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

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

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

[‡]Dept. of Biomedical Informatics
The Ohio State University
Columbus, OH, 43210

{kurc.1,saltz.1}@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 an 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. There is a variety of recent grid computing projects where interactivity and process composition have played important roles. 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 (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 a rapid response is needed, performance can be improved by reusing previously cached results for a new query. Efficient scheduling of queries from multiple clients is another optimization that can be applied to improve system performance. 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 [8, 17].

A grid-based cluster 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. How-

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

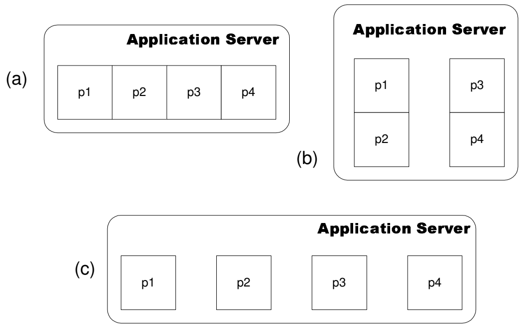


Figure 1. An application server may use many different parallel configurations depending on what is most efficient for an application. (a) shared memory, (b) distributed shared memory, or (c) distributed memory.

ever, such an environment requires distributed access to and processing of data in a heterogeneous setting. Component-based frameworks and services have been gaining acceptance as a viable approach for application development and execution in distributed environments [1, 4, 9, 11, 12, 16, 19, 20, 23, 25]. Such models facilitate the implementations of applications and services that can accomodate the heterogeneous and dynamic nature of the Grid.

In previous work [6], we have developed a framework for efficiently executing multiple query workloads from data analysis applications on SMP machines and distributed-memory parallel machines. In this work, building on that framework, we are developing 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), and a persistent data caching service (cache server) (see Figure 2). In this paper, we focus on the design and implementation of the **Active Proxy-G** (APG) service. Employing APG requires adding three types of functionality to the original application structure to employ the proxy. First, the application 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. Finally, if the request can be partially answered from the cache, the application must retrieve the cached results and generate requests (subqueries) to produce the remaining results to the application server. All three types of functionality are supported as Active Proxy-G core services.

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 is a common framework for a class of data analysis applications (see Section 3).

Rodríguez-Martínez and Roussopoulos [26] 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-ship is the best solution alone. Several highly distributed applications have employed proxy servers to great benefit. Beynon et al [13] 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 [29] 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 works. However, we argue that in addition to being more generic, our framework is potentially more powerful performance-wise because it employs a semantic cache that allows an active proxy to perform computations to allow re-using cached results to satisfy request even when a data transformation function must be applied. Additionally, it employs an extensible query server engine which 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. [30] 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) [21, 28]. The most important aspect of their approach is to use parallel operation of distributed servers to supply data streams fast enough to enable various multi-user, real time applications in an Internet environment. Bethel et al. [10] show how DPSS is used for building Visapult, a prototype and framework for remote visualization of large datasets. Allcock et al. [3] points out that the data grid infrastructure will need to service thousands of users efficiently, and also highlight the point that the management of data and replicas is also an important aspect

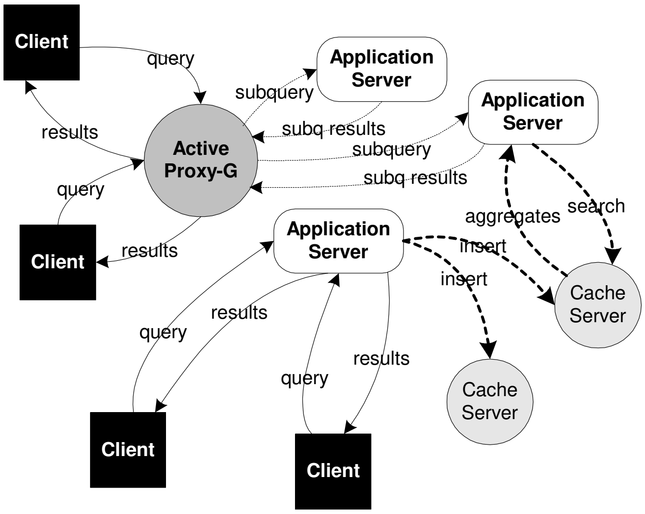


Figure 2. The suite of services, which consists of Active Proxy-G, application query servers, and persistent cache servers, for optimizing the execution of multiple query workloads in a Grid-based cluster environment.

of grid-aware applications. In some sense, our approach is complementary to these works because it also enables an application to explore the parallel capabilities of many application servers, but it also goes a step beyond since it allows the proxies to help with the computation by leveraging cached aggregates, and automatically generating subqueries transparently.

