Active Proxy-G: Optimizing the Query Execution Process in the Grid*

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.3}@osu.edu

Abstract

The Grid environment facilitates collaborative work and allows many users to query and process data over geographically dispersed data repositories. Over the past several years, there has been a growing interest in developing applications that interactively analyze datasets, potentially in a collaborative setting. We describe the Active Proxy-G service that is able to cache query results, use those results for answering new incoming queries, generate subqueries for the parts of a query that cannot be produced from the cache, and submit the subqueries for final processing at application servers that store the raw datasets. We present an experimental evaluation to illustrate the effects of various design tradeoffs. We also show the benefits that two real applications gain from using the middleware.

1 Introduction

The Grid is an ideal environment for running applications that need extensive computational and storage resources. Most research in high-end and grid computing has focused on the development of methods for solving challenging compute or data intensive applications in science, engineering, and medicine. A salient feature of the Grid is that it fosters collaborative research and facilitates remote access to shared resources by multiple client applications. There has also been an emerging set of applications that involve interactive analyses of large datasets at geographically dispersed locations. Interactivity and process composition have played important roles in a variety of recent grid computing projects. For instance, the Telescience for Advanced Tomography Applications project (*http://www.npaci.edu/Alpha/telescience.html*) is dedicated to merging technologies for remote control, Grid computing and federated digital libraries. The objective is to connect scientists' desktops to remote instruments, distributed databases, and to data and image analysis programs. The GriPhyN project (*http://www.griphyn.org*) targets storage, cataloging and retrieval of large, measured datasets from large scale physical experiments. The goal is to deliver data products generated from these datasets to physicists across a wide-area network. The objective of the Earth Systems Grid (ESG) project (*http://www.earthsystemgrid.org*) is to provide Grid technologies for storage, publication, and analysis of large scale datasets arising from climate modeling applications.

In multi-client environments, we expect data reuse across queries. Multiple users are likely to want to explore the same portion of a dataset (usually some portions of datasets tend to be of particular interest). There may also be commonalities in a sequence of queries that look at the same physical region at different points in time. When rapid response is needed, performance can be improved by reusing previously cached results for a new query. In the context of Web services, data caching and Web proxies have been shown to speed up servicing web requests by caching popular pages, only performing remote transactions when requests cannot be satisfied from the cache [10, 20].

A Grid-based environment, which consists of a collection of compute, memory, and storage systems (i.e., shared- and distributed-memory parallel machines, high-end I/O systems, and active disk-based storage systems), offers a powerful and flexible environment for developing and deploying applications that analyze large datasets. Component-based frameworks and services have been gaining acceptance as a viable approach for application development and execution in distributed environments [1, 4, 11, 13, 14, 18, 23, 24, 27, 29]. Such models facilitate the implementation of applica-

*This research was supported by the National Science Foundation under Grants #EIA-0121161, #EIA-0121177, #ACI-9619020 (UC Subcontract #10152408), #ACI-0130437, and #ACI-9982087, and Lawrence Livermore National Laboratory under Grants #B500288 and #B517095 (UC Subcontract #10184497).

0-7695-1524-X/02 $17.00 ©2002 IEEE

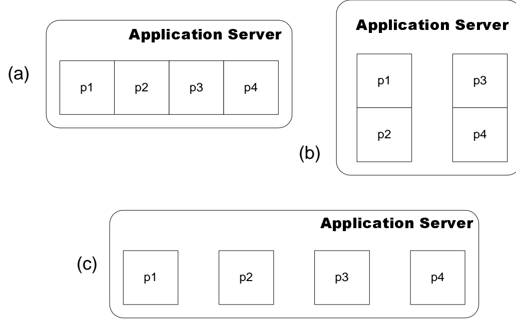


Figure 1. An application server may use many different parallel configurations depending on what is most efficient for an application. (a) shared memory, (b) distributed shared memory, or (c) distributed memory.

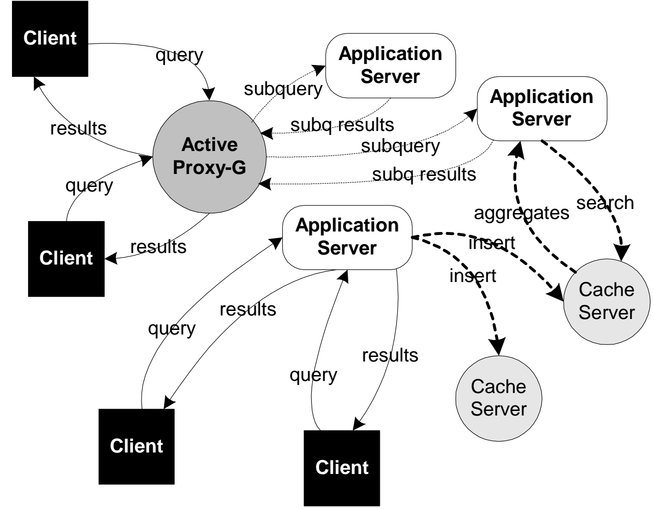


Figure 2. The suite of services, which consists of the Active Proxy-G, application query servers, and persistent cache servers, for optimizing the execution of multiple query workloads in a Grid-based environment.

tions and services that can accommodate the heterogeneous and dynamic nature of the Grid.

In previous work [7, 8], we have developed a framework for efficiently executing multiple query workloads from data analysis applications on SMP machines and clusters of distributed-memory parallel machines. In this work, building on that framework, we develop a component-based framework designed as a suite of services. This suite consists of an active proxy service, an application query processing service (application server) shown in Figure 1, and a persistent data caching service (cache server) seen in Figure 2. Here we focus on the design and implementation of the **Active Proxy-G** (APG) service. Employing the APG requires adding extra functionality to the original application structure. First, the APG must attempt to service the request solely from cached results. If that fails completely, then the request must be sent to the appropriate application server that has access to the raw data required to answer the request. As a last step, if the request can be only partially answered from the cache, the application must retrieve the cached results and generate subqueries to produce the remaining results at the application server.

2 Related Work

In designing APG, several research aspects were taken into consideration to make it a suitable platform for optimizing the execution of queries in a grid environment. Many of these aspects have been studied before, but the novelty of our approach lies in integrating several pieces in a common framework which is able to process queries with user-defined operators for a class of data analysis applications as will be seen in Section 3.

Rodríguez-Martínez and Roussopoulos [30] proposed a database middleware (MOCHA) designed to interconnect distributed data sources. MOCHA also handles queries with user-defined operators. The system handles *data reduction* operators by *code-shipping*, which moves the code required to process the query to the location where the data resides, and *data inflation* operators by *data-shipping*, which moves the input data to the client. This differs from our approach in that we specifically target multiple query optimization and use caching to avoid network bottlenecks, and also utilize the processing power of proxies, distributing the computation transparently across available resources between a proxy and the application servers. This allows efficient execution of queries for which neither code-shipping nor data-shipping is the best solution alone.

Several highly distributed applications have employed proxy servers as a means for improving system performance. Beynon et al. [15] proposed a proxy-based infrastructure for handling data intensive applications. The authors have shown that their approach can both reduce the utilization of wide-area network connections, reduce query response time, and improve system scalability. Squid [34] is a widely used web proxy that is primarily responsible for caching web pages and thus avoid the latency and server overhead incurred in retrieving pages that were recently visited. The objectives of our framework are similar to these efforts. However, we argue that in addition to being more generic, our framework potentially allows for more powerful performance optimizations because it employs a semantic cache that allows re-using cached results to satisfy re-

quests even when a data transformation function must be applied. Additionally, it employs an extensible query server engine that permits the implementation of multiple applications with different query types.

