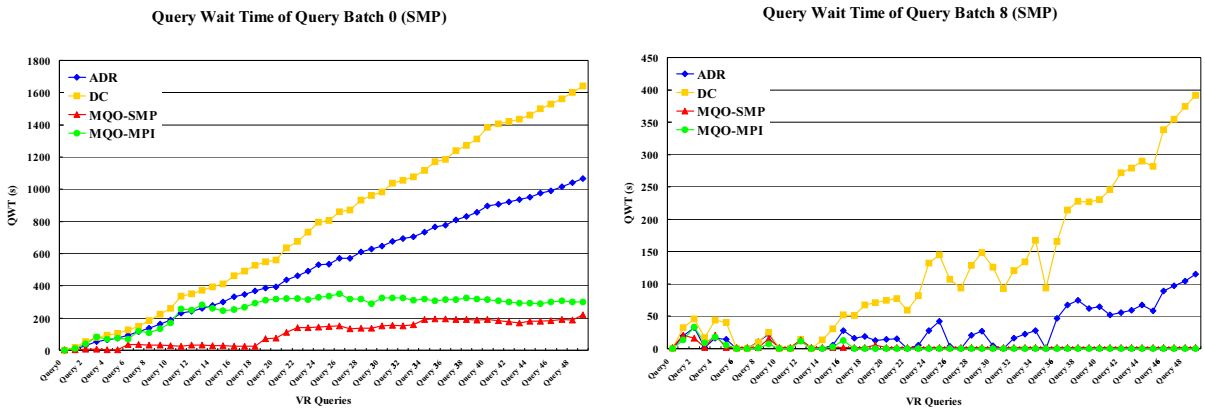
**Figure 3. Query Execution Time in the SMP environment.**




(a) Median (b) Standard Deviation

**Figure 4. Query Wait+Execution Time in the SMP environment.**

workload generator tool was used to submit batch queries to the parallel back-end of ADR and receive the results. The standard ADR configuration has front-end and parallel back-end systems that are customizable by an application developer. The workload generator serves as a simplified version of the ADR front-end. The parallel back-end consists of 16 different processes that share the same code. Each process is responsible for retrieving images that overlap the VR query range, which are stored in two different data files, and the process produces partial occupancy maps for each frame specified for a query in the ADR Local Combine step. For the ADR Global Combine step, all of the partial occupancy maps for each frame are globally combined across the processors via inter-process communication, resulting in complete occupancy maps that represent the 3D volume occupied by the foreground objects in the images. The process responsible for this global merging process is determined by ADR in round-robin fashion for each frame. Therefore the work for the Global Combine step across all the occupancy maps is allocated uniformly across the processes. ADR uses the MPICH library for inter-process communication. We configured the low-level communication device of the MPICH library as *ch_shmem*, which is appropriate for a single shared memory system. The often used MPICH *ch_p4* device does not perform as well as the shared memory device, because it employs network seman-




(a) Query Batch 0

(b) Query Batch 8

**Figure 5. Query Wait Time in the SMP environment.**

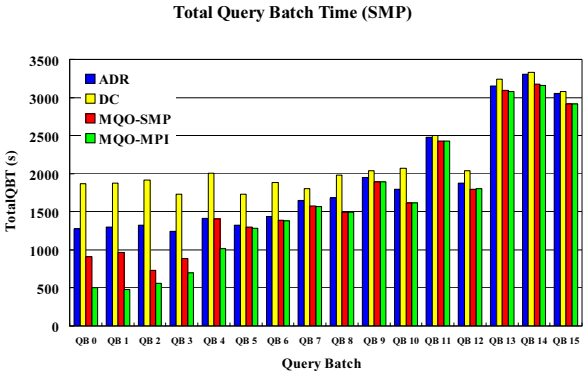



**Figure 6. Total Query Batch Time in the SMP environment.**

tics even for processes that share physical memory in the SMP. Using shared memory for MPI interprocess communication performs much better in the SMP environment since it can substantially reduce communication time through optimized message passing through shared memory (as confirmed during our trial runs).