Recent efforts in the Grid research community [22, 24] are investigating and proposing the mechanisms for executing adaptive grid programs – Grid Application Development Software (GrADS) – and the support mechanism for storing metainformation needed to control that execution – Grid Information System (GIS). Several research projects have investigated the design, implementation, and application of component-based frameworks for application development and deployment [1, 9, 11, 12, 20, 23, 25]. The Common Component Architecture project by Bramley et al. [15] leads a standardization effort for building distributed software component systems for scientific and engineering applications. The Open Grid Services Architectures effort by Foster et al. [18] draws from concepts and technologies that evolved in the Grid and web world to generalize an architecture viable for deploying largely distributed commercial and scientific applications. These initiatives are examples of how our system will have to evolve in order to be integrated to a much larger infrastructure, by being compliant to the models that are eventually become standards, protocols and best practices.

Finally, for the whole bulk of work in the multiple query optimization problem, we refer the reader to our previous works [5, 6, 7], in which we extensively discuss the related research and compare it to our own approach.

3 Query Processing and Data Reuse in Data Analysis Applications

Although many data analysis applications seemingly 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 multi-dimensional space, and 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]. Accumulator 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 accu-

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

Figure 3. The query processing loop.

mulator are post-processed 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), 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. 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 meta-information class, *QueryMI*. Customization of APG for application-specific queries is achieved by subclassing the provided base classes and implementing a set of virtual methods. The following virtual methods are required for customization:

`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 the 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 structure point to the cached intermediate result, or much more computationally intensive, as in projecting an intermediate result represented on one

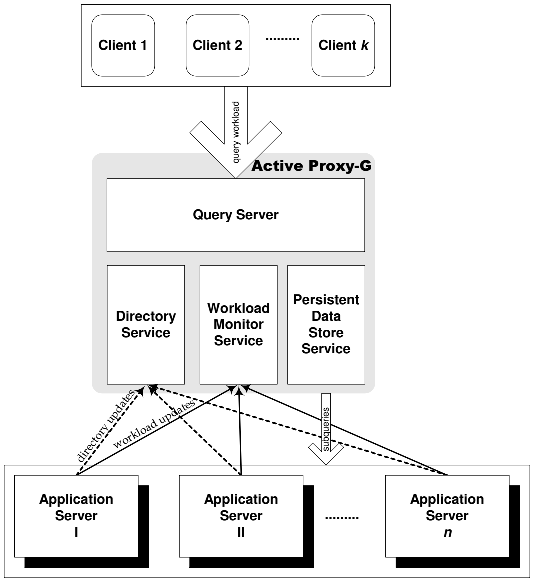


Figure 4. The APG functional components. The directory service maintains information about the location and access methods for application servers and datasets. The workload monitor collects metrics on server and network performance, and the persistent data store maintains cached results along with their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

multi-dimensional grid to a different grid, which requires performing an expensive computation.

`generateSubQueries`: Given the metainformation describing which parts of the query were answered using the cached results after the project operation, generate 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 tries to use the maximum number of cached intermediate results to process the query completely from the cache. 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 cannot reuse any cached results. 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 The 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 remote subquery is generated (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).

Algorithm 1 *void* Query::runAndShipResults()

1: **if** $qmi.isOk()$ **then**
2: $stat = findOverlaps(ovlps\_list)$
3: **if** $stat \neq COMPLETE$ **then**
4: $allocOutputMemory(ovlps\_list)$
5: $allocTmpMemory(ovlps\_list)$
6: **if** $stat \neq NO\_MATCH$ **then**
7: $project(ovlps\_list)$
8: **if** $stat == PARTIALLY\_COMPLETE$ **then**
9: $subq\_list = generateSubQueries(ovlps\_list)$
10: $ret = processSubQueries(subq\_list)$
11: **else if** $stat == NO\_MATCH$ **then**
12: $ret = executeRemotely()$
13: **else**
14: $ret = METAINFO\_ERROR$
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: $project(buf\_list)$
8: return ret

We now describe the four major components of the APG:

Query Server. 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 discussed in this section, 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 fairly assign the workload to application servers without overloading any server. Fault tolerance, is important because, in a highly distributed environment, application servers may become unavailable and later become again available frequently, and 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. APG can be interfaced with more powerful monitoring services such as the Network Weather Service (NWS) [31], to provide more sophisticated information that can be used to better determine which application server is the *best* candidate server for a given query. Such information can include overall machine utilization (as opposed to the application server utilization that is already used), network bandwidth, and network latency.

