DVS: Distributed Volatile Storage for low-latency read access

Radu Berinde, Daniel Dumitran, Igor Ganichev, Mike Lin

Abstract

We describe the design and implementation of DVS, a distributed, fault-tolerant in-memory storage system for fast random access to large datasets. The storage has a key-value interface and is optimized for read performance. It is used as a cache for static data stored on slower durable mediums to accelerate access to large datasets, such as those used in bioinformatics. The key problems that DVS solves are data location on and data distribution to a dynamic set of servers. We show that DVS can speed up a real bioinformatics application by more than 100%.

1 Introduction

We have identified a class of applications that show the following properties:

- the computations are done by programs running on multiple computers
- the expected running times of the algorithms are in the $O(n)$ - $O(n \lg n)$ range; I/O contributes to a large amount of running time
- the output of the programs is usually small; result outputting does not use significant running time
- the static data used is not small enough to fit in one machine's RAM
- the static data used is small enough to fit in the total amount of RAM available on all the machines involved in the computations

Our project is centered around the problem of efficiently storing the data used by these programs. Our main idea is to use the available RAM on the machines to store volatile copies of the data, using the network to transfer data from machines that store it to machines that need it. Thus, we create a Distributed Volatile Storage (DVS) system.

The main class of applications that fit this tight description is related to the emerging field of computational biology. Bioinformatics programs process genomes and protein sequences. Even though sometimes corrections and additions are made to a genome or protein sequence (usually by human decision), the data is static in the sense that the programs only read this data.

1.1 BLAST/BLAT

We present a particular bioinformatics algorithm as the sample use-case of our project. BLAST - Basic Local Alignment and Search Tool - is an algorithm for identifying sequences in a large database that are similar to a query sequence. BLAST is used often interactively by bench scientists and even lawyers involved in a gene patent filing, so even somewhat modest speedup factors would be worthwhile. Researchers run independent instances of BLAST on their own computer either using local copies of the data or a central file server. Many bioinformatics centers also have dedicated BLAST farms to facilitate large-scale analysis, and may not be able to make full use of these hardware resources with large sequence databases.

BLAT [7] - BLAST-Like Alignment Tool - is a program implementing a variant of this algorithm. BLAT's architecture is presented in figure 1. Part of BLAT is the *gfServer* that precomputes a hash table of words (short subsequences of a certain length) in the sequence database to identify possible regions of similarity to a given query sequence. BLAT *gfClient*s query the *gf-*

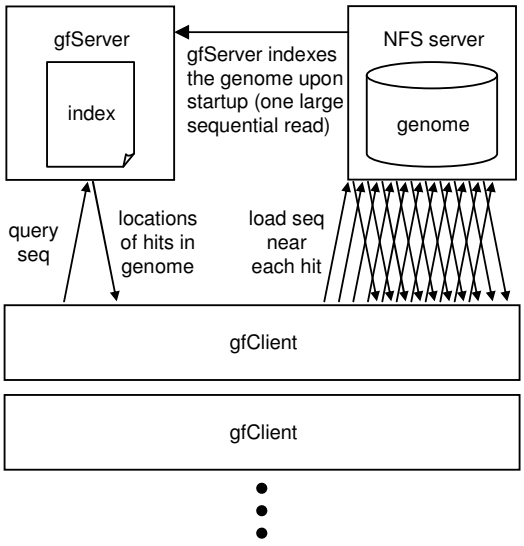

Figure 1: BLAT Architecture

Server to find candidate regions, which are then loaded from storage and processed in order to determine whether they show significant similarity with the query sequence. A region is typically less than 1,000 bytes. Usually, many database sequences have spurious word "hits" to the query sequence without significant longer-range similarity, and these spurious hits tend to distribute randomly throughout the database. Thus, the goal of our system is to increase the performance of *gfClient*s. We ran BLAT and found that a search against the NR protein sequence database, which is approximately 3GB and was stored on disk, took around 60 seconds on a modern PC, at less than 25% average CPU usage during the search step.

BLAST programs exhibit an additional property to the ones presented in the first subsection: **it reads many small randomly-distributed portions of the large static dataset**. This property indicates that the usual solution of storing the data on a hard drive (locally or remotely) is inefficient because of disk seeks - reading small portions of data randomly distributed on the storage medium is the worst possible workload for a hard disk. [[6], Fig.1] shows that the running time of NCBI-BLAST is roughly proportional with dataset size; however, when the dataset becomes too large to fit in one computer's memory, the proportion constant increases 6-fold.