In the DC framework, all the VR filters are executed as *application daemon* (`appd`) threads. Since we are using only one SMP machine in this environment, we deploy only one `appd`. Therefore, all of the filter instances are threads belonging to a single `appd` process. This differentiates DC from ADR. That is, we can guarantee that ADR uses only 16 processors for back-end processing. However, in DC additional threads may be instantiated dynamically. In order to ensure that the same amount of computational resources are employed in all the VR implementations, we used the Solaris utility *psrset*. This command enables the creation of a fixed *processor set* and the binding of an application to this set, thus ensuring that no additional processors are used (note that our SMP machine has 24 processors). Therefore, we created a processor set that contains 16 processors and bound the DC `appd` to it. DC is configured with 2 ImageReader filters, 13 LocalCombiners, 1 GlobalCombiner and a separate `Console` process. This configuration proved to be the best one we could find during our trial runs. The `Console` acts as the workload generator, similar to the front-end in ADR. The `Console` runs as a separate process and submits batch queries and receives the results from the filters. We determined that using more than two ImageReader filters decreases the overall performance of the DC implementation, because of resource contention from multiple threads attempting to read image data from the RAID disks simultaneously.

We employed two configurations of the MQO system, an SMP version and and an MPI version, to see how different mechanisms for processing multiple queries can affect the overall performance of the MQO implementation. In the SMP version of the MQO framework, each query is executed as one thread (using the Pthreads library). MQO's `Query Server` is implemented as a fixed-size thread pool that interacts with the clients to receive queries and return the results. Usually the size of this thread pool is the number of processors available in SMP system, i.e, it is the limit on the maximum number of queries that can be simultaneously serviced. However, we configured MQO's query server such that it uses a 16-thread pool since we wanted to ensure that it relies on the same resources as ADR and DC.

The MPI version of the MQO framework was originally developed for clusters of SMPs. In MQO-SMP, all queries are executed as different threads, i.e., using *inter-query* parallelism. However, MQO-MPI processes one query with multiple processes, i.e., using *intra-query* parallelism. Therefore, in a sense, MQO-SMP is similar to DC in the SMP environment since all queries are executed as threads in one server process. Similarly, MQO-MPI and ADR have in common that they use multiple server processes to execute a query. We will see from the experiment results that in the SMP environment, MQO-MPI outperforms MQO-SMP in many cases.

Both versions of the MQO system employ a fixed amount of space for semantic caching. In our experiments, this amount was fixed at 1 GB. In the SMP version, there is a common pool of 1 GB and, in the MPI version, 1 GB is uniformly split among the 16 processes, i.e., each one has 64 MB for caching. The size of the reconstructed volume for each frame in the queries in the experimental batches requires 256 KB of storage and, therefore, a 30-frame query requires 7.5 MB. It is easy to verify that 1 GB of cache space is enough to store the reconstructed volume for 4096 frames (which implies potentially caching all of the frames in our experimental dataset). This analysis allows us to estimate an upper bound for the estimated decrease in batch execution time that can be achieved by either version of MQO. Looking at Table 1, we see that for all of the batches the number of frames that undergo volumetric reconstruction is around 1500, however only about 400 of those are unique. Therefore, MQO could ideally execute a query batch in approximately 26% of the time required by either DC or ADR, since its cache space is large enough to store all reconstructed frames.

