

Free Factories: Unified Infrastructure for Data Intensive Web Services

Alexander Wait Zaranek, Tom Clegg, Ward Vandewege, George M. Church
Harvard University
await@genetics.med.harvard.edu

Abstract

We introduce the Free Factory, a platform for deploying data-intensive web services using small clusters of commodity hardware and free software. Independently administered virtual machines called Freegols give application developers the flexibility of a general purpose web server, along with access to distributed batch processing, cache and storage services. Each cluster exploits idle RAM and disk space for cache, and reserves disks in each node for high bandwidth storage. The batch processing service uses a variation of the MapReduce model. Virtualization allows every CPU in the cluster to participate in batch jobs. Each 48-node cluster can achieve 4-8 gigabytes per second of disk I/O. Our intent is to use multiple clusters to process hundreds of simultaneous requests on multi-hundred terabyte data sets. Currently, our applications achieve 1 gigabyte per second of I/O with 123 disks by scheduling batch jobs on two clusters, one of which is located in a remote data center.

1 Introduction

We built Free Factories to help the PGx team win the Archon X PRIZE for Genomics and to meet the needs of the Personal Genome Project. The prize is awarded for sequencing one hundred complete human genomes in less than ten days [29]. Doing this with Polony sequencing [17, 25] and related technologies [30, 31], as the PGx team plans to do, will involve distilling many petabytes of raw data to produce about 100 gigabytes of output. This DNA sequencing capacity can be used to help build a database of personal genome-phenome data sets; coupled with a data mining and analysis engine, this will provide opportunities for many new discoveries.

Storing and analyzing data at this scale still requires exotic computing systems [3, 15, 28] – many scientists, physicians and members of the general public would like to participate in the development of these technologies,

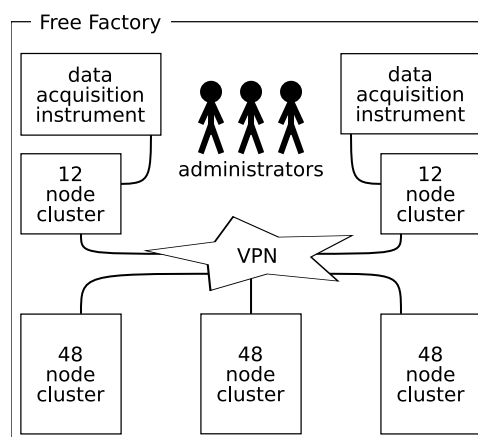


Figure 1: A Free Factory contains about five clusters of 12 to 48 nodes. Some clusters are colocated with data acquisition instruments; their sizes are limited by the available power, cooling, and space. Other clusters are located in data centers. The clusters are interconnected by relatively slow networks.

but do not have access to the resources they need to get started. Furthermore, anyone creating a database of sensitive personal information has to address privacy and disclosure concerns, and there is no single correct way to do that. The Personal Genome Project aims to overcome these obstacles and, ultimately, give individuals the tools to make genetic discoveries of their own [7, 19, 20].

To help alleviate these barriers, we consider the needs of a relatively small organization that supports several independent applications. Some such organizations support scientists in research activities, like developing new sequencing technologies; some focus on medical applications, like predicting which treatments will be effective for a particular patient. Within each of these communities, scientists and application developers generally benefit by sharing data and computing resources, although they may need to segregate some data and resources in

order to satisfy a particular security model. End users choose who to trust with their personal data, establish legal agreements with those trustees to control how their data is used, and access relevant information using web services.

With the Free Factories platform, we make the first steps toward this vision. We emphasize efficiency in a range of installation sizes, from a single 12-node cluster to several 48-node clusters. We cater to data-intensive applications that are conducive to parallel computation, but are limited by the ability of storage systems to support many concurrent tasks. We avoid proprietary software and expensive hardware. Organizations can start working with substantial data sets on small clusters, and expand their capacity by adding new clusters. Multiple small clusters also provide opportunities to enforce different privacy and data integrity guarantees for different applications and data.

Small installations of low-cost hardware provide processing and storage capacity at the scale we need for these applications, but efficient and fault-tolerant utilization of this hardware is non-trivial. We have used a pragmatic approach of selecting and arranging free software to make the best of the performance characteristics of cheap hardware. Our goal is to utilize 90% of our hardware capacity, including disk I/O bandwidth, network bandwidth, and CPU time. The result is a unified platform that makes efficient use of hardware in an environment where a variety of users and applications share storage and computation resources. We encourage others to deploy and develop this platform further. [16]

2 Design and architecture

A Free Factory provides hosting, data storage, and batch processing services for a number of web applications. These applications involve data-intensive computation: they are conducive to asynchronous parallel processing, but their performance is limited by the available disk I/O bandwidth. Their demands for CPU time are highly variable, so it is sensible for them to share a pool of CPU resources by submitting batch jobs. They also tend to share data sets with one another, so it is sensible to share a large data storage system. The application developers have common goals rather than being in competition, so it is beneficial to let them see the source code and results of one another's batch processing jobs. The applications themselves may be maintained by different development teams, so each application should run in its own independent virtual machine.

We identify the following roles in the Free Factory environment. "Users" – scientists, physicians, and members of the general public – are interested in a web service and interact with it via a web browser. "Administrators"

maintain the Free Factory infrastructure. A "trustee" sets policy and obtains funds to pay for staff, hardware, and hosting. "Developers" are the application developers and scientists who maintain Freegols. "Freegols" are web services that utilize cluster computing and storage resources. The term Freegol, or Free Golem, emphasizes the idea that the web services are developed and maintained independently of the cluster infrastructure, and independently of one another.

The canonical Free Factory (Figure 1) contains about five clusters with 12 to 48 nodes each. Some clusters can be co-located with data acquisition engines such as DNA sequencing instruments. Each cluster acts as a web hosting platform for several applications, as well as supporting the data storage and batch processing needs of those applications. A Free Factory of this size can be maintained by three administrators.

Each cluster is constructed using 1U rack-mount machines with big disks and inexpensive CPUs. Today, each of these low-cost machines offers about 240 MB/s of disk I/O bandwidth as well as 2 Gb/s of network bandwidth. With data and processing resources striped across an entire 48-node cluster with 192 disks, it is theoretically possible to achieve 11 GB/s of disk I/O during a batch job. The two clusters we have built contain 85 and 38 disks respectively. At 60 MB/s per disk this gives us 5.1 GB/s and 2.2 GB/s of available disk bandwidth respectively.