Many types of environments for executing grid-aware applications can be found in the literature. Wolski et al. [35] describe the *Everyware* toolkit that can be used to enable applications to draw computational power transparently from the Grid. Unlike our work, they target *number crunching* applications. More along the lines of APG is the Distributed Parallel Storage Server (DPSS) [25, 33]. The most important aspect of this approach is the use of parallel operations on distributed servers to supply data streams fast enough to enable the execution of various multi-user, real-time applications in an Internet environment. Bethel et al. [12] show how DPSS is used for building Visapult, a prototype and framework for remote visualization of large datasets. Allcock et al. [3] point out that the data grid infrastructure will need to service thousands of users efficiently, and also highlight that the management of data and replicas is also an important aspect of grid-aware applications. In some sense, our approach is complementary to these efforts because it also enables an application to explore the parallel capabilities of many application servers, but actually goes a step beyond since it allows the proxies to help with the computation by leveraging cached aggregates, and automatically generating subqueries transparently as is shown in Section 4.

Recent efforts in the grid research community [26, 28] are investigating and proposing mechanisms for executing adaptive grid programs – the Grid Application Development Software Project (GrADS) – and support mechanisms for storing metainformation needed to control program execution – the Grid Information System (GIS). Several research projects have investigated the design, implementation, and application of component-based frameworks for application development and deployment [1, 11, 13, 14, 24, 27, 29]. The Common Component Architecture Project (CCA) [17] leads a standardization effort for building distributed software component systems for scientific and engineering applications. The Open Grid Services Architectures (OGSA) effort [21] draws from concepts and technologies that evolved in the Grid and Web worlds to generalize an architecture viable for deploying widely distributed commercial and scientific applications. These initiatives are examples of how our system will have to evolve in order to be integrated into a much larger infrastructure, by being compliant to the models that are going to become protocols, best practices, and, eventually, standards.

Finally, for a discussion of the bulk of our work on the multiple query optimization problem, we refer the reader to our previous papers [6, 7, 9], in which we extensively discuss related research and compare it to the approach we have employed.

3 Query Processing and Data Reuse in Data Analysis Applications

Although many data analysis applications differ greatly in terms of their input datasets and resulting data products, processing of queries for these applications shares common characteristics. Figure 3 shows a pseudo-code representation of the query processing structure, which is commonly referred to as a *generalized reduction operation*.

In the figure, the function $select$ identifies the set of data items in a dataset that intersect the query predicate M_i for a query q_i. In many scientific data analysis applications, both input and output datasets can be represented in a multidimensional space, and, in this case, M_i describes a range query. That is, only the data items whose coordinates fall inside the range bounds are retrieved. The data items retrieved from the storage system are mapped to the corresponding output (or accumulator) items (line 6). Application-specific aggregation operations are executed on all the input items that map to the same output item (lines 7 and 8)[1] using an accumulator data structure which is a user-defined data structure to maintain intermediate results. The aggregation operations employed in this loop are *commutative* and *associative*. That is, the output values do not depend on the order input elements are aggregated. To complete the processing, the intermediate results in the accumulator are postprocessed to generate final output values (lines 9 and 10). In many data analysis applications, the most computationally expensive part of the loop is the reduction phase (lines 4–8).

The characteristics of this generalized reduction loop make it possible to develop common programming and runtime support for a wide range of applications and to implement optimizations for processing of both single and multiple queries. As we noted above, the aggregation operations applied on the input data are commutative and associative. As a result, the input data can be partitioned into data subsets, an intermediate result can be computed from each data subset, and the intermediate results can then be *combined* to create the output data. This property of the query processing loop has two implications for optimizing the execution of queries. First, for a given single query, by partitioning input data into subsets, lines 4 – 8 of the query processing loop can be executed in a parallel or distributed environment. The partitioning of the input data can be done by declustering and storing the input dataset across the machines (or application servers) in the system. Second, the intermediate results (and potentially the output data) can be cached and reused to decrease query execution time, when multiple query workloads are presented to the system. A query q_i may share input elements i_e (line 5), accumulator elements a_e (line 8),

[1] This phase is called the *reduction phase* because the output dataset is usually (but not necessarily) much smaller than the input dataset.

$I \leftarrow$ Input Dataset
$O \leftarrow$ Output
$A \leftarrow$ Accumulator
$M_i \leftarrow$ Query Metainformation
1. $[S_I] \leftarrow Intersect(I, M_i)$
(* Initialization *)
2. **foreach** a_e **in** A **do**
3. $a_e \leftarrow Initialize()$
(* Reduction *)
4. **foreach** i_e **in** S_I **do**
5. **read** i_e
6. $S_A \leftarrow Map(i_e)$
7. **foreach** a_e **in** S_A **do**
8. $a_e \leftarrow Aggregate(i_e, a_e)$
(* Finalization *)
9. **foreach** a_e **in** A **do**
10. $o_e \leftarrow Output(a_e)$

Figure 3. The query processing loop.

and output elements o_e (line 10) with a query q_j. For the portions of query q_i that cannot be answered from cached results, the corresponding data subset can be extracted from the input dataset and an intermediate result can be computed. This intermediate result can then be *combined* with cached intermediate or output results. However, a_e and o_e generated by a query usually cannot be directly reused by another query because they may not exactly match a later request, but require that a data transformation (or *projection*) be applied.

The middleware we describe targets optimized execution of this processing loop via reuse of computed results by in-core and persistent data caching, efficient scheduling of queries for evaluation, and efficient query execution (i.e., Map and $Aggregate$ functions) in a distributed Grid environment.

4 Active Proxy-G

The current design of Active Proxy-G (APG) implements a multi-threaded runtime environment in order to simultaneously handle queries submitted by a large community of users, and also to manage multiple connections with application servers. The APG also performs dynamic workload distribution across multiple application servers, using a scheduling model that employs metrics that depend on the current and past workload of an application server.

To enable an application to use the APG design, it must be structured around an abstract *Query* class, and its related query metainformation class, *QueryMI*. Customization of APG for application-specific queries is achieved by subclassing the provided base classes and implementing a set of virtual methods:

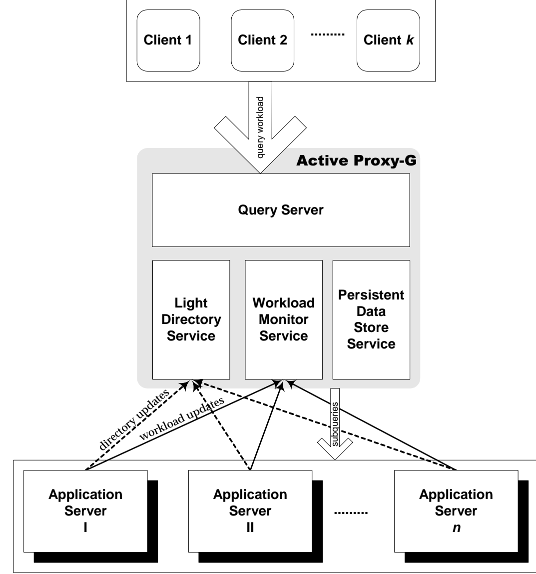


Figure 4. The APG functional components. The directory service maintains information about the location and access methods for application servers and datasets. The workload monitor collects metrics on server performance, and the persistent data store maintains cached aggregates along with their associated semantic tags.