Figures 3, 4, 5, and 6 depict the performance of the various implementations of the VR application in the SMP environment. In these charts, we see that the two variants of MQO perform better than both DC and ADR. Interestingly, as seen in Figure 3(a), MQO-SMP has the worst performance in terms of the observed QET for batches QB0, QB1 QB2, and QB3. This is explained by how queries are processed by the query server, i.e., queries are usually allocated to a single thread and because of the small query interarrival time, multiple queries, in particular, the ones for which reuse can be leveraged will block, while the reusable results are computed. In other words, the system infers that reuse is possible, but the reusable result is still being computed and this triggers a wait period for all queries already in the system whose results are going to be computed based on previously reconstructed frames. As the system becomes less busy, results can be immediately reused and queries are computed more quickly. Note that MQO-SMP may assign more than one thread for a query, if there are no other waiting queries in its scheduling queue. However, this

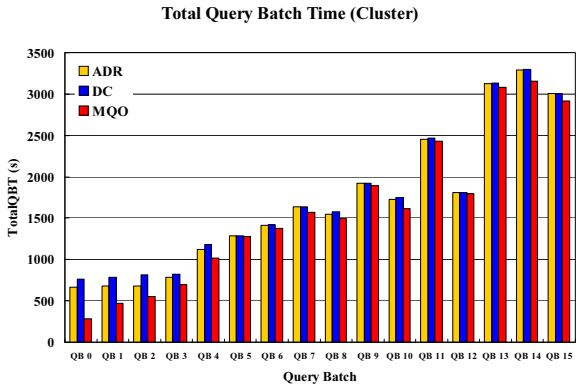
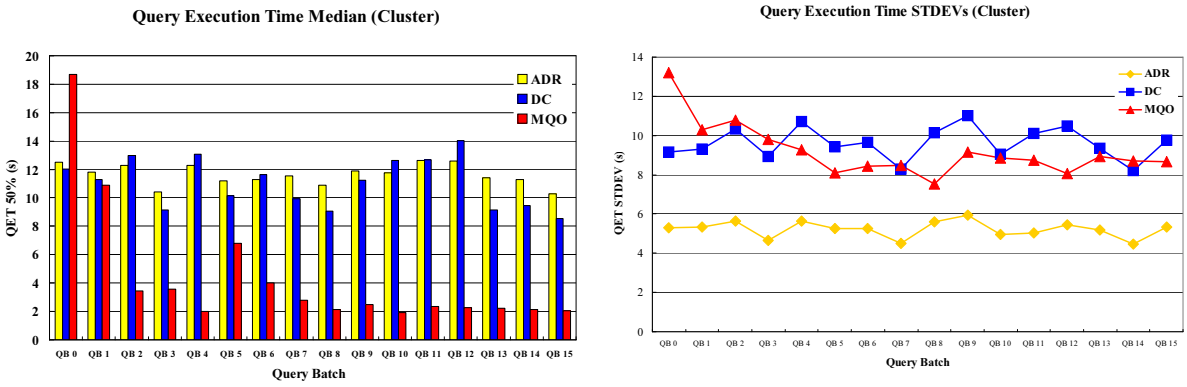



**Figure 10. Total Query Batch Time in the cluster environment.**

strategy provides performance improvements if the system is not overloaded. Because of MQO-SMP's adaptive behavior, the standard deviation for batch execution time depicted in 3(b) is consistently much higher (but decreasing as the system becomes less loaded) than for the other implementations where queries are processed in parallel, but one at a time. It is also evident in this picture that ADR, by employing asynchronous I/O operations, shows very low variability in the QET metric.

For QB0, Figure 5 shows that MQO-SMP incurs the least amount of delay between the submission of a query and when it is scheduled for execution, since queries are assigned to as little as one processor (thread). In fact, a query may execute sequentially with other queries using other processors. The QWT slope for ADR and DC shows that those implementations are not able to keep up with the incoming queries, as the wait time increases for queries submitted later.