We use virtualization to deploy cache, storage, and batch processing services on every single node in each cluster: CPU-intensive jobs can make use of every CPU in the cluster, while data-intensive jobs can make use of every disk. This layout allows us to achieve high I/O throughput even while many concurrent processes are working on the same data set. We have achieved as much as 1 GB/s of I/O on a cluster with many concurrent processes; this compares favorably with a 12-disk RAID-6 system, which we have to limit to a single reader in order to achieve a sustained throughput of 100 MB/s.

On each machine, a "warehouse instance" runs the processes that implement cache, storage, and batch processing services. Warehouse instances are implemented as virtual machines on the nodes that are used for hosting Freegols, and consume entire physical machines in other cases. Each cluster manages its own cache, storage, and batch processing services using a number of controller processes that run in a virtual machine. Freegols and batch programs use the warehouse client library to communicate with these controller processes and the warehouse instances' service processes (Figure 2).

2.1 Commodity hardware, free software

When building an affordable, high-availability, data-intensive web service it is desirable to have control of the

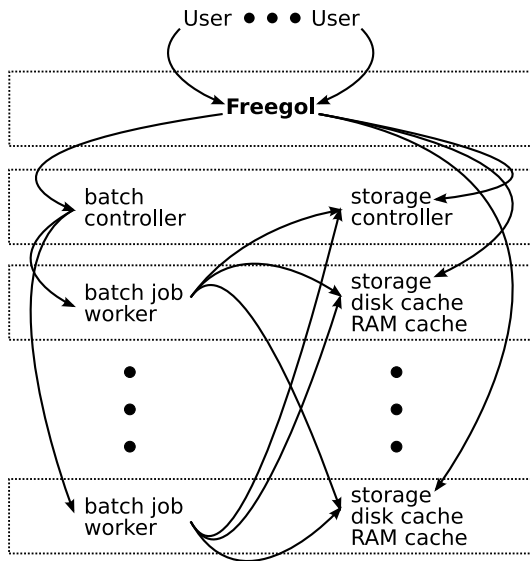


Figure 2: Dotted lines denote virtual machines. Batch processing workers in warehouse instances are dispatched by the batch controller on behalf of Freegols. RAM cache, disk cache, and long term storage services are accessed by Freegols and batch jobs using the same client library.

system’s total cost of ownership. Part of our strategy is to avoid proprietary technologies in favor of free software. This way, we can build on existing tools and have confidence that we can replace or modify them when necessary.

When choosing hardware, we are interested in maximizing the usable disk, RAM, CPU, and network bandwidth per unit cost. At present dual gigabit ethernet, one terabyte SATA disks, and dual-socket quad core motherboards seem to best fit our needs. (We prefer larger disks even at a higher cost per gigabyte because disks have high failure rates independent of size [2, 12, 23, 24] and manual intervention is expensive even in a fault-tolerant system.) Full-bandwidth 48-port switches are also available at low cost. Therefore, the most affordable way to configure a large number of disks and CPUs today is to build a number of 48-node clusters, interconnected by relatively slow network links or virtual private networks.

Given the limited size of each cluster, scalability requires that applications have access to more than just one cluster. We expose the network topology to the applications so that they can make informed decisions about where and when to perform computation, and where to store data, depending on the varying availability of these resources on different clusters. (Our intent is for future client libraries to help applications make good scheduling decisions; currently it is practical for a developer to select one of our two clusters when running a job.)

Each cluster is self-contained; hardware and network failures do not cascade to other clusters. The small cluster size makes it feasible to deploy entire clusters at once, rather than performing incremental upgrades to a large cluster. Multiple clusters can be used to increase confidence in the repeatability of computational results, and to monitor the effects of different combinations of hardware, software, and usage patterns.

To illustrate the cost of a Free Factory we consider purchasing a 48-node cluster with 192 1 TB disks for \$170,000. Annual operating costs include \$27,000 of power (18 kW at \$0.17/kWh), \$25,000 for network access and floor space (at the rate we pay at Harvard), and \$50,000 for a part-time administrator. Thus, the total cost to deploy the cluster is \$272,000 for the first year, and \$102,000 per year thereafter.

For a point of reference, we consider Amazon Web Services, a popular computing platform that allows an organization to pay for computation and storage on an as-needed basis. This is often less expensive than using dedicated hardware because the cost of processing is determined by average demand rather than peak demand. However, for the data-intensive applications we consider here, the strategy of frequently allocating and releasing compute nodes is less beneficial because of the time spent copying data to and from the nodes each time. In effect, a CPU-on-demand system requires a higher allocation rate in order to do the same work, compared to a dedicated hardware approach where data is kept close to the processors and can be read at full speed whenever it is needed.

We overlook this distinction for the sake of making a direct comparison with the Amazon EC2 and S3 services [1]. Amazon EC2 offers an “extra large instance” with two 1 TB disks and four virtual CPUs for \$0.80 per hour. Thus, a 48-node cluster is roughly equivalent to 96 extra large instances. If the cluster achieves 25% CPU utilization, its value is comparable to 24 extra large instances at \$168,000 per year. Meanwhile, S3 provides long term storage for \$0.15/GB. At this rate, it costs an additional \$43,000 per year to store 24 TB of data (half of the long term storage capacity of the cluster). The actual amount of data transferred to and from S3 depends on the application; if 15 TB is transferred to Amazon at \$0.10/GB, and 2 TB is transferred out of Amazon at \$0.17/GB, then the transfer cost is \$1,840 (traffic between EC2 and S3 is free). The total cost of the Amazon service over two years is \$424,000, compared to \$374,000 for the first two years of a Free Factory.

2.2 Freegols and virtualization

There are a wide variety of languages, toolkits and methodologies for deploying scalable web services. One

factor that contributes to Amazon EC2's popularity is that it permits web service developers to choose their own tools. We found that giving developers this freedom suited our environment too.

Virtualization encourages a model wherein developers have "root" privileges on their own virtual servers, or Freegols. Typically, a Freegol is configured as an Ubuntu server with common application server software like Apache and MySQL. The warehouse client library is easy to install and upgrade using the native package manager. Our goal is to make it easy for developers to start using Freegols to deploy services; part of this strategy will be to port the Perl client library to other popular languages like Python.

RAM, virtual processors, and network bandwidth are shared among Freegols, cache and storage service processes, and batch jobs. If necessary, a developer can ensure that a Freegol does not share these resources with other Freegols by getting an allocation for all of the available CPU and RAM.