`findOverlaps`: This method queries the cache for intermediate results that may be used completely or partially to satisfy the query being evaluated. It is customized to allow application-specific ways of reusing an intermediate result through one or more data transformation operation. *findOverlaps* returns a list of intermediate results for reuse, which are tagged with semantic information that is later used for applying the correct data transformation operation (i.e. the project method described next). Each returned result also contains an overlap index that tells how much of the cached result can be used. The overlap index serves the purpose of giving the runtime system the opportunity to either reuse the cached intermediate result or recompute it from the input data, in case the projection operation is more expensive than using the input data.

`project`: Given a cached intermediate result, *project* transforms the intermediate result into the output object required by the query being executed. The projection operation may be as simple as a copy operation (when a 100% match has been found), making the

query output data structure point to the cached intermediate result, or much more computationally intensive, as in projecting an intermediate result represented on one multi-dimensional grid to a different grid.

`generateSubQueries`: Given the metainformation describing which parts of the query were answered using the cached results after the project operation, this method generates a list of subqueries to compute the remaining parts of the query.

The proxy runtime system uses the first two methods during its query processing phase, as shown in Algorithm 1. The algorithm uses all possible cached intermediate results to process the query completely without having to resort to retrieve and process the input dataset. The query processing goes through the following phases: a) error checking (line 1), b) cache lookups and memory allocation (lines 2-6), c) projection of the cached results into the query output buffer (line 7), d) generation and processing of subqueries (line 9), and e) remote execution of the query at an application server (line 12), if the query results are still not completely computed. The algorithm `runAndShipResults` is executed for *all* queries, including the subqueries generated from a given query. In that way, subqueries can also benefit from the use of cached aggregates in a recursive fashion.

If a query cannot be fully satisfied from cache (i.e. it needs access to the input dataset(s)), the APG ships the query (or parts of it, as subqueries) to an application server, as is shown in Algorithm 2. The algorithm executes the following steps: a) the Light Directory Service (described later in this section) is consulted to find the appropriate application server(s) to send the query to (line 1), b) the query metainformation is cloned and a subquery is generated for remote execution (lines 2 and 3), c) the query is shipped to the application servers for remote execution (line 4) and, d) the results are retrieved from the application servers and projected to answer the current query (line 6).

We now describe the four major components of the APG:

Algorithm 1 *void* Query::runAndShipResults()

1: **if** $qmi.isOk()$ **then**
2: $\quad stat = findOverlaps(ovlps_list)$
3: $\quad$ **if** $stat \neq COMPLETE$ **then**
4: $\quad\quad allocOutputMemory(ovlps_list)$
5: $\quad\quad allocTmpMemory(ovlps_list)$
6: $\quad$ **if** $stat \neq NO_MATCH$ **then**
7: $\quad\quad project(ovlps_list)$
8: $\quad$ **if** $stat == PARTIALLY_COMPLETE$ **then**
9: $\quad\quad subq_list = generateSubQueries(ovlps_list)$
10: $\quad\quad ret = processSubQueries(subq_list)$
11: $\quad$ **else if** $stat == NO_MATCH$ **then**
12: $\quad\quad ret = executeRemotely()$
13: **else**
14: $\quad ret = METAINFO_ERROR$
15: $sendResults(ret)$

Algorithm 2 *int* Query::executeRemotely()

1: $s = LDS.getServerFor(qmi.getDataName())$
2: $tqmi = qmi.clone()$
3: $tq = createQuery(tqmi)$
4: $tq.submit(s)$
5: $ret = tq.getResults(buf_list)$
6: **if** $ret == QU_OK$ **then**
7: $\quad project(buf_list)$
8: return ret

Query Server (QS). This component is responsible for receiving queries from clients or other APGs. A priority queue is used for storing queries to be scheduled for execution. Queries selected for execution are instantiated by invoking the `runAndShipResults` method in Algorithm 1. The flexibility of this component comes from its ability to easily incorporate new types of queries. Once an application developer provides the customized methods that describe the query metainformation, the data lookup and retrieval, and the data aggregation operation, the query server can handle processing for a new query type.

Light Directory Service (LDS). Once the runtime system determines that a query must be computed from input data, the system must select the application server(s) to use for remotely executing the query. There may be only a single application server that has both the processing capabilities (i.e., it implements the processing functions required by the query) and the input dataset. Alternatively, there may be several application servers that can answer the query. LDS stores information about the location of input datasets and the availability of query processing capabilities for different types of queries. This service provides interfaces for adding new application servers and new input datasets, as well as for registering new types of queries. LDS also has a `getServerFor` method that provides information about the *best* application server to use for a given a query. The best application server is defined in terms of policies implemented by the workload monitor service that we describe next.

Workload Monitor Service (WMS). There are two important aspects to be taken into consideration when many application servers are available: load balancing and fault tolerance. We would like to assign the workload to application servers without overloading any server in particular. Fault tolerance, is important because, in a highly distributed environment, application servers may become unavailable

and later become again available in a reasonably frequent fashion. The runtime system must take into account such environmental changes to better assign queries to servers. To provide these features, WMS continually monitors its registered application servers and collects several metrics related to their query server thread utilization and disk I/O activity. These metrics are used for defining policies to implement the `getServerFor` method for LDS.

Persistent Data Store Service (PDSS). APG efficiency relies heavily on being able to identify reuse opportunities from cached intermediate results. Using a single APG to serve as a proxy for a large community of clients increases the probability of obtaining matches, but the APG must also deal with sets of client requests in which there is little or no temporal locality. Additionally, when many clients for different applications are interacting with an APG, the overall working set in the APG may be quite large, because users may be interested in different *hot spots* over the entire collection of datasets. Although main memory is becoming increasingly cheap, it is still orders of magnitude more expensive than secondary storage. Furthermore, secondary storage can be persistent across APG invocations. PDSS is therefore implemented as a large secondary-storage based Data Store [5]. It is a two-tier hierarchy, using both a portion of main memory plus a large chunk of secondary storage. The employed replacement policies insure that more useful intermediate results are maintained in memory, while other intermediate results may get swapped out to the secondary storage cache [6]. Intermediate results are only purged from the persistent cache when secondary storage space becomes insufficient to hold newly computed intermediate results.

5 Integration with Other Grid Services

The Grid research community has been prolific in developing the software infrastructure, as well as the standard protocols and APIs, for enabling the transparent execution of highly distributed applications. There is a large body of work on middleware services for resource discovery and allocation, file storage and retrieval, security, authentication, and authorization. Our work is complementary in that it focuses on services that target improving the execution of data analysis operations in a collaborative environment. In this paper, we chose to implement in-house versions of various services to speed up the construction of the prototype system. However, our prototype can clearly be integrated with currently available infrastructure to enhance its functionality. Specifically and most strikingly, there are two services in our framework that can be immediately replaced by or interfaced to more powerful equivalents – the Light Directory Service and the Workload Monitor Service.

The Metacomputing Directory Service (MDS) [19] of the Globus toolkit[2] provides functionality for storing and querying information about Grid resources. The MDS architecture is organized around two basic elements: information providers, which are the low-level data collection entities, and aggregate directory services. As far as the functionality provided by LDS is concerned, the LDAP data model used by MDS is capable of storing the information the Active Proxy-G system needs in terms of identifying the application servers with the *correct* processing capabilities for a given query instance. From the standpoint of clients and APGs running in a distributed environment, employing the MDS infrastructure presents a broader advantage in the sense that clients could themselves leverage the directory information to pick the best APG according to some specific qualitative characteristic. Additionally, one APG could, based on directory information, route a query for processing at another APG instead of at an application server.