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. It is a two tier hierarchy, using both a portion of main memory plus a large chunk of secondary storage. The replacement policies used insure that more useful intermediate results are maintained in memory, while other intermediate results may get swapped out to the secondary storage cache. Intermediate results are only purged from the persistent cache when secondary storage space becomes inadequate to hold new computed intermediate results.

5 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 [5].

5.1 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 [7]. 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 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 *findOverlaps* method, was implemented to intersect two regions and return an overlap index, which is computed as

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

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 intermediate 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.

5.2 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 [14, 27]. 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). A query q_i is described by a query meta-information 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 a subset of the set of cameras for a given dataset. 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. Deeper octrees represent a higher resolution for the output 3-dimensional 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 algorithm employs parts of an earlier implementation [14] 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 meta-information. 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 is provides a customization of *findOverlaps* method.

Algorithm 3 *float* overlap(M_i, M_j)

if $D_i \neq D_j$ **then**
    return 0;
$v_{ovlp} \leftarrow \frac{CommonVolume(B_i, B_j)}{Volume(B_j)}$
$f_{ovlp} \leftarrow \frac{|F_i \cap F_j|}{|F_j|}$
if $C_i \supset C_j$ **then**
    $c_{ovlp} \leftarrow \frac{|C_i|}{|C_j|}$
else
    $c_{ovlp} \leftarrow 0$
$d_{ovlp} \leftarrow 1 - 0.1 \times (d_i - d_j)$
return $v_{ovlp} \times f_{ovlp} \times c_{ovlp} \times d_{ovlp}$

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 implemented the *frames* project function. We will explore the usefulness of the other projection operations in future work.

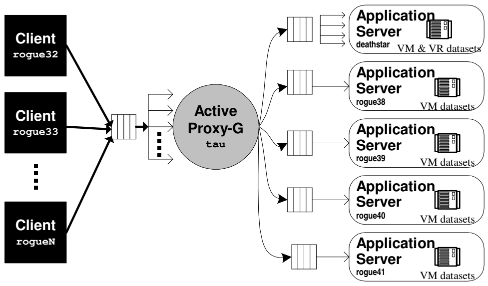


Figure 5. Experimental setup.

6 Experimental Evaluation

Many factors affect the performance of the Active Proxy-G. One basic question concerns the portion of the query workload that can be serviced from cached results stored at the APG. Therefore the first set of experiments show results for various sizes of the APG cache, as well as for various levels of multithreading 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 what this capability is able to improve performance-wise 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 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.

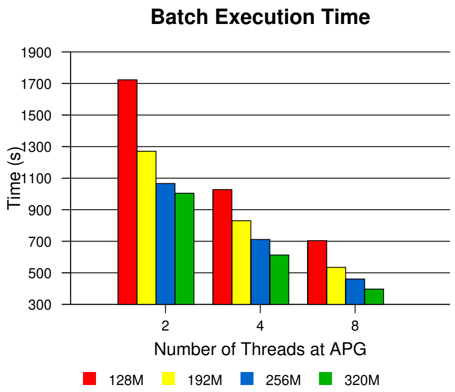


Figure 7. Batch execution time as reported by the APG. We varied the number of simultaneous executed queries by increasing the Query Server thread pool and the amount of cache space available to the Persistent Data Store Service.

Evaluation of scalability and the size of the PDSS.