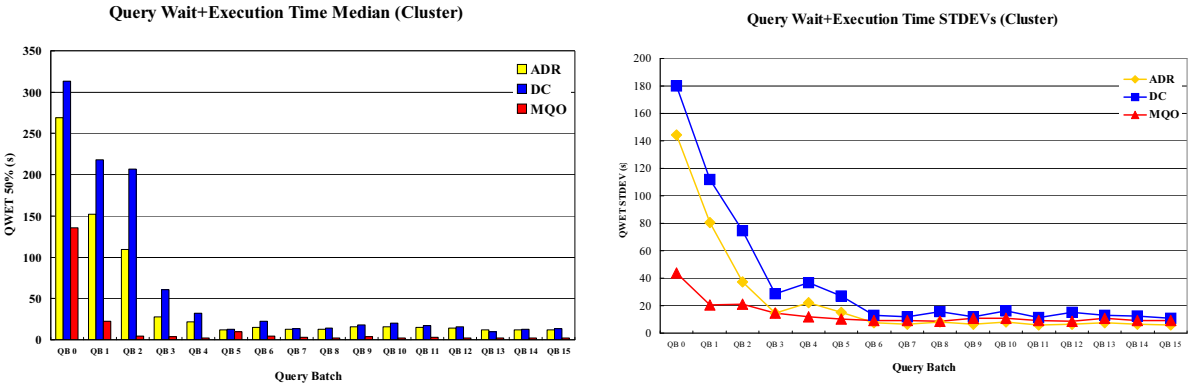
Analyzing the performance of the different implementations using the combined QWT and QET metrics, i.e., the *Query Wait and Execution Time* (QWET), provides us with a user perspective of how long it takes to actually execute the queries. This metric is particularly important in interactive systems. One interesting point is seen when Figure 3(a) is contrasted with Figure 4(a), in particular for QB0 (i.e., when the systems are subjected to the the highest workload), MQO-SMP and MQO-MPI behave quite differently. While, MQO-SMP shows a longer QET on average, it makes up for that in terms of of a much lower QWT on average, which results in faster response time (QWET). Although that advantage does not appear consistently, even for batches QB1, QB2, and QB3, MQO-SMP is clearly superior in terms of reducing QWT as it can execute more than one query simultaneously. From the QWET metric, we see




(a) Median

(b) Standard Deviation

**Figure 7. Query Execution Time in the cluster environment.**



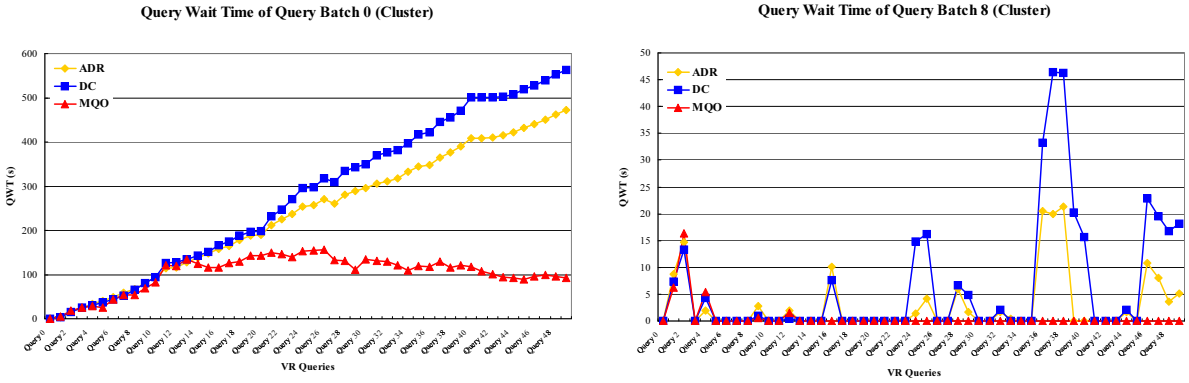

(a) Median

(b) Standard Deviation

**Figure 8. Query Wait+Execution Time in the cluster environment.**

that both MQO systems perform much better than ADR and DC, show the large benefits of optimizing for reuse when there is locality and the systems are subjected to intense workloads. Those performance improvements are less obvious as the system becomes less busy (QB7 to QB15), even though a great deal of locality still occurs across the queries.