In addition to Freegols, it is often beneficial to set up virtual machines on the cluster for applications that do not use the cache, storage, or processing services. Cluster administrators are likely to use virtual machines to deploy common network services like web proxies, DNS caches, and backup servers.

2.3 Cache and storage services

The objectives of our storage services are to: (1) minimize I/O bottlenecks in order to make the best use of available CPU cycles; (2) provide a low-latency shared cache with automatic garbage collection; (3) provide long term storage with high read and write throughput and provisions for usage accounting. The storage services must yield good performance when used directly from Freegols, as well as from batch jobs. Inspired by the Google File System [13], Bigtable [6] and the plethora of raw materials made possible by free and open source software, we felt we could build a system that suited our needs perfectly.

We implement a three-level content addressable storage service. We use Memcached [9] as a low latency RAM cache. This is well suited to small strings (less than one megabyte) and it works well even with many concurrent clients because it does not employ a central controller. We use MogileFS [9] to implement a cluster-wide distributed disk cache. This gives good performance for block sizes up to 64 MiB. For long term storage we have developed software that minimizes the role of a central controller while providing opportunities for usage accounting. All of these storage and cache services are accessible to all applications and batch jobs in the Free Factory.

Aggregate I/O bandwidth is limited by several factors. Our storage services are designed to minimize the effects of these factors.

1. Disk seeks reduce aggregate disk read and write bandwidth. We minimize seeks by storing data in contiguous 64 MiB segments when possible, and ensuring that each segment read/write operation is not interrupted by any other disk activity. This means that readers tend to wait longer before they start to receive data, which is why general purpose operating systems do not use this strategy; however, in this environment, high throughput is more valuable than low latency.
2. A gigabit network interface can only handle two concurrent readers at full disk transfer rates. To prevent this from limiting throughput when many concurrent processes are accessing the same data set, we stripe every data set across all of the cluster nodes.
3. Central controllers get bogged down when they try to handle too many concurrent clients. Our storage service can read and write blocks without involving the storage controller in real time. Our disk cache is indexed in RAM, so most reads do not involve the controller.
4. Writes are slow because robustness requires storing multiple copies of each block. When an identical copy of an output block already exists in the storage service, our content-addressing approach allows the client library to transparently skip unnecessary write operations.

The cache and storage services use an MD5 addressing approach: the name of each block of data is the MD5 checksum of the data. This naming scheme provides several benefits.

1. The client library, when retrieving a block from the cache and storage services, computes the MD5 checksum of the data and compares it with the block name. If the checksum does not match, the client library can try to fetch the data from a different host, a different storage service, or a remote cluster. This verification process is completely transparent to the application.
2. Multiple jobs often produce the same output. For example, a Freegol may re-run a job every time the job's source code is modified in the source repository, and every time an operating system upgrade is performed, to make sure the job still produces the same result. If the output is identical to the previous run, no additional storage space is consumed

by the new job. We encourage developers to make use of this property by segregating their main output from their diagnostic messages (which are likely to contain timestamps and the like, but are usually small) and by avoiding non-deterministic outputs (sometimes this involves simple tricks like using the “Minimal” option in the `IO::Compress::Gzip` Perl module).

3. Freegols and concurrent batch jobs can share data without worrying about overwriting blocks at inopportune times.

Our simple tests have demonstrated that reading 64 MiB files sequentially results in throughput exceeding 90% of a disk’s maximum sustained transfer rate. Real data used by Freegols, on the other hand, is likely to include many smaller files. To help Freegols achieve high throughput when working with smaller files, we introduce a “manifest” file format. In addition to increasing performance, the manifest format is a valuable tool for managing large data sets.

A manifest is an index to a collection of data files, analogous to a directory tree in a traditional filesystem. It is stored as a plain text file. Each line of the text file represents a “stream”; each stream contains a set of data files. The content of the data files is stored in a manner similar to a UNIX `tar` archive: the data from all files in a stream are concatenated, the result is split into 64 MiB blocks, and the blocks are written to cache or storage. The manifest file specifies the MD5 checksums and sizes of these data blocks, as well as the names of the individual files and their positions within the stream. The manifest file itself can be split into 64 MiB blocks and stored, and its unique key – the list of MD5 checksums of those blocks – can be used to retrieve it. (If this list of checksums is inconveniently long, the list itself can be stored in a separate block, whose MD5 checksum then serves as a more concise key to the large manifest.)

This manifest format has several noteworthy features. It is concise: a short key is enough to specify a large collection of data. It is portable: if two jobs running on different clusters produce identical output, the resulting manifest keys are also identical. The integrity of the data blocks, and the manifest itself, are easily verified. It is efficient to read an entire stream worth of data from disk, even if the stream represents many small files. However, random access – reading and writing small files in various streams out of order – is not efficient. We expect applications to be cognizant of this restriction, and read and write entire streams whenever possible.

Once a manifest is written to the cache or storage service, it can be retrieved, or used as the input to a batch job, by any Freegol that knows its key. Also, each cluster has a central database of manifest names. To attach a

name to a manifest, a Freegol sends a request to the cluster’s storage controller specifying the manifest key, the desired name, and – to avoid race conditions – the manifest key that was previously associated with the name. Naming a manifest has several consequences.

1. Any Freegol can look up the name to retrieve the manifest key.
2. The data set is considered to be valuable to the signer. If the blocks referenced by the manifest are in long term storage, those blocks should not be deleted.
3. The old manifest, if it is not associated with any other names, is no longer considered valuable; the data blocks mentioned in it may be deleted if they are not mentioned in any other named manifests.

Optionally, a Freegol may also specify a list of PGP keys indicating entities that have permission to overwrite this manifest name.

One drawback to content-addressable storage is that in-place updates are inefficient. For example, if a 32 MiB stream is written, and a new version of the stream is written afterward that has 16 MiB of additional data appended to the original 32 MiB, then both the 32 MiB version and the 48 MiB version may be written to disk. We find this acceptable for the following reasons. The manifest format allows the 48 MiB stream to be stored by referencing the original 32 MiB block followed by the new 16 MiB block, if the existence of the 32 MiB block is known when the second version is written. In any case, we are willing to sacrifice some storage space in order to avoid race conditions.