1.2 Current solutions

Some bioinformatics clusters, like the one used by Broad Institute, store the data using hard disks on a central file server. The data is put in files that are accessed through a shared filesystem. We feel that the productivity of such a cluster can be increased as long as the size of the data corresponds to our requirements.

The current state-of-the art solution for parallel BLAST is mpiBLAST [6]. It is a parallel BLAST implementation tailored for clusters; it works by dividing the database between workers and distributing queries to workers which hold the corresponding data segment. mpiBLAST aims to give each machine a portion of the data that fits in the machine's RAM, making it capable to process respective queries without disk I/O. This is a particular solution for BLAST on clusters, and our system cannot compete with it. The purpose of our system is to provide a more general solution to an entire class of applications without redesigning them from scratch - including single-machine BLAST programs that researchers run on their own machines rather than a cluster.

Since the static data fits in the RAM of the machines in the cluster, it probably fits on a single hard drive. Each machine has a hard drive anyway, so a possible solution is to store all the data on every machine's hard disk. One problem with this solution is that the data is hard to manage - updating the genomes or sequences would be hard. Another problem is that even a local hard drive might be slower than a network transfer, especially with non-sequential reads (like in the BLAST case).

1.3 System requirements

We identify the requirements of a system that can efficiently serve the data to programs that process it; these are the main challenges we face in designing the system:

Performance. When run on machines on the

same network the latency of the read(**get**) operation must be smaller than the latency to read from a local hard drive - the average seek time. The motivation for this decision is the BLAST usage scenario, where an alternative solution would be to store hard copies of the genomes on each machine. Comparable write (**set**) performance is preferred, although not required.

Weak Consistency. The system must guarantee that a read never results in the wrong data; a read must return either the correct data or nothing if the data cannot be found (perhaps lost because of failures).

Failure resistance. The system must adapt to machine failures; the data lost because of failure must be repopulated from the slower stable storage.

Manageability. The system must allow addition and removal of new machines (within some pre-established limits) during operation without any reconfiguration on the already running machines.

One might argue that the last requirement is not very important; however, the manageability feature implies that the system allows recovered machines to rejoin the system. This is very important; if the set of machines in the system can only decrease, the system will always need a total restart at some point in time. Restarting the whole system is not only slow - the whole data must be read from stable storage - but it also requires human intervention. Running programs would also probably need restarting, or would at least have to wait while the system is being re-populated.

A scenario where this last feature is vital is one where there is no dedicated cluster available. An example is where researchers interactively run (single-cpu) BLAST on their local machines, either using local copies of data or a shared file server. In this scenario, the researchers should be able to share their machine's resources through the storage system by voluntarily running a server program in the background when they choose to do so - this clearly shows that the system needs to adapt to machines joining and parting.

2 Related Work

A system centered around caching data in the free RAM of multiple machines is *memcached*. The goal of the *memcached* system is to provide regular cache semantics to the applications on a cluster while using memory on multiple machines of the cluster. In particular, the cache starts out empty and the applications check the cache before they access any object from a database (typically stored on the disk). If the object is not in the cache, applications request it from the database and then put it into the cache. When the cache fills up, either the least recently used object is discarded or subsequent *set* operations are ignored, depending on configuration.

While the design of *memcached* is not documented, we were able to study the source code which is publicly available. The distributed aspect of *memcached* is implemented solely in the client library; there is no coordination between the *memcached* servers. Each server acts as a stand-alone storage server and the client library chooses the server on which to store a given key. *memcached* cannot handle new servers joining the system while its running - each client program initializes the library with the list of servers. It does handle failures in the sense that once a server has failed, it is marked as deactivated and all future requests with keys corresponding to the failed server are routed to the next available server in the list; however, once a server crashed, it will not be used again by a running client, even if the server recovers. Because of this issue, the image of the system can easily become inconsistent between different clients and thus inefficiently use resources by storing the same data multiple times. These compromises are probably acceptable in *memcached* usage scenarios and they greatly simplify the system and its implementation.