Another interesting result is seen in Figure 6. When the system is under a great deal of stress (QB0 to QB4), the multi-query optimization strategies are most beneficial. That is, any savings in resource usage observed for a query immediately translates into decreased response time for other queries in the system. It is also interesting to observe that MQO-MPI is clearly superior to MQO-SMP for those query batches. The primary reason is that MQO-SMP processes several queries at the same time. This causes internal competition for the I/O subsystem in the SMP, as low locality is achieved in disk operations. MQO-MPI is more disciplined in that regard because only a single query is being processed at a time. Another observation is that MQO-MPI is able to process QB0 in around 35% of the time required by ADR. A back of the envelope calculation shows that the lower bound in terms of batch execution time is 26% (400 unique frames out of a total of 1540, as seen in Table 1). Therefore, we can estimate MQO's overhead, i.e., the functionality required to support multi-query optimization, as around 10%. Figure 6 also shows that as the system be-




(a) Query Batch 0

(b) Query Batch 8

**Figure 9. Query Wait Time in the cluster environment.**

comes less busy (QB5 to QB15), reuse does not improve the MQO implementations' performance in terms of total query batch time, despite the fact that it still decreases the execution time of individual queries (because of the large inter-arrival times of the queries)..

## 5.4 Cluster Environment

In the cluster environment the input dataset was declustered across multiple local disks. We employed a declustering strategy based on Hilbert space-filling curves [31], which has been shown to result in close to optimal workload balance for Volumetric Reconstruction queries on contiguous regions in the multi-dimensional space [14]. We used 16 nodes for storing the dataset files. Each node hosted two dataset partitions stored on two local hard disks, since having one partition per disk enables better I/O performance.

ADR employed a total of 17 nodes for the experiments, one for the workload generator and 16 for its parallel back-end nodes, each of which stores the two data files. We used the generic *ch_p4* MPICH device in this environment. We measured the execution times of queries in ADR as the *server-side computation time*. Because ADR (and MQO) uses MPI for interprocess communication, there is an overhead from MPI initialization time that does not exist for the DC implementation, which uses TCP/IP communication between multiple `appds`. To ensure a fair comparison, we inserted timing code into the ADR server code, and measured the processing time in the parallel back-end system. Since a query is executed in parallel across multiple cluster processes (one per processor), we used the maximum server-side execution time across the all processes as the query execution time.

As does the ADR implementation, the DC implementation reserves one node for running its console process. In the shared memory environment, we used only one DC `appd` since the application runs on single SMP machine. In the cluster environment, we deployed 16 `appds` across the 16 available processors, each of which is assigned one ImageReader filter and one LocalCombiner filter. Since we want to use exactly the same resources as for ADR, we placed the GlobalCombiner filter on one of the 16 processors (appds). Hence, the DC implementation uses 16 ImageReaders, 16 LocalCombiners and 1 GlobalCombiner resulting in a total of 33 filters. To exploit effective dynamic scheduling of filters, we used DC's demand-driven policy to connect the streams between the transparent copies of the ImageReader and LocalCombiner filters. That policy ensures that the runtime system checks the status of the LocalCombiners on all nodes and directs the output of an ImageReader copy to the LocalCombiner copy with the lightest current load. Because the LocalCombiners perform the most compute-intensive tasks in the system, a good load balancing of all the workload in the system is imperative. The demand-driven policy achieves this goal better than the default round-robin policy.

We employed the MPI version of the MQO system on the cluster, since that is the only version that can run in the cluster environment. We deployed one MQO process onto each of the 16 cluster nodes used for the server, and used the other cluster node for the workload generator. MQO was configured with one worker thread on each node, since each node has only one processor. That implies that each node can only execute one query at a time, and since each query must be executed on all server nodes (to access the data on all nodes).