This illustrates one of the useful aspects of the manifest format. A manifest key specifies the data itself, not just a set of filenames. If a modified version of a large data set appears, the previous key can still be used to access the old data, and a program requesting the original version will never unexpectedly receive the newer data. This simplifies application design, and provides a significant practical benefit for scientific applications and other environments where repeatability is of major concern.

Periodic garbage collection is inexpensive. The storage controller can quickly read all of the manifests that appear in the manifest name database, and produce a list of blocks that are still in use. The 64 MiB block size ensures that the resulting list is small compared to the amount of stored data. Garbage collection is accomplished by comparing this list against the list of blocks stored on disk. We have not yet implemented automatic garbage collection but we have found that a list of 588,000 distinct block names, representing 11 TB of data referenced by 1151 distinct manifests, can be generated in 70 seconds.

2.4 Batch processing services

The objectives of our batch processing services are to: (1) use as many as possible of the available CPU cycles on all machines; (2) make it easy to repeat jobs many times on various clusters to check for bugs and inconsistencies; (3) handle occasional failures gracefully; (4) keep statistics about performance and failure rates.

Batch processing is coordinated by a batch controller on each cluster. The batch controller accepts requests from Freegols to schedule new jobs. The batch controller starts new jobs when the requested number of warehouse instances become available. Freegols can expect the batch controller to occasionally pause and resume a job, or reduce its resource allocation, depending on subsequent job submissions. (Our current implementation uses a simple greedy scheduling algorithm, and the batch controller only pauses and resumes jobs when specifically requested by a Freegol.)

Freegols can retrieve a list of current, pending, and previous jobs from the batch controller. This list includes specifications and statistics for each job, including inputs, outputs, start and finish times, and (for active jobs) what portion of the job has been completed so far. Freegols can poll the batch controller to determine the status of their own jobs, get hints about how busy the cluster is, and look up details of jobs that other Freegols have submitted.

The execution of a batch job is supervised by a job manager process running on the same virtual machine as the batch controller. The job manager supports a computation strategy similar to MapReduce [11]. Each job consists of a number of steps, each of which is performed on a single warehouse instance. Each job step stores some output in the cache; the job manager assembles the output into a manifest at the end of the job. Additionally, each job step has the ability to enqueue more job steps.

The program that performs the work of a single job step is called a “mr-function” (from “MapReduce function”). Mr-functions are kept in a revision control system. Administrators and developers can update existing mr-functions and create new ones, subject to access controls on the revision control repository. Once it is committed to the repository, a mr-function can be used in a job submission by any Freegol.

The most convenient way to construct a batch job is to use a single manifest as input, and schedule one job step for each stream in the manifest. Each job step reads one full stream from the input manifest from start to finish, and writes one full stream in the output manifest. The client library comes with tools and examples to make it easy to write mr-functions that use this strategy.

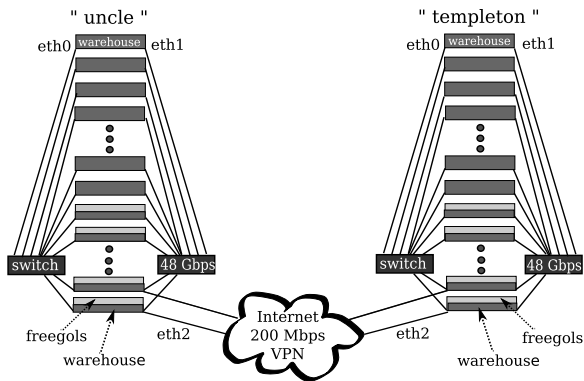


Figure 3: On each cluster, a few physical machines run the Xen hypervisor (mixed dark/light gray blocks). These machines are partitioned into warehouse instances, Freegols, and other virtual machines. Other physical machines are dedicated to warehouse instances (dark gray only).

3 Implementation

3.1 Commodity hardware, free software

We are currently operating two clusters using a variety of commodity off-the-shelf hardware, free software such as GNU/Linux [14] and our own custom software that is released under the GNU GPL [16]. Our two clusters are located a few kilometers apart and connected by Harvard University’s fiber optic network. “Uncle” is our experimental research cluster; “templeton” is our production cluster. Our clusters are depicted in Figure 3. Uncle is largely made up of 32-bit dual-CPU Intel Xeon machines, many of which are four years old. Templeton’s hardware is more recent: each machine has two dual-core AMD Opteron 64-bit processors.

Each cluster consists of 47 machines. All newer machines have four disk slots and the older machines are diskless. Each newer machine has two gigabit ethernet ports, which are connected to two 48-port gigabit switches. Two to four “headnodes” have a third ethernet port connected to the upstream switch, and optionally a fourth port connected to an out of band management network. The headnodes act as gateways to the internet and as VPN endpoints.

Our VPN is a simple OpenVPN [22] point-to-point setup, terminated on a headnode at each end. The VPN data rate reaches about 200 Mb/s. Throughput is limited by the processing speed of one endpoint that has one dual-core Opteron 265 CPU at 1.8 GHz. The other endpoint has 2 single-core Opteron 250 processors at 2.4 GHz; CPU load is considerably lower there.

Uncle currently runs the latest release of Ubuntu [5], a popular flavor of the Debian [27] GNU/Linux system that many lab members run on their desktops, while

Templeton runs the latest “long term support” release of Ubuntu. We chose Ubuntu over Debian because of Ubuntu’s predictable six month release schedule, but we expect to build a Debian cluster in the future. We like the Debian and Ubuntu philosophy and have found that packaging our software using Debian/Ubuntu tools to be a good way to automate the installation of the client library and its dependencies. We believe that inclusion in major community projects, such as Debian, is an excellent way to both reach a wider audience and to further improve our installation automation. Ultimately, we aim to be distribution agnostic.

The latest machines that we purchased came pre-installed with coreboot [8], a free software BIOS that has a number of advantages over proprietary alternatives. All source code is open and available under the GPL. Serial console support is reliable. Boot time is much faster: coreboot takes only a few seconds to bring the machine into a state where it can start booting the operating system. Also, we can exactly configure the platform to our needs, which allows us to make the platform both more reliable and more secure.

We use Opegear [21] CM4148 console servers on each cluster for out-of-band access to the serial console of each physical machine. The Opegear console servers are embedded Linux machines for which the entire source code is available for download. The company provides instructions for modifying the firmware, and for building the firmware from source. We also use networked power distribution units to allow remote power cycling of any device in the cluster via our out-of-band management network.