Two of the most important aspects as far as performance is concerned in the *componentized* architecture we propose are the level of multithreading and amount of caching space available for storing reusable aggregates. In order, to evaluate the impact of these two variables, we assembled an experimental setup of clients, application servers, and an instance of the Active Proxy-G which is depicted in Figure 5. We have instantiated 5 application servers on 5 nodes in our experimental cluster: $deathstar$ is an 8-processor 550MHz Pentium III Linux SMP machine hosting 8 datasets (two for Volumetric Reconstruction and six for the Virtual Microscope), $rogue38 \ldots rogue41$ are single-processor 650 MHz Pentium III machines hosting the same six VM datasets as $deathstar$. $Deathstar$ had a single application server serving both VM and VR queries, with up to 4 simultaneous queries under 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 UltraSparc III Solaris SMP machine. We employed 12 clients in total. Four of them generated 16 queries each for volume reconstruction, and the other 8 generated 32 pixel averaging VM queries each. The VM clients used a workload model which emulates the behavior of real users as described in [13]. 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), in which "hot frames" were pre-selected, and the length of a "hot interval" was characterized by a mean and a standard deviation. Each VM dataset is a 10000×10000 3-byte pixel

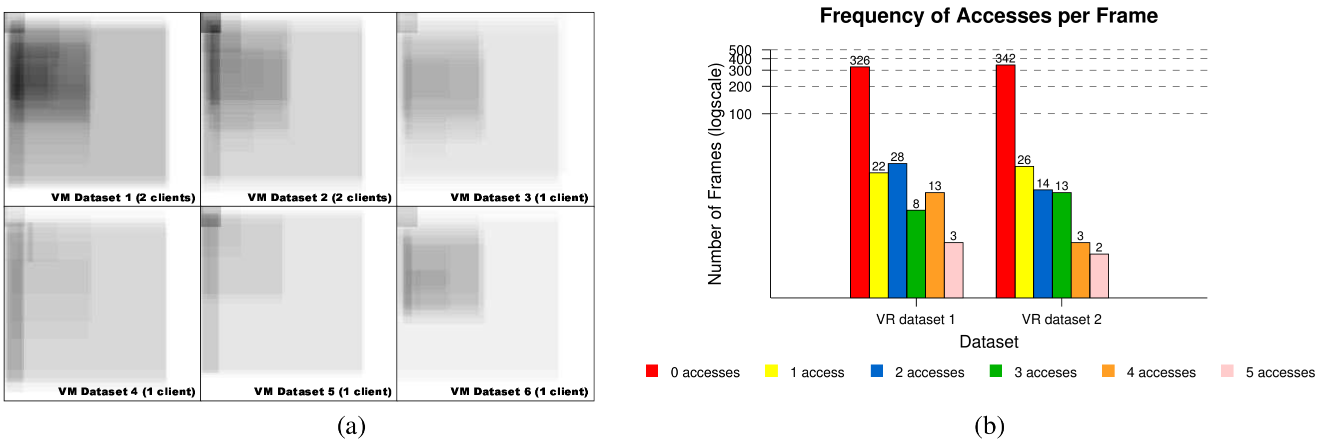


Figure 6. Caching relies on locality amongst the queries submitted to the system. (a) VM workload: 6 images were used. The darker a region is, the more popular it was – meaning that important features were located in those regions. (b) VR workload: each of the datasets was composed of 400 frames (for each of the 13 cameras). The chart shows the popularity of each of the frames as a function of the number of accesses generated by the collection of queries that accessed them.

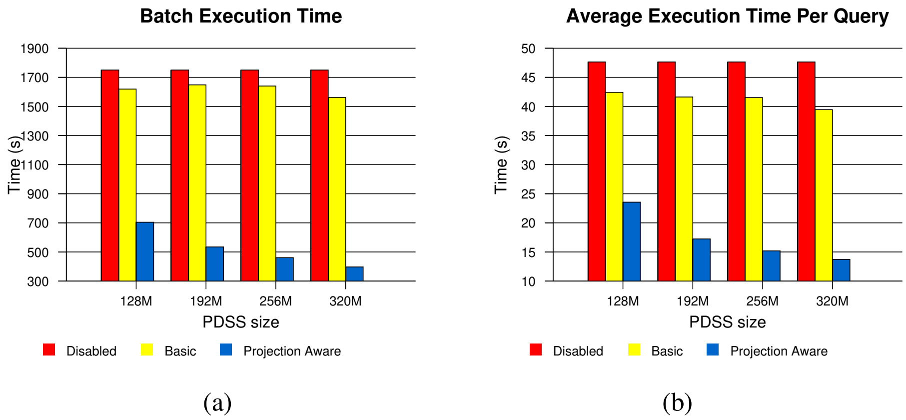


Figure 8. Caching strategies. *Basic* **allows reuse of aggregates that are a complete 100% match.** *Projection-Aware* **allows reuse of aggregates when a partial overlap is found and a data transformation is necessary. (a) Batch execution time. (b) Average execution time per query.**

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 2 collections. In Figure 6, we see the level of locality present in the experimental workload. This figure gives an idea of the size of the working set, which is essentially the metric that will drive how effective PDSS will be in improving the system performance for the incoming queries.