The results in Figures 7, 8, 9, and 10 show trends that are very similar to the SMP experimental results. One interesting observation is that the multi-query optimization strategies are not as important from the standpoint of total batch query execution time as in the SMP environment. As seen in Figure 8, only QB0, QB1 and QB2 show substantial differences in performance among MQO, ADR and DC. The processors in the cluster are considerably faster than the SMP processors and the I/O workload is spread across 32 disks. Therefore, the systems do not appear to be as busy as in the SMP, which becomes evident when contrasting Figure 9(b) to Figure 5(b). However, reuse is still important when the QWET metric is used to compare the three implementations.

MQO can also be configured to generate non-optimized query plans, i.e., no reuse is enabled (but the reuse infrastructure is still in place, so some overhead still occurs). Although the results are not shown in the graphs, we ran the same experiments using the unoptimized MQO configuration in order to capture the overhead caused by the reuse infrastructure. Our results show that both ADR and DC outperform MQO in such a situation. That is, for scenarios in which there is low locality in the queries, both ADR and DC will produce better overall system throughput and user response time than the unoptimized MQO.

Finally, considering all the experimental evidence we have gathered, we observed that in both the SMP and cluster environments, ADR tends to outperform DC. In our experimental environments, DC incurs the overhead of data copies as data flows from one filter to the next. When we compare these two systems employing the TotalQBT metric, the difference can be substantial. For example, ADR runs in around 68% of the time required for DC for Query Batch 0 in the SMP environment. It should be noted, however, that the DC architecture was tailored for heterogeneous environments, where applications are functionally decomposed and different components may execute on different machines. Therefore, it is interesting to see that as far as TotalQBT is concerned, when the query server is moderately loaded, all three middleware system implementations exhibit similar throughput.

### 5.5 Lessons Learned

We present a short summary of key points from what we have learned from the experimental study:

1. Workload characterization is very important in designing a middleware system. The impact caused by different optimizations relies primarily on variables such as the expected inter-arrival time of queries and the amount of locality in the workload.

2. Carefully scheduling and coordinating the utilization of I/O resources plays an important role, as competition for resources among multiple entities (threads, processes, etc.) can negatively impact system performance.

3. Reuse is an important optimization, but comes at a cost that can be substantial in some cases. Moderately loaded systems do not appear to substantially benefit from reuse, and when there is poor locality in the workload reuse strategies become a liability.

4. Resource adaptation is paramount. ADR achieves adaptation by employing asynchronous I/O, DC achieves adaptation by dynamically load balancing across filter copies, and MQO adapts its query execution strategy from essentially sequential for a single query (i.e., executed within a single thread/processor) to parallel for one query (i.e., multiple threads/processors) as the overall MQO system becomes less loaded. Employing all of these resource adaptation techniques at the same time and dynamically modifying query server behavior can potentially yield greater performance in comparison to any one of the three existing systems, at the expense of more architectural complexity in the middleware.

## 6 Conclusions and Future Work

Any experimental study such as the one conducted in this paper has limitations. Although we have attempted to study the middleware systems using different workloads and runtime environments, many aspects of such a study are closely tied to the application characteristics and system configuration. Our major aim was to present trends and, based on them, suggest improvements to data analysis middleware systems. One limitation of the present study is that in both computational environments we employed, the parallel resources are homogeneous. However, when heterogeneity is an issue (and, in Grid environments that is becoming prevalent), DC is expected to outperform both ADR and MQO, because of its ability to adapt to the current runtime environment. We intend to address the issue of heterogeneity in resources (processor, network, etc.) in a future study.

We have shown that leveraging reuse across queries and designing middleware for multi-query scenarios can provide large performance improvements. On the other hand, there are overheads associated with such an architecture. Therefore, a system that can change its behavior based on the workload offered should be able to provide increased performance both in system throughput and in user response time. Coordination across multiple threads in performing I/O operations is also very important. With the exception of ADR, which carefully plans its I/O operations, neither DC nor MQO provide an internal mechanism for optimizing I/O

operations, and performance is degraded by lack of support for I/O coordination.