Aside from the Linux kernel, our software relies on many other open source packages. Notably, Slurm and Munge [18] provide authenticated inter-process communication between the warehouse instances and the batch controller. We rely on Slurm to track which warehouse instances are available for running jobs, and to allow the batch controller to execute batch job steps on the warehouse instances. It does this well, with low latency. It is also a convenient tool for administrative tasks like installing packages and updating configuration files on many instances at a time.

Memcached and MogileFS [9] provide the RAM and disk cache services respectively. MogileFS provides distributed storage with low-latency replication. Used in conjunction with Memcached, it performs well as long as there are not too many concurrent writes.

Perl modules from CPAN are used by the client library, controllers, and service programs for HTTP request handling, data compression, and MD5 hashing. The batch controller uses a MySQL database as a job queue and an archive of past jobs. Subversion provides revision control for the programs that run in the batch processing sys-

tem, as well as the client library and the service software itself.

3.2 Freegols and virtualization

Some physical machines are configured as warehouse instances, dedicated to providing processing, cache, and storage services. Others are partitioned into virtual machines using the Xen hypervisor – always with one virtual machine configured as a warehouse instance, along with one or more Freegols and other virtual machines controlled by the cluster administrators.

To keep configurations simple, we set aside 4 GiB of RAM on each warehouse instance for use in batch jobs, and allocate the remainder to Memcached processes.

On virtualized machines and dedicated warehouse instances alike, we use RAID-1 to protect all of the local filesystems. We have found that dedicating two entire disks to RAID-1 results in an excess of RAID-protected space. It is wasteful to allocate that space to the storage services, which can already accommodate disk and node failures without RAID-1. Linux allows us to partition the first two disks, assign one partition on each to a software RAID-1 array managed by the Linux Volume Manager, and allocate the remaining space to the MogileFS cache. This is more efficient than whole-disk mirrors, but it still forces us to commit to a partitioning scheme early on. Different permutations of Linux RAID and volume management tools could give us greater flexibility, but we have chosen to avoid the extra complexity that would result. In the future we hope iSCSI will provide more flexible options without creating too much work for administrators.

Developers, and some of their users, have access to shell accounts on their Freegols. Our security model is largely perimeter-based at this point, because our users are relatively trustworthy. Specifically, a cluster has one virtual machine with an unprivileged account that is shared by all users. To connect to the SSH port on a Freegol, a user must log in to this shared account using an SSH private key, and specify the name of the Freegol in a remote command string. The `authorized_keys` file in the shared account instructs the SSH server to ignore the client-supplied command and instead run a script that establishes a tunnel to the requested Freegol (the SSH server makes the supplied Freegol name available in an environment variable). Before establishing the tunnel, the script checks a list of permitted combinations of SSH public keys and Freegols. This login procedure is easy to express in a `ProxyCommand` directive in the user’s SSH client configuration file.

Virtual machines are used for deploying DNS caches and servers, an SMTP server for routing incoming mail, and a local Ubuntu mirror site. One virtual machine runs

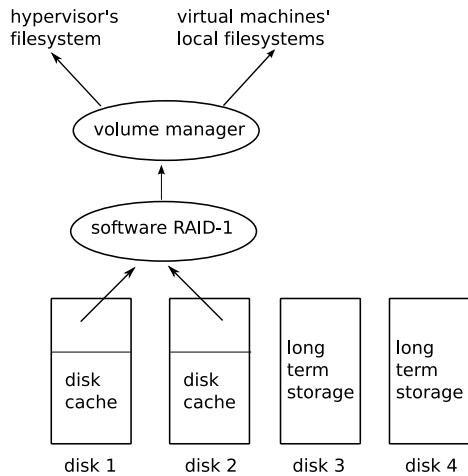


Figure 4: On a physical machine that is running several virtual machines, a software RAID-1 volume is exported to the Linux logical volume manager, which provides space for local filesystems. The remaining portions of the first two disks are used by the disk cache service. The third and fourth disks are dedicated to the cluster’s long term storage service.

a dedicated reverse HTTP proxy. All HTTP and HTTPS traffic to the cluster is forwarded by the headnode to this proxy, which forwards each request to the appropriate Freegol and returns responses to the clients.

Since Harvard routinely receives high volume web traffic, we tested whether our setup can also support a high volume web service. We configured the virtual machine running our reverse proxy server with moderate resources: four shared Opteron 265 cores and 1 GiB RAM. We then launched 2,000,000 requests at the proxy server (with keepalive disabled to simulate a worst-case scenario), with 500 simultaneous requests each coming from two physical machines on our LAN. All requests were handled without errors and completed in 657 and 660 seconds on client 1 and 2 respectively. The mean time to complete a request was 0.3 seconds. Both clients completed 90% of requests within 0.2 seconds, and 99% of requests in 4.5 seconds. The slowest request from client 1 was completed in 93 seconds while the slowest request from client 2 was completed in 107 seconds. During this test, a few of us continued to use other Freegols from multiple points on the Internet and found no perceptible difference in response times. We conclude from this that a Freegol on our cluster can withstand popularity spikes without adversely affecting other Freegols.

The cache, storage, and processing services involve several databases and controller processes, which we run on a single virtual machine. It would also be possible to distribute these processes across several virtual machines; we have not thoroughly explored the perfor-

mance implications of this choice.

Developers frequently benefit from having separate “development” and “production” Freegols for a given application. The virtualization approach makes it easy for us to deploy these quickly at minimal cost.

3.3 Cache and storage services

The cache and storage services consist of three software layers (RAM cache, disk cache, and long term storage), a manifest name server, and a client library that is used by Freegols and batch jobs.

The client library contains most of the intelligence. It helps applications split and combine data into 64 MiB blocks; it chooses RAM or disk cache for different block sizes according to tunable settings; it constructs hashes when storing data; it avoids writing blocks to cache if they are already present; it stores each block on multiple warehouse instances when writing; and it constructs and parses manifests.

The client library is built on the assumption that all of the underlying storage services are unreliable. It verifies data integrity during retrieval operations. It attempts to retrieve blocks from alternate sources when data is corrupted or missing.

The aim of the storage services is to maintain the highest possible aggregate throughput when reading and writing blocks.

Our RAM cache uses Memcached. The Memcached client library and servers implement a distributed shared-nothing hash table. Requests for a name are mapped to weighted buckets by the Memcached client library, and then fulfilled by the Memcached server. We use the default library, which permits rehashing of blocks in the event that a server becomes temporarily unavailable, but requires that the entire cache be flushed if bucket sizes or the number of nodes change. In our system, such cache flushes should not be common because we deploy clusters with a fixed number of nodes by design. Memcached has a built-in limit of 1 MB for each data block, and our client library uses this as the default maximum size for cached items; however, if the application specifies a larger limit, the client library transparently splits larger blocks in 1 MB chunks when storing, and reassembles them when retrieving them.