A system which greatly influenced our design choices is the Porcupine mail service [1], which implements a decentralized design using node homogeneity. Making the system homogeneous

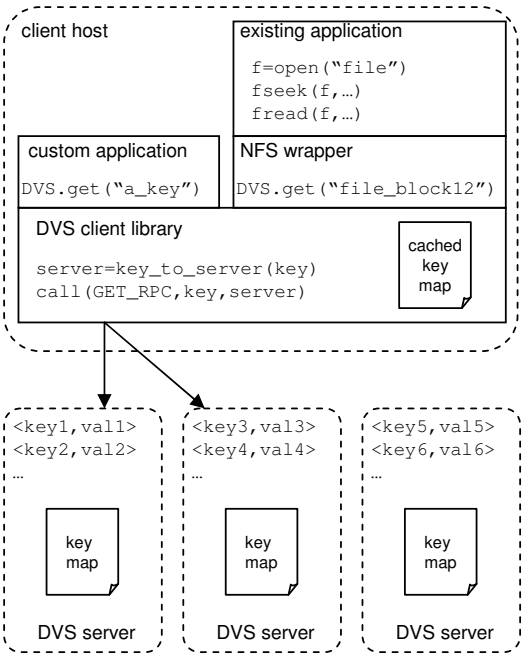

Figure 2: DVS General Architecture

avoids introducing a single point of failure as well as eliminates a possible performance bottleneck. DVS uses a dynamic membership algorithm similar to the one used in Porcupine and uses a similar in-circle-chatter method.

3 Design

We attempt to fulfill the requirements with a system that exhibits *functional homogeneity*, that is, all operations can be performed by any machine in the cluster. Thus, we avoid designating one of the machines as "the manager" for any kind of system data. Any data structure needed must be maintained by all machines. The scheme of the DVS system is presented in figure 2.

The main challenge for DVS was to create protocols and policies that allow the system to behave in the desired manner while restricting decisions taken by any machine to involve only parts of the dataset maintained by that machine.

3.1 Buckets

We chose a key/value pair interface for DVS, since it provides the generality that we need in order to support any application that fits our requirements. Since any key/value pair can be stored on the system, we have to define some way to distribute the keys among the live DVS servers. Because of the manageability requirements of DVS, we cannot statically define a key-to-host mappings; the system must maintain the key-to-host mapping dynamically.

Since the number of different keys can get quite large, a simple key-to-host list would easily become unmaintainable. Thus, we choose a "hybrid" strategy: we split the key-space into a fixed number of *buckets*, and define a static key-to-bucket mapping while dynamically maintaining a bucket-to-host mapping. The number of buckets should be a small multiple of the number of possible DVS server (e.g. 1600 for a cluster with 100 machines). The dynamic bucket-to-host map has several thousand entries, so its size is only a few kilobytes.

3.1.1 Key-to-bucket static map

For the static key-to bucket distribution we use a deterministic hash function (known to all machines) to map any key to a bucket. Because we use DVS to cache file blocks (see 3.7), most keys have values of the same size (8Kb; some have less). Thus, we can assume that the hash function distributes the data evenly, resulting in buckets of approximately equal size.

3.1.2 Bucket-to-node dynamic map

The dynamic map is maintained by all nodes in the cluster. It stores for every bucket a list of nodes which have copies of the keys in that bucket. We allow a node to add or remove itself from one of these lists. The challenge in implementing this behavior is to define a *rebalancing protocol* that maintains agreement between servers when more than one node tries to change the map at the same time.

We require that at any point in time - even during a change of the bucket map - any server that is listed in a bucket's server list must still have the values for the keys in the bucket, this being able to fulfill *get* requests.

3.2 Rebalancing Mechanism

DVS allows each server to decide what buckets it should maintain. First, we focus on the mechanisms that allow the behavior we want: at any point, any server must be able to start or stop maintaining a bucket. Later in the paper we discuss the policies that define the server decisions; the important point is that as long as the mechanisms are correct, the policies can be improved with minimal effort.

There is a problem when there is only one server maintaining a bucket, and that server wants to dispose of the bucket. In this case, the server is not allowed to simply remove itself from the bucket's server list in the bucket map because the bucket's values would be lost. To solve this problem, we define a "need-to-remove" flag for each bucket. In the presented scenario, the server sets this flag and waits until another server replicates the bucket before removing itself from the bucket's server list.

The operation that enables any node to add, remove, or flag as "need-to-remove" a bucket is called a "rebalancing" operation. It involves making a change to the dynamic bucket-to-node map; this operation can be invoked by any node usually because of nodes running out of memory, failing, or joining, as well as for load balancing. Note that when a server decides to start maintaining a bucket, it first gets all the values in that bucket before invoking the rebalancing operation. Similarly, a node removing a bucket first invokes the rebalancing operation before discarding the values in a bucket.