The Workload Monitoring Service presented in this paper interacts with Application Servers to collect information about the utilization level of the servers. The advantage of having a specialized WMS is that the overall monitoring process can be customized for metrics that are the most critical for a given set of applications. The main drawback of this approach is that load introduced by other applications sharing the same Grid resources are not directly taken into account. The efficiency of the WMS can be improved by interfacing with other Grid monitoring services, such as the Network Weather Service (NWS) [36], to gather more sophisticated information, including overall machine utilization (as opposed to the application server utilization that is currently used), network bandwidth, and network latency. Such information can be used to better determine which application server is the *best* candidate server for a given query. Moreover, addition of customized NWS *sensors* would provide support for collecting some of the metrics used by our scheduling algorithm. In particular, the NWS was designed to be used by dynamic schedulers such as ours, and one of its foremost capabilities is to employ predictors for forecasting how a metric will behave in the near future. Such information would provide the possibility to better schedule the workload based on how a given application server is expected to behave, as opposed to relying only on current behavior as is currently done. In a production Grid environment this capability could certainly be beneficial, since such environments can display much more complex workload behaviors than our simple experimental setup.

The OGSA document [21] discusses how the functionality implemented by the LDS is supposed to be supported by a generic *discovery service*. One of the features of the discovery service is a notification facility that enables clients interested in being warned about particular events to be informed of their occurrence. Such a facility would enable,

[2] *http://www.globus.org*

for example, APG to automatically learn about new application servers as they become available. The OGSA document also describes some *higher-level services* that can be used by WMS. One of them is a set of *instrumentation and monitoring services* that are supposed to be used primarily for ensuring the integrity of the system, but can also be used for better workload scheduling decisions.

Another very important issue that we have not tackled in this work is security and authentication. Employing MDS partially solves this problem if information about the necessary credentials and access control lists are stored. However, the security problem is more complicated than that. An important issue is authentication. Currently, our prototype relies simply on socket communication. No authentication or encryption is employed, which means that any client is able to query the APG or the application servers directly. Reliance on the vanilla UNIX authentication schema is clearly not an option in terms of scalability, because this alternative requires that a client have accounts on all machines hosting the servers. Utilizing the Grid Security Infrastructure (GSI) [22] would solve the scalability problem. More specifically, clients, APGs, cache servers, and application servers must establish a trust relationship. The concept of a *user proxy* is particularly useful since a query will typically use resources available at multiple servers/hosts/domains, and the user proxy would be responsible for acting on behalf of the real user, replacing manual authentication.

This paper has focused on the design and implementation of the APG component of our framework. APG interacts with one or more application servers to answer queries that require input data, and with the Persistent Data Store Service for retrieving cached aggregates. Data access is achieved through a Data Source abstraction [5, 7]. The Data Source presents the underlying storage subsystem as a page-based storage medium. That is, a dataset is assumed to have been partitioned and stored in terms of fixed-size pages. In our current system, we have implemented Data Sources for the Unix file system on disk-based storage. In a Grid environment, the characteristics and interfaces of storage systems can vary widely. The Storage Resource Broker [31] (SRB) provides a unified, Unix-filesystem-like I/O interface to a variety of storage resources. The Application Servers and PDSS can benefit from the SRB infrastructure for storing input and cached data on a heterogeneous collection of storage systems. Moreover, the Metainformation Catalog (MCAT) component of SRB can be used to store metadata information about the datasets controlled by our framework.

As we have seen from this brief discussion, our framework presents a feasible infrastructure for distributing the query execution process across the Grid. The deployment of this system in a Grid environment can leverage multiple available technologies as we have described. In the near future, we plan to make our framework compliant to these still evolving architectures and service standards, which will make it fully integrated into the Grid ecosystem.

6 Applications

We now briefly describe the two applications used as case studies for this paper. A more detailed description of these applications can be found in [6].

6.1 Analysis of Microscopy Data

The Virtual Microscope (VM) [2] is an application designed to support interactive viewing and processing of digitized images of tissue specimens. The raw data for such a system can be captured by digitally scanning collections of full microscope slides at high resolution. A VM query describes a 2-dimensional region in a slide, and the output is a potentially lower resolution image generated by applying a user-defined aggregation operation on high-resolution image chunks.

We have implemented two functions to process high resolution input chunks to produce lower resolution images in VM [9]. Each function results in a different version of VM with very different computational requirements, but similar I/O patterns. 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. Several types of data reuse may occur for queries in the VM application. A new query with a query window that overlaps the bounding box of a previously computed result can reuse the result directly, after clipping it to the new query boundary (if the zoom factors of both queries are the same). Similarly, a lower resolution image needed for a new query can, in some cases, be computed from a higher resolution image generated for a previous query, if the queries cover the same region. In order to detect such reuse opportunities, an overlap function, which is called in the *findOverlaps* method, was implemented to intersect two regions and return an overlap index, which is computed as

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

Here, I_A is the area of intersection between the intermediate result and the query region, O_A is the area of the query region, I_S is the zoom factor used for generating the interme-

diate result, and O_S is the zoom 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.

6.2 Volumetric Reconstruction for Multi-perspective Vision

The availability of commodity hardware and recent advances in vision-based interfaces, virtual reality systems, and more specialized interests in 3D tracking and 3D shape analysis have given rise to multi-perspective vision systems. These are systems with multiple cameras usually spread throughout the perimeter of a room [16, 32]. The cameras shoot a scene over a period of time (a sequence of *frames*) from multiple perspectives and post-processing algorithms are used to develop volumetric representations that can be used for various purposes, including 1) to allow an application to render the volume from an arbitrary viewpoint at any point in time, 2) to enable users to analyze 3D shapes by requesting region-varying resolution in the reconstruction, 3) to create highly accurate three dimensional models of shapes and reflectance properties of three dimensional objects, and 4) to obtain combined shape and action models of complex non-rigid objects.

A query into a multi-perspective image dataset specifies a 3D region in the volume, a frame range, the cameras to use, and the resolution for the reconstruction. The query result is a reconstruction of the foreground object region lying within the query region (a background model is used to remove stationary background objects, but that will not be further discussed in this paper). Formally, a query q_i is described by a query metainformation 5-tuple M_i:

1. a dataset name D_i,
2. a 3-dimensional box B_i: $[x_l, y_l, z_l, x_h, y_h, z_h]$,
3. a set of frames F_i: $[f_{start}, f_{end}, step]$,
4. the depth of the octree (number of edges from the root to the leaf nodes), which specifies the resolution of the reconstruction: d_i, and
5. a set of cameras C_i: $[c_1, c_2, \ldots, c_n]$.

Semantically, a query builds a set of volumetric representations of objects that fall inside the 3-dimensional box – one per frame – using all the available frames for the specified set of cameras. For each frame, the volumetric representation of an object is constructed using the set of images from each of the cameras in C_i. The reconstructed volume is represented by an octree, which is computed to depth d_i. The deeper octrees are, the higher is the resolution for the 3D object representation. Each individual image taken by a camera is stored on disk as a data chunk. A 3-dimensional volume for a single time step is constructed by aggregating the contributions of each image in the same frame for all the cameras in C_i into the output octree. The aggregation operations are commutative and associative.