In Figure 7 we see that the amount of space allocated to the persistent data store can greatly influence the time for executing the batch of 296 queries submitted to the system. In terms of multithreading level, we see a drop of no less than 40% in execution time, when we increase PDSS size by 192MB[2]. In varying the multithreading level, we see speedups ranging from 1.50 up to 1.67. The APG was also instrumented to collect the *average overlap ratio* which is the average fraction of a query that is answered from cached aggregates (which is completely computed without the help of any of the application servers). We observed that it increases from approximately 0.60 up to approximately 0.80 as the PDSS size increases regardless of the multithreading level. Nonetheless, the average overlap ratio for an equivalent configuration in terms of PDSS size, but different multithreading level is higher for lower multithreading level, which suggests that more queries being executed simultaneously will compete for memory, as expected. This competition causes the ejection of potentially useful aggregates. This particular result shows that when a higher multithreaded level is enabled, an equivalent increase in the space available to the PDSS is necessary to keep the same average overlap ratio.

Evaluation of the caching strategies.

A novel aspect in our framework is concerned with reusing cached aggregates by applying transformation functions. Intuitively it is a potentially profitable approach in situations where the transformation is less expensive than recomputing the aggregate from input data. The results in Figure 8 compare a *basic* cache implementation, with no caching and the *projection-aware* caching we employ using the same workload as in the previous experiment. It can be seen that using the basic caching is responsible for a decrease of around 10% in the batch execution time compared to not caching at all, however enlarging the PDSS does not achieve much. As a matter of fact, the overlap ratio stays constant at 0.27 which means that the whole working set is kept in the cache even at 128MB. The projection-aware caching benefits represents a decrease from 56% up to 75% in terms of batch execution when compared to the basic caching which shows the improvements made by our approach.

[2]PDSS uses an incore and an out-of-core area. The incore area was fixed at 128MB, and the out-of-core portion varied from 0 to 192MB. In order to avoid the effects of the operating system buffer cache we used the `directio` primitive to hint the OS for performing direct I/O disk operations. We also use synchronous read/write disk operations, by opening the file descriptors with `O_SYNC`, `O_DSYNC`, and `O_RSYNC`.

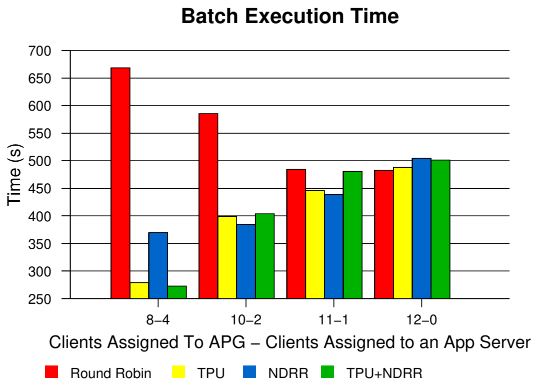


Figure 9. Impact of different application server assignment policy for various workloads and configurations. Under each bar is the workload configuration in terms of how many clients are using the Active Proxy (first number) and how many are interacting directly with an application server (second number).

Evaluation of the application server assignment policy.

As previously seen in Figure 2, queries can be shipped for execution at the application servers either from the APG or directly from the clients. Moreover, variances in performance and load for each of the application servers, as well as, network bandwidth and latencies may be responsible for wildly varying query response times. The Workload Monitor Service implements a simplified facility that can help the APG to better assign queries to application servers in situations where multiple candidates are available. Although in its current form the WMS does not collect metrics for network behavior, it is able to gather metrics of thread pool utilization and disk load at the application servers. It accomplishes that by polling (currently at a periodic rate of once every 15 seconds) each of them. 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 $NDRR$. The former shows the percentage of threads in the query server thread pool that is busy at the time of polling. And the latter shows what the disk read rate has been since the last polling period. It is normalized by assuming an *ideal* 30MB/s sustained transfer rate, which happens to be the nominal transfer rate for the disks in our machines. The experiments in Figure 9 show the results for a round robin assignment, and for variations of load based strategies. The combined load

l for a given application server is actually computed by the equation $l = TP_u + NDRR$. The decision for server assignment is achieved by finding the server with the smallest load amongst the *capable* servers as informed by the LDS. In case of ties, round robin is used to pick one from the tied servers. This is a simplified model, which is biased to queries that have similar I/O and computation requirements. However, it is reasonably effective as the results show.