We have shown that being adaptive to runtime constraints and to workload characteristics is important for an SMP environment as well as for a cluster environment. With the advance of Grid technologies, adaptation will play an even more important role for designing high-performance, truly distributed applications. We intend to investigate mechanisms for automatic and quick adaptation to changes in resource availability and user's demands on middleware systems.

## References


[1] G. Agrawal, A. Sussman, and J. Saltz. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):747–754, July 1995.

[2] H. Andrade. *Multiple Query Optimization Support for Data Analysis Applications*. PhD thesis, Department of Computer Science, University of Maryland, December 2002.

[3] 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 Conference (SC 2001)*, Denver, CO, November 2001.

[4] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Active Proxy-G: Optimizing the query execution process in the Grid. In *Proceedings of the 2002 ACM/IEEE SC Conference (SC 2002)*, Baltimore, MD, November 2002.

[5] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Scheduling multiple data visualization query workloads on a shared memory machine. In *Proceedings of the 2002 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002)*, Fort Lauderdale, FL, April 2002.

[6] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Exploiting functional decomposition for efficient parallel processing of multiple data analysis queries. In *Proceedings of the 2003 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003)*, Nice, France, April 2003.

[7] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Optimizing the execution of multiple data analysis queries on parallel and distributed environments. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):520–532, June 2004.

[8] Association for Pathology Informatics. *http://www.pathologyinformatics.org*.

[9] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The $R^*$-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM International Conference on Management of Data (SIGMOD 1990)*, pages 322–331, May 1990.

[10] M. Beynon, C. Chang, U. Catalyurek, T. Kurc, A. Sussman, H. Andrade, R. Ferreira, and J. Saltz. Processing large-scale multidimensional data in parallel and distributed environments. *Parallel Computing*, 28(5):827–859, May 2002. Special issue on Data Intensive Computing.

[11] M. 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, October 2001.

[12] E. Borovikov and L. Davis. A distributed system for real-time volume reconstruction. In *Proc. of Computer Architectures for Machine Perception*. IEEE Computer Society, September 2000.

[13] E. Borovikov, A. Sussman, and L. Davis. An efficient system for multi-perspective imaging and volumetric shape analysis. In *Proceedings of Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia*, April 2001.

[14] E. Borovikov, A. Sussman, and L. Davis. A high performance multi-perspective vision studio. In *Proceedings of the 2003 ACM International Conference on Supercomputing (ICS 2003)*, pages 348–357, San Francisco, CA, June 2003.

[15] A. Bottino and A. Laurentini. A silhouette-based technique for the reconstruction of human movement. *Computer Vision and Image Understanding*, 83, 2001.

[16] D. L. Brown, W. D. Henshaw, and D. J. Quinlan. Overture: Object-oriented tools for applications with complex geometry. In *Proceedings of the 1999 International Conference on Scientific Computing in Object-Oriented Parallel Environments (ISCOPE 1999)*, pages 96–107, San Francisco, CA, December 1999.

[17] M. Cavazza, R. Earnshaw, N. Magnenat-Thalmann, and D. Thalmann. Motion control of virtual humans. *IEEE Computer Graphics and Application*, 18(5):24–31, 1998.

[18] C. Chang, A. Sussman, and J. Saltz. CHAOS++. In G. V. Wilson and P. Lu, editors, *Parallel Programming Using C++*, Scientific and Engineering Computation Series, chapter 4, pages 131–174. MIT Press, 1996.

[19] Y. E. Cho, M. Winslett, S. wen Kuo, and Y. Chen. Parallel I/O for scientific applications on heterogeneous clusters: A resource-utilization approach. In *Proceedings of the 1999 International Conference on Supercomputing*, pages 253–259. ACM Press, June 1999.