The disk cache service is implemented using MogileFS. MogileFS uses a MySQL database to store the locations where each file is stored. This single database quickly becomes a performance bottleneck when it is accessed for every cache read and write operation on the entire cluster. Our client library alleviates this problem by using Memcached to cache the file locations: as a result, most read operations do not involve querying the MySQL database.

For long term storage, we implement a simple service called Keep. Each warehouse instance with long term storage space runs a network server that accepts HTTP “GET” and “PUT” requests. The client library is responsible for replication, fault tolerance, and load balancing as described below.

A “PUT” request is a signed request to copy data from cache to long term storage. The Keep server fetches the data itself from the local cache or from a remote cluster, according to the hints that come with the request. This approach is very efficient in the case where applications first store a lot of data in the disk cache, and later choose to keep some of that output in long term storage. This case has turned out to be so common that we have not yet implemented an operation that writes directly to long term storage.

When a “PUT” request results in a disk write, an accounting entry is also recorded, with the requestor’s IP address and cryptographic signature. Currently, the server does not verify the signature because this has not been necessary in our environment, but it does demand that each request arrive in the form of a PGP signed message. It also records a timestamp and the full text of the request; and it requires that the message include a timestamp within 5 minutes of the server’s system clock. We do not expect the overhead of checking signatures to cause an inordinate performance burden because of our large block size.

The client library uses MD5 checksums of data blocks to distribute them evenly among the available Keep servers. First, we specify that a given cluster has a fixed number of servers (there should be one on each cluster node). For a given block of data, we define eight preferred storage positions, derived from eight substrings of the block’s 128-bit MD5 checksum: portions of the checksum are used to compute a list of eight different Keep servers, and these Keep servers are tried one by one until enough copies have been written.

When a block has been written to Keep, the client library notes which of the eight preferred Keep nodes were used, and encodes this information as a hexadecimal number, where the least significant bit corresponds to the first preferred storage position. The block location is given as the letter “K” (from “Keep”) followed by the hexadecimal number, the symbol “@”, and the name of the cluster. For example, if a block is stored on the templeton cluster using the second and third servers in the probe order, the list of locations is encoded as “K06@templeton”. This string, along with the size of the block (an unusually short 3 byte block in this case) are appended to the MD5 checksum using the delimiter “+”:

```
acbd18db4cc2f85cedef654fccc4a4d8+3+K06@templeton
```

This resulting block name, stored in a manifest, provides enough information for a Freegol or a batch job to retrieve the block, regardless of which cluster it is stored on.

This notation provides an opportunity to support other storage systems – for example, “S” might designate blocks stored using Amazon’s S3 service – and to cite multiple clusters and storage systems, when a block is stored in multiple locations.

A “GET” request is much like a request for a static HTML page: the MD5 hash provided in the client’s request is the name of the disk file where the data is stored on the server. A Keep server may have up to four disks available for long term storage; in that case, it may have to perform four directory lookups in order to locate a file.

The storage controller maintains a database of manifest names and keys in a MySQL database. Any Freegol can connect to the storage controller’s “warehoused” network server program and retrieve the key currently associated with a given name, or the entire list of names and keys. A Freegol can also submit a signed request to update the database by changing the key for a given name, or adding a new name. Currently, the “warehoused” program does not verify signatures of these requests because this has not been necessary in our environment, but it does demand that each update request arrive in the form of a PGP signed message, and that the Freegol correctly specify the key currently associated with the name.

The client library includes functions for reading and writing blocks to the cache and storage services. The library also provides convenient functions for constructing manifests while storing data, reading streams and individual files from an existing manifest, looking up manifest keys by name, and updating the name database. Command line tools are provided for submitting jobs, looking up details of current and completed jobs, reading and writing data sets, and copying data sets from one cluster to another. These programs also serve as examples of how to use the client library.

The warehouse client library makes it convenient for Freegols and batch jobs to read and write data on remote clusters as well as the default local cluster. We provide this flexibility – rather than requiring data to be explicitly copied from cluster to cluster using a separate mechanism – for several reasons. It allows applications to achieve various levels of data redundancy, either synchronously or asynchronously, according to their needs. It supports the convenience of running jobs on remote clusters without arranging for all of the required data to be copied ahead of time. Generally, it follows the trend of doing a reasonable thing by default while offering the flexibility to accommodate the diverse needs of a variety of applications.

3.4 Batch processing services

The “warehoused” program accepts signed job submissions from Freegols, and answers queries about previously submitted jobs. The “mapinit” program starts queued jobs when instances become available, using Slurm’s `salloc` command to reserve warehouse instances. At the start of each job, the “mrjobmanager” program first retrieves the specified version of the appropriate “mr-function” from a Subversion repository, then invokes it on one of the allocated warehouse instances. The mr-function examines the input object (normally a manifest) and instructs mrjobmanager to queue a number of job steps.

During the course of a job, mrjobmanager allocates individual job steps to warehouse instances, monitors their output and exit codes to detect failures, and re-queues them when they fail. Each job step is expected to store some output blocks and send the blocks’ names to mrjobmanager by printing them to its standard error file descriptor. When all job steps have completed, mrjobmanager reads these blocks and assembles them into a final output stream. This final output stream is expected to be a manifest, although this is not enforced. Finally, the database table is updated to reflect the output key (i.e., a list of output blocks) and the time when the job finished.

While a job is running, mrjobmanager keeps the job table updated with the number of job steps in progress, finished, and remaining. These figures can be retrieved by any Freegol from the batch controller, and displayed to users as a progress indicator.

Order of execution and output assembly is controlled by job step numbers and level numbers. Step numbers begin at zero and are assigned sequentially by mrjobmanager; in the final stage of mrjobmanager the step numbers determine the order in which the job steps’ output fragments are assembled into the final output stream. Level numbers can be used by mr-functions to control the order in which job steps are scheduled: a job step with level L will never begin until all job steps with level less than L have completed. Each job step can also be given a short input string; this can be used by the mr-function to keep track of which portion of the job each step is expected to compute, in case the job step number itself is not convenient for that purpose.