The workload we used to gather the results in Figure 9 are 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. From the original 12 clients in the first setup (first cluster of bars in Figure 9), 8 submitted queries to the APG, and 4 to 4 of the application servers directly (hosted at $rogue38$, $rogue39$, $rogue40$, and $rogue41$). In the second, 10 submitted queries to the APG, and 2 to 2 other application servers (located at $rogue40$ and $rogue41$), and so on. The results in the chart are the ones reported by the APG. Because the number of queries submitted to it varied due to the varying numbers of clients submitting queries to the APG, the clusters of bars are not directly comparable. Nevertheless, the trend is clear, the more distorted the assignment is, the more effective the load-based strategies become, since it better assigns queries to relatively unloaded application servers. Indeed, for the 8-4 configuration, the decrease in the batch execution time can be as large as 59%. For the 12-0 assignment, the load-based variations are slightly more expensive than round robin. The explanation is twofold. First, the system is under a very high workload and any policy that distributes the work in an equitable mode is going to perform comparably. Second, the load-based strategies are slightly more expensive to compute than round robin. An important observation, is that for higher distortion levels (i.e. 8-4), $NDRR$ alone will not provide accurate information about to where a given query should be routed for execution. In fact, we measured the execution of a single typical VR and VM queries (assuming no reuse), and a VR query spends 52.5% of its execution time on computation and 47.5% on I/O, and a VM query spends 84.3% on computation and 15.7% on I/O. These metrics show why $NDRR$ alone does not provide good information about the load of an application server. Since our queries are more computation intensive rather than I/O intensive, TP_u alone did not behave as badly, however for I/O intensive queries it certainly would. Hence, the linear combination of the two variables seems to be a nice compromise towards the average case. Of course, if knowledge about the workload is available more precise strategies can be designed, an a self-tuning policy can be devised.

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. In a highly distributed environment as the one we envision for our infrastructure, the load can vary from very light to wildly 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, because in the experimental configurations there are 5 application servers, and 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 100% when the APG is configured for handling 8 simultaneous queries.

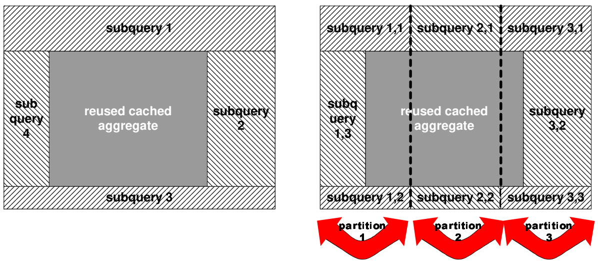
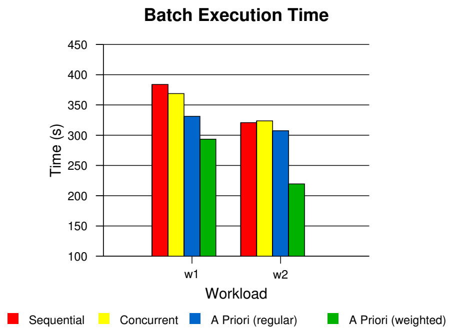


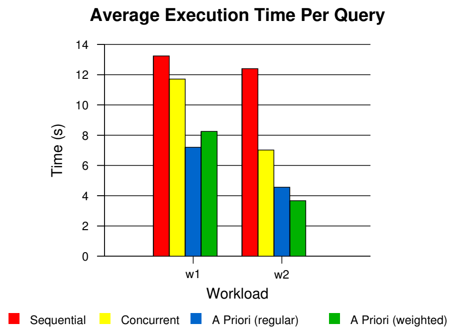
Figure 10. The right diagram shows the automatically generated subqueries which can be potentially concurrently executed by the system. The left diagram shows a query being *a priori* partitioned, and the subqueries generated.