Our implementation of the Volume Reconstruction (VR) algorithm employs parts of an earlier implementation [16] as a black-box, and that implementation returns an octree for each frame in a sequence of frames. Therefore, the octrees for each frame requested by a query are cached along with the associated metainformation. One potential reuse opportunity is the generation of a lower resolution octree from a higher resolution octree that was computed for an earlier query. In order to detect such possible reuse cases, we implemented the function in Algorithm 3, which provides a customization for the *findOverlaps* method.

Algorithm 3 *float* overlap(M_i,M_j)

```
1: if D_i ≠ D_j then
2:    return 0;
3: v_ovlp ← CommonVolume(B_i,B_j) / Volume(B_j)
4: f_ovlp ← |F_i ∩ F_j| / |F_j|
5: if C_i ⊃ C_j then
6:    c_ovlp ← |C_i| / |C_j|
7: else
8:    c_ovlp ← 0
9: d_ovlp ← 1 − 0.1 × (d_i − d_j)
10: return v_ovlp × f_ovlp × c_ovlp × d_ovlp
```

The transformation of the cached results into results for incoming queries requires the utilization of *project* functions that transform the aggregate appropriately. Algorithm 3 hints at several projection operations: (1) for the *query box* – multiple volumes can be composed to form a new volume, or a larger volume can be cropped to produce a smaller one; (2) for entire *frames* – use the cached frames as necessary; (3) for *cameras* – if the new query requires more cameras than were used for a cached octree, generate a new octree from the images for the new cameras, and merge the two octrees; (4) *depth* – use a deeper octree to generate a shallower one. For the experiments in this paper, we have only implemented the *frames* project function. We will explore the usefulness of the other projection operations in future work.

7 Experimental Evaluation

Many factors affect the performance of the Active Proxy-G. The first set of experiments show results for various sizes of the APG cache, as well as for various levels of multi-threading in the APG service.

A key aspect of our framework lies in the capability of projecting an aggregate into another by performing a data transformation operation. We experimentally show how

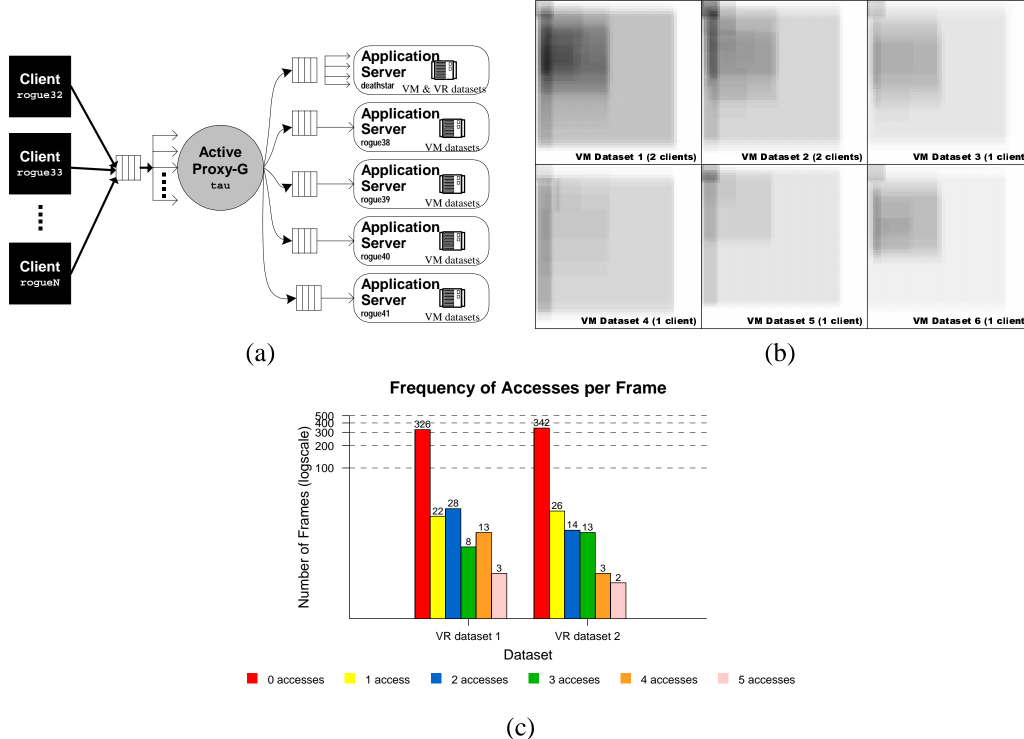


Figure 5. Caching relies on locality amongst the queries submitted to the system. (a) Experimental setup. (b) VM workload - 6 images were queried. Darker regions indicate those that are queried more frequently, implying that important features are located in those regions. (c) VR workload - each datasets contains 400 frames for each of the 13 cameras. The graph shows the popularity of the frames as a function of the number of accesses generated by the set of queries.

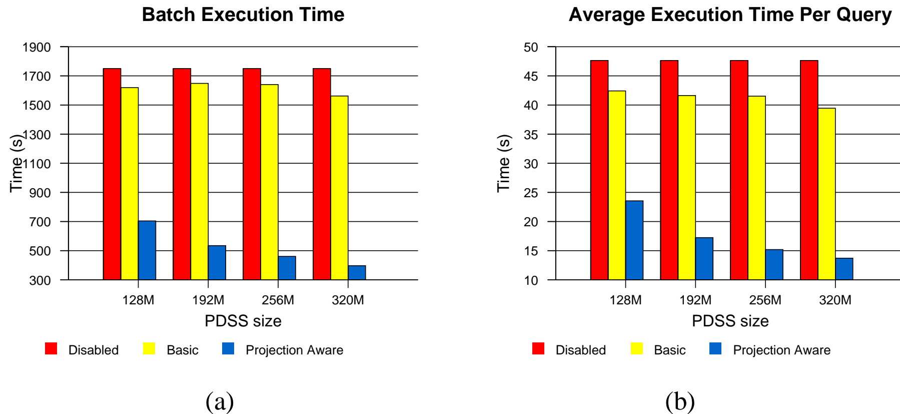
much this capability is able to improve the system performance in the second set of experiments.

Once the APG detects that a query cannot be serviced solely from cached results, it is necessary to generate and forward subqueries to one or more application servers. Multiple application servers may be able to service the query, because the datasets and computational capabilities (in terms of executing queries of a given type) may be replicated at multiple distinct sites. The equitable distribution of queries that are shipped off to the application server is important to achieve good load balancing across all candidate application servers. A third set of experiments will compare variations of load-based scheduling strategies against a round-robin baseline case.

Finally, we investigate and compare several approaches for executing subqueries: 1) they can be sequentially submitted and executed, 2) they can be submitted as a group of concurrent asynchronous subqueries to the application servers, or 3) we can employ an *a priori* partitioning of a query into multiple subqueries. The last two approaches may yield performance benefits, especially in situations where the APG and/or some of the application servers are not heavily loaded. The APG and application server are both multithreaded servers, so under a light workload, a subset of its thread pool may be idle, and therefore the generation of multiple subqueries should be able to employ more of the available computational power that would otherwise be wasted. The last set of experiments evaluates these query execution strategies.