[20] L. Davis, E. Borovikov, R. Cutler, D. Harwood, and T. Horprasert. Multi-perspective analysis of human action. In *Proceedings of Third International Workshop on Cooperative Distributed Vision*, November 19-20, 1999.

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

[22] S. Fink, S. Kohn, and S. Baden. Efficient run-time support for irregular block-structured applications. *Journal of Parallel and Distributed Computing*, 50(1):61–82, April 1998.

[23] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[24] High Performance Fortran Forum. High Performance Fortran – language specification – version 2.0. Technical report, Rice University, January 1997. Available at http://www.netlib.org/hpf.

[25] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluating compiler optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21(1):27–45, April 1994.

[26] W. Humphrey, S. Karmesin, F. Bassetti, and J. Reynders. Optimization of data-parallel field expressions in the POOMA framework. In *Proceedings of the 1997 International Conference on Scientific Computing in Object-Oriented Parallel Environments (ISCOPE 1997)*, pages 185–194, Marina del Rey, CA, December 1997.

[27] Y.-S. Hwang, B. Moon, S. D. Sharma, R. Ponnusamy, R. Das, and J. H. Saltz. Runtime and language support for compiling adaptive irregular programs. *Software–Practice and Experience*, 25(6):597–621, June 1995.

[28] D. A. Keim and H.-P. Kriegel. VisDB: A system for visualizing large databases. In *Proceedings of the 1995 ACM International Conference on Management of Data (SIGMOD 1995)*, page 482, San Jose, CA, May 1995.

[29] 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 Conference (SC 1999)*, November 1999.

[30] J. López and D. O'Hallaron. Evaluation of a resource selection mechanism for complex network services. In *Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing (HPDC 2001)*, pages 171–180. IEEE Computer Society Press, August 2001.

[31] B. Moon, H. V. Jagadish, C. Faloutsos, and J. Saltz. Analysis of the clustering properties of the Hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, January/February 2001.

[32] E.-J. Ong and S. Gong. Tracking hybrid 2D-3D human models from multiple views. In *Proceedings of the IEEE International Workshop on Modeling People*. IEEE Computer Society Press, Los Alamitos, Calif., 1998.

[33] OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface – Version 2.0 March 2002*, March 2002. *Available at http://www.openmp.org*.

[34] H. Saito, S. Baba, M. Kimura, S. Vedula, and T. Kanade. Appearance-based virtual view generation of temporally-varying events from multi-camera images in the 3D room. Technical Report CMUCS99127, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, April 1999.

[35] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI–The Complete Reference, Second Edition*. Scientific and Engineering Computation Series. MIT Press, 1998.

[36] M. Spencer, R. Ferreira, M. Beynon, T. Kurc, U. Catalyurek, A. Sussman, and J. Saltz. Executing multiple pipelined data analysis operations in the Grid. In *Proceedings of the 2002 ACM/IEEE SC Conference (SC 2002)*, Baltimore, MD, November 2002.

[37] S. Stillman, R. Tanawongsuwan, and I. Essa. A system for tracking and recognizing multiple people with multiple cameras. In *Proc. Second Int'l Conf. Audio- and Video-based Biometric Person Authentication*, pages 96–101, 1999.

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

[39] D. Thompson, J. Braun, and R. Ford. *OpenDX: Paths to Visualization*. Vis, Inc, 2000.

[40] A. Utsumi, H. Mori, J. Ohya, and M. Yachida. Multiple-human tracking using multiple cameras. In *Proc. of the third IEEE Int'l Conf. Automatic Face and Gesture Recognition (Nara, Japan)*. IEEE Computer Society Press, Los Alamitos, Calif., 1998.

[41] S. X. Yang, D. Gannon, P. Beckman, J. Gotwals, and N. Sundaresan. pC++. In G. V. Wilson and P. Lu, editors, *Parallel Programming Using C++*, Scientific and Engineering Computation, chapter 13, pages 507–546. MIT Press, 1996.