There are many situations in which problems arise if two servers try to change the bucket map at the same time. For example, if two servers maintain a bucket and they both decide to stop maintaining it, the bucket data would be lost - whereas if only one succeeds the rebalancing, the other server would flag the bucket rather than removing it or perhaps would not even decide to remove the bucket after examining the new configuration. The chance of two concurrent rebalancing attempts occurring at the same time is not small; a server usually decides to rebalance because of some external event, like a failure.

In general, a server must decide any reconfiguration upon examination of the latest bucket map; with concurrent updates, sometimes there is no "latest" bucket map and it would be very hard to define correct policies for rebalancing decisions. Thus, when two concurrent rebalacing attempts are made, only one should succeed. The failed attempt should not be blindly retried; the server might reach a different decision upon reexamination of the new configuration.

We describe the protocol that serializes the attempts to rebalance data that is similar to TRM[3]. We use the concept of a global lock associated with the right to change the bucket-to-node mapping. To serialize the acquire attempts, each node maintains a Lamport clock[2]. When a node wants to make a change to the bucket-to-host map, it reads its current Lamport clock and uses it as an epoch ID in a broadcasted "acquire" message. The node then waits for "ok" responses form all other hosts. Every node remembers the highest epoch ID received; if a node received an acquire message with a lower epoch ID, it replies with a "deny" message.

The result of this protocol is that exactly one node will receive "ok" from all other nodes. This node has obtained the lock, and denies any further acquire attempts until it is ready to release the lock. To complete the rebalancing, the node broadcasts a "commit" message including the desired change to the mapping and waits for acknowledgments; afterwards, the lock can be released.

If at any point during these three rounds a machine is detected as failed, the failed node is simply removed from the live set list and the operations continue as usually until the rebalancing change is finished.

If the machine initiating the attempt fails during the transaction, some machines may have received the changes and some might have not. However, this is not important since a node only initiates mapping changes related to that particular node and whenever a node fails, all the other nodes remove the references to that node

from the mappings.

To make sure that the rebalancing decision is made on the basis of the latest bucket map, the decision must be recomputed after the lock is acquired. Thus a DVS server checks (every few seconds) the bucket map and decides, based on the rebalancing policies, whether a change must be made. The server then attempts to acquire the lock, retrying until successful. After the lock is acquired, the server once again checks the bucket map and reaches a (perhaps different) decision, upon which it acts to commit.

3.3 Get protocol

The *get* protocol is simple - the client has a list of nodes in the cluster and chooses a random host to send the request to (if the host doesn't respond, another one is randomly chosen). The list of nodes doesn't need to be complete. Any node in the cluster can hash the key to find the bucket and examine the bucket-to-node map to find a node holding the bucket to forward the request to. The second node replies to the intermediary node, which sends the value to the client.

We can optimize this protocol: the key-to-bucket hash function can also be applied by the client. If the client had a copy of the bucket-to-node map, the request could be sent directly to a node holding the bucket, thus eliminating the overhead due to the node in the middle. An important point is that this map does not need to be up-to-date on the client; an incorrect entry simply results in a regular "unoptimized" request. The clients request the bucket map from a random node at large time intervals (20-30 seconds), so that the overhead of this optimization is minimal.

Note that the read requests are not affected by rebalancing operations; at any point before, during, or after a rebalancing, any server listed in the bucket map as a maintainer for a bucket in the bucket map still has the values for the corresponding keys.

3.4 Put protocol

To write a key value, a client again chooses randomly a node to send the request to. Any node can send the data to the node(s) responsible for the bucket. The same optimization can be used - the client chooses a server by reading the bucket entry in its copy of the bucket map. A DVS server receiving a write request sends the value to each server and waits for acknowledgments before returning.

If a write operation overlaps with a rebalancing operation, a new server starting to maintain a bucket might not receive the value of the new key. This issue is not a problem given our weak consistency model. Rebalancing operations are not frequent, so the DVS performance is minimally affected. Note that if needed, this problem can be fixed by storing in the bucket map two lists for each bucket - a list of servers to read from and a list of servers to write to; the idea would be that a server adding a bucket first adds itself to the write list, transfers the bucket values, and then adds itself to the read list.