Evaluation of scalability and the size of the PDSS

Two of the most important performance related features in the APG architecture we propose are the level of multithreading employed and the size of the cache for storing reusable aggregates. In order to evaluate the impact of these two variables, we assembled an experimental setup consisting of clients, application servers, and an instance of the Active Proxy-G, as depicted in Figure 5 (a). We have instantiated 5 application servers on 5 nodes in a heterogeneous configuration: $deathstar$ is an 8-processor 550MHz Pentium III Linux SMP machine hosting 8 datasets (two for Volumetric Reconstruction and six for the Virtual Microscope) and $rogue38 \ldots rogue41$ are single-processor 650 MHz Pentium III machines hosting the same six VM datasets as



(a) (b)

Figure 7. Caching strategies. *Basic* **allows reuse of aggregates that are a complete match.** *Projection-Aware* **caching allows reuse of aggregates when partial overlap occurs and data transformation is necessary. (a) Batch execution time. (b) Average execution time per query.**

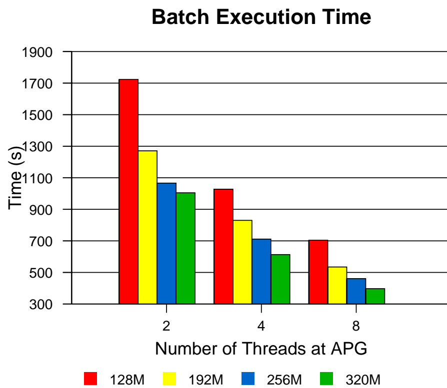


Figure 6. Batch execution time as reported by the APG. We varied the number of simultaneous executed queries by increasing the Query Server thread pool and the amount of cache space available to the Persistent Data Store Service.

$deathstar$. $Deathstar$ had a single application server serving both VM and VR queries, with up to 4 queries under simultaneous execution. The $rogue$ nodes are uniprocessor machines, hence a single query can be serviced at any given time. The APG was hosted on tau, which is a 24-processor UltraSparc III SMP Solaris machine. We employed 12 clients. Four of them generated 16 queries each for volume reconstruction, and the other eight each generated 32 pixel averaging VM queries. The VM clients used a workload model that emulates the behavior of real users as described in [15]. Each VR client submitted queries constructed according to a synthetic workload model (since we do not have real user traces for the application at this time). A query requests a set of volumes associated with frames selected with the following algorithm: the center of the interval is drawn randomly with a uniform distribution from the set of "hot frames", the length of the interval is selected from a normal distribution characterized by mean and standard deviation, and the $step$ value is selected randomly as either 1, 2, or 4. The depth and the 3-dimensional query box were fixed, as was the dataset, and we have used all the available cameras. Each VM dataset is a 10000×10000 3-byte per pixel image, totaling around 1.8GB for the six images. Each VR dataset is a 5200 frame collection (13 cameras, 400 frames), totaling approximately 780MB for the two collections. Figures 5 (b) and (c) give a measure of the amount of spatio-temporal locality in the experimental workload. The figures provide an estimate of the size of the working set for each set of queries, which is the metric that drives how effective the PDSS is as far as improving the system performance.

In Figure 6 we see that increasing the amount of space allocated to the persistent data store greatly decreases the time needed to process the set of 296 queries submitted

to the system[3]. For a fixed number of threads, we see a drop of at least 40% in execution time when PDSS size is increased by 192MB. For a fixed cache size, increasing the multithreading level from two to eight shows a decrease in execution time of up to 67%. The APG was also instrumented to measure the *average overlap ratio*, which is the average fraction of a query that is answered completely from cached results. We observed that the ratio increases from approximately 0.60 up to approximately 0.80 as the PDSS size increases, independent of the multithreading level. However, the average overlap ratio for a fixed PDSS size, when varying the multithreading level, is higher for lower multithreading levels, suggesting that executing more queries simultaneously increases competition for memory. This competition causes potentially useful aggregates to be ejected from the cache. This result shows that using higher multithreading levels also requires increasing the space available to the PDSS, to achieve the same average overlap ratio.

Evaluation of the caching strategies

A novel aspect in our framework is the reuse of cached aggregates by applying transformation (projection) functions. Such reuse is profitable when performing the transformation is less expensive than recomputing the aggregate from input data. The results in Figure 7 compare the performance of *projection-aware* caching with both *no* caching and a *basic* semantic cache implementation, where basic semantic caching only uses a cached aggregate if it *exactly* matches the requested aggregate. The workload is the same as in the previous experiment. The figure shows that basic caching improves batch execution time by around 10% compared to no caching at all, and that increasing the cache size does not significantly improve performance. In particular, the overlap ratio with basic caching remains constant at 0.27, meaning that the whole working set for this caching strategy fits into a 128MB cache. On the other hand, projection-aware caching decreases batch execution time from 56% up to 75% compared to basic caching, showing the benefits of the projection-aware approach. Similar results are also observed when measuring the average query execution time.

Evaluation of the application server assignment policy

As was shown in Figure 2, queries can be shipped to application servers either from the APG or directly from clients. Moreover, variances over time in the load on each of the application servers, as well as in network bandwidth and latency, may be responsible for widely varying query response times. The Workload Monitor Service implements a prototype facility that can help the APG better assign queries to application servers when multiple candidates are available. Although the current implementation of the WMS does not collect metrics for network behavior, it is able to gather metrics for thread pool utilization and disk I/O load at the application servers. This is accomplished by polling each application server periodically (for these experiments, once every 15 seconds). Several individual metrics are collected, but for the purposes of this experiment only two of them are relevant – *thread pool utilization*, referred as TP_u, and *normalized disk read rate* referred as $NDRR$. TP_u shows the percentage of threads in the Query Server thread pool that are busy at the time of polling. $NDRR$ shows what the disk read rate has been since the last poll was taken. $NDRR$ is normalized to a percentage by assuming an *ideal* 30MB/s sustained transfer rate, which is the nominal transfer rate for the disks in our experimental machines. Figure 8 shows the results for a round robin assignment of queries to application servers, and for several variations of the load-based strategies. The combined load l for a given application server is computed as $l = TP_u + NDRR$. Server assignment for a query is determined by finding the server with the smallest load amongst the servers able to service the query, obtained from LDS. In case of ties, round robin is used to select a server. This is a simplified model, that behaves well for sets of queries having similar I/O and computational requirements. Albeit simple, it is reasonably effec-

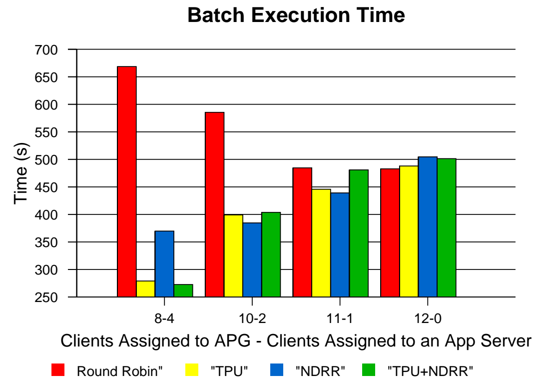


Figure 8. Impact of different application server assignment policy for various workloads and configurations. Under each bar is the workload configuration in terms of how many clients are using the Active Proxy (first number) and how many are interacting directly with an application server (second number).

[3] PDSS uses space both in memory and on disk. The memory space was fixed at 128MB, and the disk portion varied from 0 to 192MB. To minimize the effects of the operating system file buffer cache we used the Solaris `directio` primitive to perform direct disk I/O operations.

tive as the results show.