Typically, a mr-function completes step 0 by reading its input manifest and submitting one new job step for each stream in the manifest. Each of these job steps will read the input stream data, write output data in the form of a stream, store the stream description (one line of the manifest) as a short block in the cache, and report the hash of this short block to the job manager. When all job steps have finished, mrjobmanager looks up all of the individual job step hashes and assembles them, ordered by

job step number, into one final output manifest. This final assembly step is inexpensive because it only involves lists of hashes and filenames; the job manager does not read or write any of the output data itself.

4 Applications and results

4.1 Freegols

We have implemented a few sample applications that demonstrate the warehouse client library and allow us to characterize performance.

The Genomator application is a storage/publication service. It currently allows users to browse and download images from a 300 gigabyte PMAGE data set [17]. In interactive mode, it converts images from TIFF to JPEG format and applies an ImageMagick “normalize” operation to increase contrast. This does not involve any batch processing, and the data set could have easily fit on a single disk; however, implementing the service as a Freegol gave us features like mirrored disks and scalability using existing hardware and staff resources.

The Regol application continuously re-schedules previously completed batch jobs when the cluster is idle, using the same inputs and parameters but substituting the current revision of the relevant mr-function. This helps us notice bugs as they are introduced into the source code repository. It is also a good source of information about performance characteristics of mr-functions, hardware configurations, and resource usage patterns. Regol is deployed on our “templeton” production cluster and is able to view and submit jobs on both clusters.

The Administrator web interface provides a generic job submission and monitoring interface. Users can select a mr-function and revision number, choose an input manifest from the list provided by the storage controller, and specify tunable parameters specific to the mr-function. Each of our clusters has its own Administrator web interface.

4.2 Mr-functions

The mr-functions we have implemented are generally concerned with problems in bioinformatics, specifically low cost DNA sequencing. Here we present some simpler applications that give a rudimentary illustration of the platform. Timing results from two of these can be found in Figure 5.

“mr-pivot-images” reads a manifest with F images (one per frame position on a slide, typically about 2000) in each of C streams (one per imaging cycle, typically about 75) and outputs a manifest with C images in each of F streams. The structure of the input data is determined by the image acquisition process; the output struc-

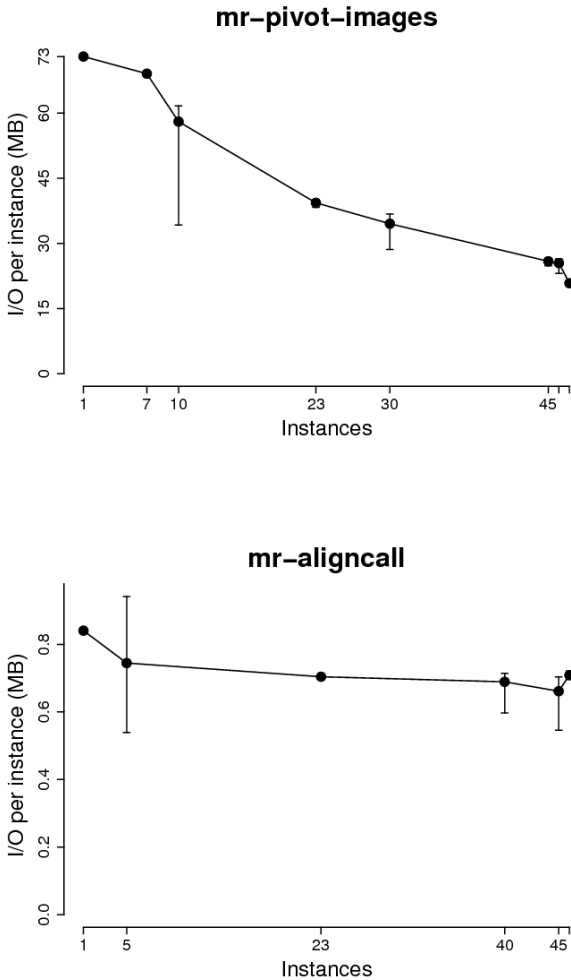


Figure 5: Timing figures from the templetion cluster show the effect of concurrency on per-process I/O speed. Mr-pivot-images shows diminishing returns as more instances are allocated: the data is read from the same set of disks regardless of allocation, and job speed is limited by the disk cache service’s ability to handle many concurrent readers. In contrast, mr-align-call’s performance is nearly linear with larger node allocations.

ture is suitable for image alignment and analysis. This is a relatively inefficient operation because each of F job steps reads pieces from each of C streams. It performs 3.3 terabytes of I/O on our 300 gigabyte input data set. We will certainly want to optimize this if we intend to use it frequently; meanwhile it provides a convenient way to measure a cluster’s performance under heavy I/O load.

“mr-aligncall” reads each stack of images produced by “mr-pivot-images”, analyzes and compares the images according to tunable parameters, and outputs short segments of DNA sequence as strings of A, C, G, and T characters.

“mr-zhash” uncompresses its input (if compressed), computes hashes for individual files, and outputs text files similar to the output of the Linux “md5sum” command line tool. It is useful for determining whether two compressed data sets are equal when decompressed.

“mr-copy” writes a copy of a data set read from a remote cluster (to make subsequent computation faster) or from the local cluster (to verify that all data is readable and passes checksum verification).

As the “mr-pivot-images” example suggests, we have found that the speed of an I/O-limited function is highly dependent on how closely the mr-function’s operation corresponds to the way the data is arranged in the input manifest. Ideally, each step in a batch job reads all of its input data from a single stream, in the exact order in which it is processed. In the case of “mr-aligncall”, this is easy to achieve because 2000 stacks of 75 2 MB images are processed by 2000 independent job steps. In contrast, for subsequent stages of our DNA sequencing applications, we spend much of our effort finding efficient ways to perform operations that are conceptually similar to “mr-pivot-images”. In this sense, the manifest format is an expression of the performance characteristics of the storage service: if we design our workflows to cater to the structure of manifests, then we get the best performance from our system.

4.3 Storage tools

We use two command line tools to move data back and forth between the storage service (or cache) and a Free-gol’s local filesystem. “whput” copies a UNIX filesystem tree to the cache, stores the resulting manifest in the cache, and optionally attaches a name to the manifest via the storage controller. “whget” fetches a manifest from storage, downloads the blocks, computes the MD5 checksums of the individual files, and optionally writes the files to the local filesystem. These tools – and the warehouse client library in general – can be used from any host with access to TCP/IP ports on the warehouse instances, such as an administrator’s workstation.