3.5 Live set detection

We do not aim to handle all the possible cases of failures and network partitions. Since the dataset is static, it is not a fatal matter if live set inconsistencies arise due to network partitions and such. In the worst case, the live machines' resources might be very inefficiently used and the system would need a total restart; however, the programs will still run correctly (albeit slower).

We focus on ensuring that the live machines detect failed nodes and rebalance accordingly, and that a new machine can join the system.

3.5.1 Node failures

The only way in which a node can be declared failed is if it does not reply to an RPC. RPCs are used not only for regular requests and messages, but also in periodic in-circle pinging - each machine periodically sends a message to the machine with the successive node ID, except the one with the highest ID which sends statistics to the

one with the lowest ID. The node ID is a concatenation of the host's IP address and port number. Note that the in-circle messaging scheme could also be used to exchange usage patterns and statistics useful for advanced load balancing schemes.

When a node detects a failure of another node, it broadcasts a "node failed" message to all the other machines. The other machines remove this node from their live set list and bucket-to-host mappings. Nothing else needs to be done; the other nodes will automatically notice that some buckets are unmanaged by any node and will initiate rebalancing operations.

3.5.2 New node joining

When a new node joins the system, it must have the address of at least a node in the system; it sends an RPC to that node requesting a list of the live machines in the system.

After this step, the new node needs a copy of the bucket-to-node mapping. If a rebalancing occurs at this time, the "current" copy of the mapping might be unclear. To avoid this issue, the new node acquires the global lock used in rebalancings by broadcasting an "acquire" message. If the attempt is successful, it requests the mapping from a random node and then sends a blank "commit" message to release the rights.

The challenge lies in correctly solving the case when several machines try to join the system at (roughly) the same time. They can both join the system unaware of each other; in this state, the global lock mechanism would fail to ensure serialization of mapping changes and problems would occur.

To solve this case, we add functionality to the global lock mechanism: first, when a node receives an "acquire" message from an unknown node, it adds the node in its live set; second, when a node sends an "ok" reply to an "acquire" message related to a join operation, it piggybacks its list of the live nodes. Upon receiving the "ok" messages, the initiating node immediately processes these lists merging them with its own live set, and also sends "acquire" to any new nodes in these lists.

Assuming an initial consistent state of live views and a number of new nodes joining at the same time, the state of the system will eventually become consistent; the new nodes that join the system will immediately attempt to acquire the lock in order to request the bucket mapping; after all the nodes have acquired the lock (in some order), they have all become aware of each other.

Note that an entry of a failed node might make its way into a live set because of the view merging upon receiving "ok" messages; however, this is not a problem since the failure of the node will be re-detected and it will be eventually removed from the live set.

3.6 Rebalancing policy

We need to define the policy that affects the decisions behind the rebalancing operations. We have focused on supplying all the necessary mechanisms (locking, need-to-remove flag, in-circle messages) in order to allow any reasonable rules for rebalancing.

In our implementation, each server is started with an argument pre-setting the maximum amount of memory it should use. We use the following rules, given in their priority order:

1. A DVS server with free RAM must try to add an orphan bucket - one that is unmaintained by any server.
2. A DVS server with free RAM must try to add a bucket flagged as "need-to-remove".
3. A DVS server with free RAM should add a random bucket with a minimum replication factor
4. A DVS server maintaining a replicated bucket must try to remove it in favor of adding an orphan or flagged

Since the rules are applied in a serial order, it is clear that they will always enforce a maximum number of maintained buckets. In fact, if the cumulated available memory is approximately equal to the dataset size, only rule 1 is sufficient. If there is more available memory, the

other rules maintain a fairly even load balancing (in terms of incoming read requests). Note that in the current implementation, a server never flags a bucket with the need-to-remove flag; however, we did implement the functionality to allow better rebalancing policies. Rule 1 is applied however many times needed per rebalancing, while the other rules are applied only once per rebalancing because 2, 3 involve data transfer and 4 involves trashing data.

A number of improvements to these rules are possible. For example, if the servers exchange information about processing power and performance, rule 3 could choose buckets from slow or overloaded servers. Servers could adapt to CPU usage of other processes. We chose not to implement any advanced load balancing rules because in our tests with the BLAT alignment tool, the CPU usage of the DVS server processes was small (5-10%).

3.7 Interface

The client library interface is simple, exposing *get* and *set* functions. It is possible to write custom applications that use DVS directly; they would have to use *set* to upload and reload data when needed.