Idle resources under lighter workloads can be better leveraged by making use of more sophisticated query planning and execution strategies. In Figure 11, we present the results for exploring two new query planning and execution strategies, namely *Concurrent* subqueries and *A Priori Partitioning*. The former 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[3]. And the latter consists of slicing a query into multiple subqueries when the query is received by the Query Server (Figure 10) 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. In order to evaluate the improvements for 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

[3]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.



(a)



(b)

Figure 11. 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 $weighted$ denotes a query partitioning schema for the *a priori* partitioning strategy (see details in the text) (a) Batch execution time. (b) Average execution time per query.

queries each (using 2 different datasets). The *a priori* partitioning implements a heuristic for limiting how may subqueries to generate. An application specific metric[4] as well as the number of application 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. Either *regular* in which each application server accounts for one partition, or *weighted* where each application server gets as many partitions as its multithreading level.

Figures 11 (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 in the 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 $w2$ in terms of 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. This is explained by the fact that the more subqueries are generated due to partitioning, the better utilization of idle resources is achieved (higher parallelism) at the expense of more complicated query processing (due to an increase in bookkeeping and projection operations). In this case it implies in a higher overhead that cannot be paid off by the the higher Query Server utilization.

[4]For VM queries, the number of recommended partitions is computed as so each partition will require at least 4MB of input data. For VR queries, each frame in the set of frames in the query metainformation is used to generate a subquery by replicating the other attributes.

7 Conclusions

The main aim of our work was to elaborate on an architecture for supporting data analysis applications in the context of a highly distributed environments as the computational Grid. The design was based on services that can be integrated in different ways in order to accommodate specific workload scenarios as well as utilization scenarios. In particular, we have extensively evaluated an active proxy configuration which 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 queries based on aggregates it caches locally. This approach results in faster queries response, 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. Elsewhere [6] we have shown that an active caching approach is able to increase the performance of a single data analysis application. In the current work, we have shown that by making it available as a service and incorporating it into an active proxy, a larger community of clients using different applications and datasets can also be benefited further by a more rational use of resources. In fact, we foresee the APGs being used as web proxies in the sense that they are located closer to users where locality is greater, shielding them from network latencies and outside disruptions. Hierarchical networks of APGs can also be employed to ensure better scalability.

Albeit fully functional, our framework can be expanded in several different ways. Most importantly, it can easily leverage resources and technologies already made available by the Grid research community. Two of its important pieces, the Light Directory Service (LDS) and the Workload Monitor Service (WMS), are already described under other names in the Grid literature. In fact, the OGSA document [18] discusses how the functionalities implemented by the LDS are supposed to be supported by a generic *discovery service*. One of its nice features is the notification facility which enables clients interested in being notified of particular events to be informed of their occurrences. Such facility would enable the APG to automatically learn about new application servers becoming available for example. As far as the WMS goes, the OGSA describes some *higher-level services* and one of them is the class of *instrumentation and monitoring services* which are going to be used primarily for ensuring the integrity of the system, but can also be used for better workload scheduling decisions. In the near future, we plan to make our framework compliant to these still evolving architectures which will make it fully integrated in the Grid ecosystem.

References

[1] M. Aeschlimann, P. Dinda, J. Lopez, B. Lowekamp, L. Kallivokas, and D. O'Hallaron. Preliminary report on the design of a framework for distributed visualization. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 1833–1839, Las Vegas, NV, June 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 *the USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[5] H. Andrade, T. Kurc, A. Sussman, E. Borovikov, and J. Saltz. Servicing mixed data intensive query workloads. Technical Report CS-TR-4339 and UMIACS-TR-2002-21, University of Maryland, February 2002.

[6] 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.

[7] 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 International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, April 2002.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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 *Proceeding of the Ninth IEEE International Symposium on High Performance Distributed Computing*, Pittsburgh, PA, August 2000.

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

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

[18] 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.

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

[20] C. Isert and K. Schwan. ACDS: Adapting computational data streams for high performance. In *14th International Parallel*

& Distributed Processing Symposium (IPDPS 2000), pages 641–646, Cancun, Mexico, May 2000. IEEE Computer Society Press.

[21] 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.

[22] 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.

[23] R. Oldfield and D. Kotz. Armada: A parallel file system for computational. In *Proceedings of CCGrid2001: IEEE International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, May 2001. IEEE Computer Society Press.

[24] 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.

[25] 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)*, August 2000.

[26] 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.

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

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

[29] 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.

[30] 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.

[31] 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.