The workload used to obtain the results in Figure 8 is exactly the same as for the first experimental setup. However, some of the clients interacted directly with an application server and some with the APG. Figure 8 shows performance results, where P–S means P and S clients submitted queries to APG and the application servers (hosted on $rogue38 \ldots rogue41$), respectively. The results in the graph were reported by the APG. Because the number of queries submitted to the APG varied across the configurations due to the varying numbers of clients submitting queries to the APG, the results for different configurations are not directly comparable. Nonetheless, the overall trend is clear - the more unbalanced the assignment is (because of clients submitting directly to the application servers), the more effective the load-based strategies become, since it is better to assign queries to relatively unloaded application servers. Indeed, for the 8-4 configuration, the decrease in the batch execution time compared to the round robin policy is up to 59%. For the 12-0 assignment, the load-based policies achieve slightly lower performance than round robin, for two reasons. First, the system is under a very high workload and any policy that distributes the workload equitably will perform comparably. Second, the assignment for the load-based strategies is slightly more expensive to compute than for round robin. An important observation, is that for greater imbalances (i.e. 8-4), $NDRR$ alone does not provide accurate information about where a given query should be sent for execution. We broke down the execution times for one typical VR query and one VM query (without using the cache), and a VR query spends 52.5% of its execution time on computation and 47.5% on I/O, while a VM query spends 84.3% on computation and 15.7% on I/O. These metrics show why $NDRR$ alone does not provide accurate information about application server load. Since the queries are computationally intensive rather than I/O intensive, TP_u alone was not inaccurate; however for I/O intensive queries it should be. Therefore a combination of the two variables appears to be a good compromise solution. If knowledge about the workload to be presented is available, more accurate strategies can be designed, and, ideally, a self-tuning policy can be devised and implemented.

Evaluation of query execution strategies

An important aspect in the design of a query processing engine is given by the expected load it is supposed to handle, since strategies to optimize for low and high workloads may greatly differ. In a highly distributed environment such as the one we envision for our infrastructure, the load can vary from very light to very intense. All the experimental results seen so far were obtained under severe stress, since we had from 8 up to 12 clients sequentially submitting queries to the APG. In fact, we employed 5 application servers, and there are potentially up to 8 queries being serviced at a given moment (4 $rogue$ nodes in addition to 4 simultaneous queries on $deathstar$), the application server utilization is almost constantly at 100% when the APG is configured for handling 8 simultaneous queries.

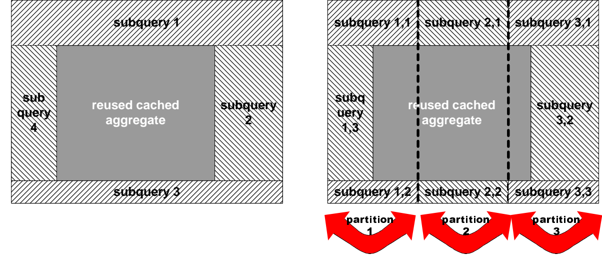
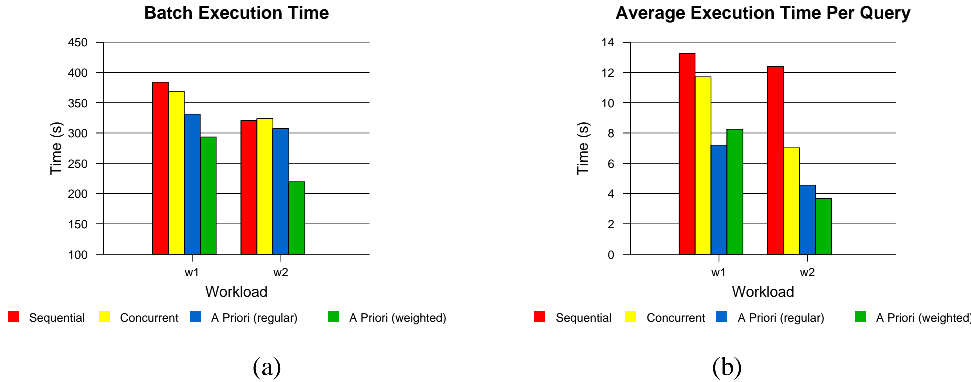


Figure 9. The right diagram shows the automatically generated subqueries which can be potentially concurrently executed by the system. The left diagram shows a query being *a priori* partitioned, and the subqueries generated.

Idle resources under lighter workloads can be better leveraged by making use of more sophisticated query planning and execution strategies. In Figure 10, we present the results for exploring two new query planning and execution strategies, namely *Concurrent* subqueries and *A Priori Partitioning*. *Concurrent* consists of executing the maximum possible number of subqueries generated for the completion of a parent query (as seen in step 9 of Algorithm 3) concurrently as new threads[4]. *A Priori* consists of slicing a query into multiple subqueries when the query is received by the Query Server (Figure 9) and executing them concurrently, if possible. Both strategies are aimed at using idle threads both at the proxy and at the application servers to parallelize the execution of a single query, assuming that many application servers will be able to serve a given query. To evaluate the performance effects of these strategies, we used an experimental setup in which we employed two different workloads $w1$ and $w2$. $w1$ employed 8 clients, 4 submitting 8 VR queries each (using 2 different datasets), and 4 submitting 32 VM queries each (using 2 different datasets). $w2$ employed 4 clients, 2 submitting 8 VR queries each (using 2 different datasets), and 2 submitting 32 VM queries each (using 2 different datasets). The *a priori* partitioning implements a heuristic for limiting how many subqueries to generate. An application specific metric[5] as well as the number of appli-

[4]The use of this strategy does not guarantee that all the subqueries are going to be executed as new threads. That is only the case, if the Query Server has idle threads at the time of execution, otherwise it will fall back to sequential execution.

[5]For VM queries, the number of recommended partitions is computed



(a) (b)

Figure 10. Query execution strategies. Underneath each bar is the workload, where $w1$ denotes a workload of 8 clients, and $w2$ denotes a workload of 4 clients. $Regular$ and $weighted$ denote the query partitioning schema for the *a priori* partitioning strategy (see details in the text) (a) Batch execution time. (b) Average execution time per query.

cation servers that can potentially be used to process a given query (as returned by the Light Directory Service) is used. The smallest number between the two is used as the number of partitions. LDS is able to compute the number of application servers for a given query using two methods. *Regular* in which each application server accounts for one partition, and *weighted* in which each application server gets as many partitions as its maximum multithreading level.

Figures 10 (a) and (b) show the results for the combination of workloads and partitioning strategies in terms of time for executing the whole batch of queries and also for each query on average. Some observations can be made about the results. The concurrent execution of subqueries is responsible for a decrease of 4% for workload $w1$ and an increase of 1% for workload $w2$ for batch execution time. A decrease from 12% up to 43% is observed in the average query execution time per query. The *a priori* partitioning is responsible for a further decrease both in the batch and in the per query metric which can be as large as 14% for batch execution and as large as 63% for an average query when compared to sequential execution for the *regular* variation. The *weighted* variation shows decreases as large as 33% for batch execution and as large as 70% for the average query execution time. As far as the partitioning strategy is concerned, a more aggressive partitioning (weighted) seems to yield a larger decrease in execution time for the batch. However for the larger workload $w1$ that did not happen for the average query execution time per query. And the reason is when more subqueries are generated because of partitioning, better utilization of idle resources is achieved due to higher parallelism, at the expense of more complicated query processing due to an increase in bookkeeping and the use of projection operations. In this case, it incurs higher overheads that cannot be compensated by higher Query Server utilization levels.

so each partition will require at least 4MB of input data. For VR queries, the recommendation will be for as many partitions as the number of frames being requested in the query metainformation.