However, the programs that we are trying to provide data to are already designed to work with files; changing them could be a substantial effort, depending on their implementations. Because of this, we chose to write an NFS loopback server as a "wrapper" for DVS. The NFS server acts as a DVS client and simulates a filesystem, providing a transparent interface.

The data loading and reloading mechanism is also implemented in the NFS server. The wrapper assumes that the files are available on a shared filesystem and keeps them open. The wrapper can be started with an "-upload" argument to upload all the data to DVS. Each key corresponds to an 8Kb block in the file. The NFS server handles *read* requests by issuing one or two DVS *get* requests. If any DVS *get* returns an empty value, the NFS server reads the corresponding block from the "real" file, issues a DVS *set* with the data and returns to the client.

4 Performance

We measured the performance of our system by running BLAT on modern PCs with SATA hard disks, connected with a switched gigabit network. BLAT was not modified in any way; the tests were ran through NFS wrapper. As described in 1.1, BLAT has a *gfServer* and a *gfClient* part. We are interested in improving the performance of the *gfClient* reads; throughout our tests, we ran two BLAT *gfServer*s on different machines on the same network. We compare our results against clients using copies stored on the local hard drives.

We first tested the performance of a single client; we queried around 1500 predicted gene sequences from the fruit fly *Drosophila*

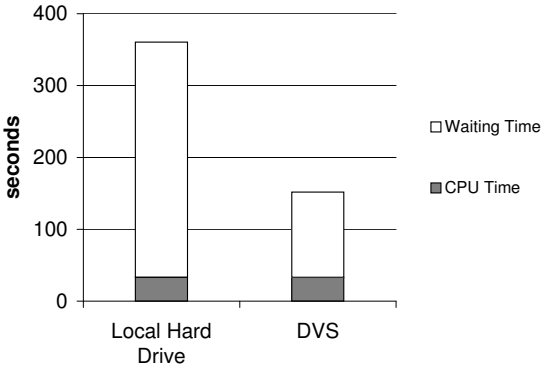

Figure 3: Performance of BLAT running from local hard drive versus BLAT using DVS (3 servers)

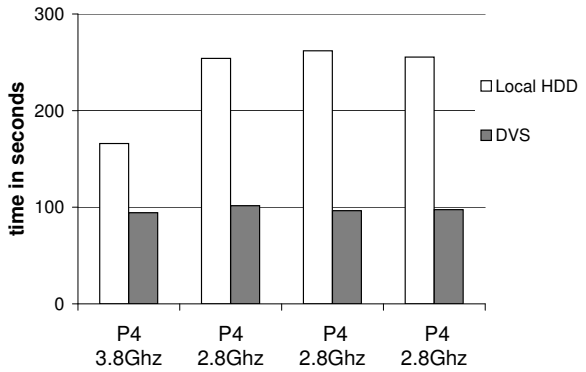

Figure 4: Four machines, each machine runs a DVS server and a BLAT client

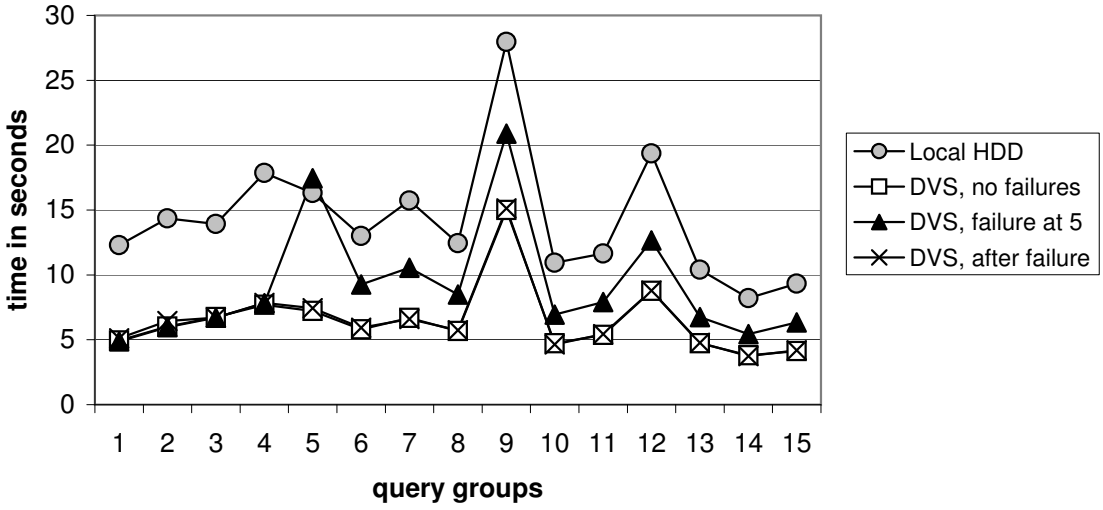

Figure 5: Performance comparison when one of four DVS server fails

melanogaster against the human chromosomes 2-9, 20-22, X and Y. The query sequences total to around 250kb, while the human (database) sequences use 900Mb. We ran a single BLAT client and 3 DVS servers on different machines. Using DVS, the client completed in about 90 seconds, about 2.5 times faster than a client using the local hard drive (figure 3). In both cases, the CPU time used by the client was around 30 seconds.

We also tested four BLAT clients running concurrently on four machines; each client each machine also runs a DVS server. The dataset was the entire human genome taking around 1.7Gb. The clients finished 2-3 times faster using DVS (figure 4). Note that the times are not comparable with the ones in the previous test, since the dataset is different.

To test the performance loss due to failures, we ran a benchmark with the 900Mb dataset, using four DVS servers. Each DVS server used 320Mb of memory; thus about 42% of the buckets were replicated. We split the query sequences into 14 groups; the results are shown in figure 5. First, we ran BLAT using a copy of the dataset stored on the local hard drive for reference, since the difficulties of the query groups varied. We also ran BLAT using DVS (no failures). We restarted the experiment and crashed a DVS server at the 5^{th} query group. The graph shows a peak at that group, showing the overhead of the rebalancing operations and RPC timeouts. After the peak, the queries operate at slower speed because a part of the data was lost, resulting in read faults; the NFS wrapper has to load pieces of data from the hard drive. Because the dataset does fit into the memory used by the three live servers, and the rebalancing operations maintain a good load balance, the read faults repopulate the lost data; subsequent runs, even with only three servers, operate at the same speed with the first DVS run (without failures).

The results of these benchmarks show that DVS provides a significant improvement over a local hard drive, which should be faster than a file server on the network, given that the clients read different parts of the dataset. However, the gigabit ethernet that we used for our benchmarks is a very good platform for DVS. We tested DVS on slower networks and the results were unpromising; on a 10Mbps network, DVS was 50% slower than the local hard drive, even with BLAT's seek-unfriendly reads. Tests using the 6.824 lab computers showed that even a 100Mbs network might not be fast enough for DVS to be a worthwhile investment.

5 Conclusions

We argue that many bioinformatics algorithms are disk bound because of sparse reads from a

large dataset. We present DVS as our solution to speed up reads given some restrictions on the dataset. The results of our benchmarks show that DVS is capable of significantly speeding up bioinformatics algorithms like BLAT. However, it is important to keep in mind that communication is the limiting factor in DVS and a fast network is required. Fortunately, gigabit networks are becoming increasingly popular while hard drive performance improves very slowly.

Acknowledgments. We thank Prof. Morris and Emil Sit for their guidance, feedback and help with technical issues regarding this project.

References

[1] Yasushi Saito, Brian N. Bershad, Henry M. Levy. Manageability, availability and performance in Porcupine: a highly scalable, cluster-based mail service. Department of Computer Science and Engineering, University of Washington

[2] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, July 1978.

[3] Flaviu Cristian and Frank Schmuck. Agreeing on processor group membership in asynchronous distributed systems. Technical Report CSE95-428, UC San Diego, 1995.

[4] Danga Interactive. *memcached*: a distributed memory object caching system. http://www.danga.com/memcached

[5] H. Lin, X. Ma, P.Chandramohan, A. Geist, N. Samatova. Efficient Data Access for Parallel BLAST. *IEEE International Parallel & Distributed Processing Symposium*, Denver, CO, April 2005

[6] A. Darling, L. Carey, and W. Feng The Design, Implementation, and Evaluation of mpiBLAST. *4th International Conference on Linux Clusters: The HPC Revolution 2003 in conjunction with the ClusterWorld Conference & Expo*, San Jose, CA, June 2003

[7] JW. Kent. BLAT - The BLAST-Like Alignment Tool. *Genome Research 12(4):656-664*, 2002