8 Conclusions

The main goal of this work is to elaborate on an architecture for supporting data analysis applications in the context of highly distributed environments, such as the computational Grid. The design is based on services that can be integrated in different ways to accommodate specific workload scenarios as well as utilization scenarios. In particular, we have extensively evaluated an active proxy configuration that concentrates the workload of multiple clients in such a way that it is able to leverage its own computational ability to respond partially or completely to queries based on aggregates cached locally. This approach results in faster query responses, decreased use of network resources, decreased utilization of possibly remote-located application servers, and effectively better utilization of available resources by partitioning and concurrently executing subqueries. We have also shown quantitatively how each aspect of our design impacts the overall performance of the system. Elsewhere [7] we have shown that an active caching approach is able to increase the performance of a single data analysis application. In this work, we have shown that by making this capability available as a service and incorporating it into an active proxy, a larger community of clients using different applications and datasets can also benefit from more optimized use of resources. In fact, we foresee the APGs being used as Web proxies in the sense that they will be located closer to users where locality is greater, partially shielding them from network latencies and outside disrup-

tions. We also anticipate hierarchical networks of APGs being employed to ensure better scalability.

Acknowledgments. We would like to thank the anonymous reviewers of our paper. Their excellent comments helped us significantly in improving the presentation and content of the paper, specifically towards suggesting ideas on how our prototype can be integrated with the available Grid infrastructure.

References

[1] M. Aeschlimann, P. Dinda, L. Kallivokas, J. López, B. Lowekamp, and D. O'Hallaron. Preliminary report on the design of a framework for distributed visualization. In *Proceedings of the Parallel and Distributed Processing Techniques and Applications (PDPTA99)*, Las Vegas, NV, 1999.

[2] A. Afework, M. D. Beynon, F. Bustamante, A. Demarzo, R. Ferreira, R. Miller, M. Silberman, J. Saltz, A. Sussman, and H. Tsang. Digital dynamic telepathology - the Virtual Microscope. In *AMIA98*. American Medical Informatics Association, November 1998. Also available as University of Maryland Technical Report CS-TR-3892 and UMIACS-TR-98-23.

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

[4] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, 2000.

[5] H. Andrade, T. Kurc, U. Catalyurek, A. Sussman, and J. Saltz. Persistent caching in a multiple query optimization framework. In *Proceedings of the 6th Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Washington, DC, March 2002.

[6] H. Andrade, T. Kurc, A. Sussman, E. Borovikov, and J. Saltz. On cache replacement policies for servicing mixed data intensive query workloads. In *Proceedings of the 2nd Workshop on Caching, Coherence, and Consistency, held in conjunction with the 16th ACM International Conference on Supercomputing*, New York, NY, June 2002.

[7] 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 Supercomputing Conference*, Denver, CO, November 2001.

[8] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Multiple query optimization for data analysis applications on clusters of SMPs. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, Berlin, Germany, May 2002.

[9] 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*, Fort Lauderdale, FL, April 2002.

[10] M. Arlitt, R. Friedrich, and T. Jin. Performance evaluation of web proxy cache replacement policies. In *Proceedings of Performance Tools'98*, Palma de Mallorca, Spain, September 1998.

[11] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on I/O and Parallel and Distributed Systems*, Atlanta, GA, 1999.

[12] W. Bethel, B. Tierney, J. Lee, D. Gunter, and S. Lau. Using high-speed WANs and network data caches to enable remote and distributed visualization. In *Proceedings of the 2000 ACM/IEEE Supercomputing Conference*, Dallas, TX, November 2000.

[13] M. D. Beynon, T. Kurc, A. Sussman, and J. Saltz. Design of a framework for data-intensive wide-area applications. In *Proceedings of the 9th Heterogeneous Computing Workshop*, Cancun, Mexico, May 2000.

[14] M. D. Beynon, A. Sussman, U. Catalyurek, T. Kurc, and J. Saltz. Performance optimization for data intensive grid applications. In *Proceedings of the Third Annual International Workshop on Active Middleware Services (AMS2001)*, pages 97–105. IEEE Computer Society Press, August 2001.

[15] 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*, Rhodes, Greece, June 1999. ACM Press.

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

[17] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri. A component based services architecture for building distributed applications. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC-9)*, Pittsburgh, PA, August 2000.

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

[19] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing (HPDC-10)*, August 2001.

[20] J. Dilley and M. Arlitt. Improving proxy cache performance: Analysis of three replacement policies. *IEEE Internet Computing*, 3(6):44–50, November/December.

[21] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid – an open grid services architecture for distributed systems integration, 2002. Draft document available at http://www.globus.org/research/papers/ogsa.pdf.

[22] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 83–92, November 1998.

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

[24] C. Isert and K. Schwan. ACDS: Adapting computational data streams for high performance. In *Proceedings of the 2000 IEEE International Parallel and Distributed Processing Symposium*, pages 641–646, Cancun, Mexico, May 2000. IEEE Computer Society Press.

[25] W. Johnston, J. Guojun, G. Hoo, C. Larsen, J. Lee, B. Tierney, and M. Thompson. Distributed environments for large data-objects: Broadband networks and a new view of high performance, large scale storage-based applications. In *Proceedings of Internetworking'96*, Nara, Japan, September 1996.

[26] K. Kennedy, M. Mazina, J. Mellor-Crummey, K. Cooper, L. Torczon, F. Berman, A. Chien, H. Dail, and O. Sievert. Toward a framework for preparing and executing adaptive grid programs. In *Proceedings of NSF Next Generation Systems Program Workshop*, Fort Lauderdale, FL, April 2002.

[27] R. Oldfield and D. Kotz. Armada: A parallel file system for the computational grid. In *Proceedings of the 1st IEEE International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, May 2001. IEEE Computer Society Press.

[28] B. Plale, P. Dinda, M. Helm, G. von Laszewski, and J. McGee. Key concepts and services of a grid information service, February 2002. Draft document available at http://www.cs.indiana.edu/plale/GISggf4.pdf.

[29] B. Plale and K. Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC-9)*, Pittsburgh, PA, August 2000.

[30] M. Rodríguez-Martínez and N. Roussopoulos. MOCHA: A self-extensive database middleware system for distributed data sources. In *Proceedings of the 2000 ACM-SIGMOD Conference*, pages 213–224, Dallas, TX, 2000.

[31] SRB: The Storage Resource Broker. *http:// www.npaci.edu/ DICE/ SRB/ index.html*.

[32] R. Szeliski. Rapid octree construction from image sequences. *Computer Vision, Graphics, and Image Processing. Image Understanding*, 58(1):23–32, 1993.

[33] B. Tierney, W. Johnston, J. Lee, G. Hoo, and M. Thompson. An overview of the distributed parallel storage server (DPSS). Available at http:// www-didc.lbl.gov/ DPSS/ Overview/ DPSS.handout.fm.html.

[34] D. Wessels and K. C. Claffy. ICP and the Squid web cache. *IEEE Journal on Selected Areas in Communications*, 16(3):345–357, April 1998.

[35] R. Wolski, J. Brevik, C. Krintz, G. Obertelli, N. Spring, and A. Su. Running Everyware on the computational grid. In *Proceedings of the 1999 ACM/IEEE Supercomputing Conference*, Portland, OR, November 1999. ACM Press.

[36] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5–6):757–768, October 1999.