“whget.cgi” provides a web interface to the con-

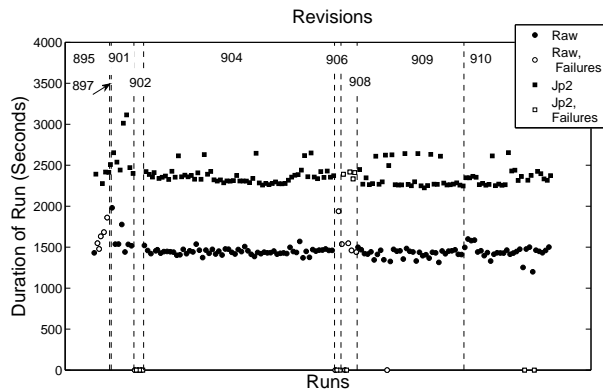


Figure 6: “Regol” helps us notice problems by re-submitting selected jobs when the cluster is idle. This graph shows two different jobs being repeated. Solid shapes indicate successful jobs. Hollow shapes indicate failures. Dashed lines indicate commits to the Subversion repository. If a string of failures begins at a vertical line and ends at another vertical line, it is most likely that the failures were caused by an errant commit.

tents of a manifest. It allows users to view filenames and sizes, click individual files to download them, and download the manifest file itself. Along with Apache’s `mod_rewrite` module, this makes it easy for a developer to selectively publish individual data sets.

4.4 Performance and reliability

In Figure 6 the repeatability of two jobs, “mr-raw” and “mr-jp2”, is illustrated. These jobs are running simultaneously; they perform approximately one terabyte of input and output in each pair of runs. The “jp2” job uses the lossless JPEG2000 format to compress the image data and is CPU intensive. The “raw” job is primarily limited by I/O. By inspection it is clear that the cluster achieves more than 400 MB/s of I/O: 1 terabyte in total, divided by 2500 seconds for the slower job.

To further explore the aggregate I/O and computational capacity of both clusters, we ran a selection of “mr-zhash” cryptographic hash functions concurrently with the “mr-pivot” function described above. The input to mr-zhash is compressed data; in each job, the amount of data processed is over 100 times the amount read from cache. This mixture of computation-intensive and I/O-intensive work was repeated over a 16 hour period, using 42 instances on each cluster. Over the 16 hour period, “mr-zhash” processed 102 TB of uncompressed data (1.5 GB/s on 12 templeton instances, 380 MB/s on 12 uncle instances) while “mr-pivot-images” performed 74 TB of I/O (1 GB/s on 30 templeton instances, 290 MB/s on 30 uncle instances).

In the above test, we ran “mr-zhash” in sets of twelve

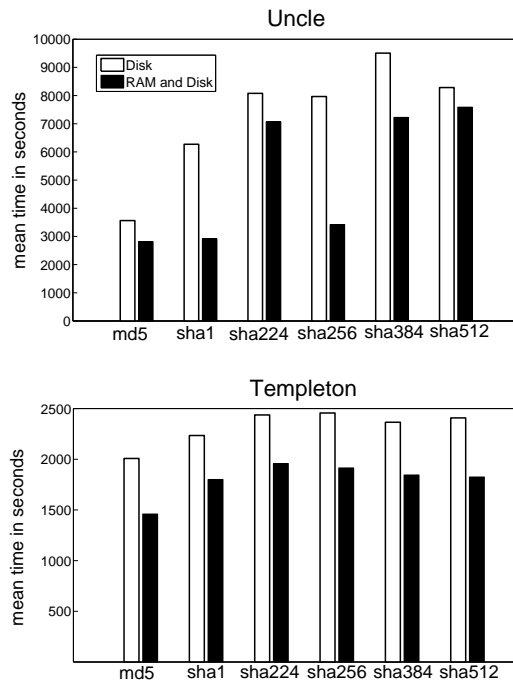


Figure 7: Mean time to read, uncompress, and run various checksum algorithms on 5.5 GB of compressed images, using an allocation of one warehouse instance per job on a busy cluster. Black bars represent jobs with the default client library configuration; hollow bars show the effect of disabling the RAM cache. Execution time is less predictable on uncle: some nodes are much slower than others, and the allocation of nodes to jobs is not random, so we see artifacts in the first graph. Templeton’s hardware is faster and more uniform; this is reflected in the second graph.

concurrent jobs – for each of six hash functions, one job with the RAM cache enabled, and one without. The results are shown in Figure 7.

Many of our design features are responses to lessons we have learned while using our two prototype clusters. For example, because we allocated space for cache on all of our disks before deploying the storage service, the storage service still shares disks with the cache. As a result, the storage service cannot serialize disk accesses. Our disk cache itself was deployed by adding a few disks at a time. This resulted in a poor distribution of data. The manifest used in the “mr-pivot-images” example has 50% of its blocks stored on only 22 disks. As we discussed above, performance suffers when a small number of disks are accessed by a larger number of concurrent processes. It is also noteworthy that even with four concurrent readers on one node, we currently achieve only 75 MB/s of I/O. We have verified with the UNIX `find` program that blocks read in 64 MiB chunks can be

read from the file system at 60 MB/s. We have verified with the `iperf` program that our network can sustain 100 MB/s. Although we have only reached 75% of this limit, rather than the 90% we set out to achieve, we look forward to deploying a new cluster with a properly balanced disk cache and segregated disks for storage. Even if the storage service never surpasses 75 MB/s per node, we are confident that we can achieve 4-8 GB/s of aggregate I/O.

4.5 Utilization

Between November 2007 and March 2008 we completed 460 million seconds of computation (18% utilization) on templeton and 600 million seconds (24% utilization) on uncle. During this time we accumulated 30 TB of data in cache on templeton (consuming 60 TB, 98% of the disk space allocated to the cache service) and 10 TB of data on uncle (consuming 20 TB, 73% of the space allocated). The two clusters consist of new and old hardware costing \$150,000 in total. Annual costs include \$25,000 for floor space, power, cooling, and network service, and about \$50,000 for staff costs.

If we had paid for this CPU time on a per-second basis at the rate charged by Amazon's EC2 for comparable instances – \$0.80 per hour for an extra large instance per node, \$0.20 per hour for two small instances for each of the 36 older uncle nodes – this would cost \$96,000 per year. Storing an average of 20 TB of data for the duration would cost an additional \$36,000 per year, at the Amazon S3 rate of \$0.15 per GB per month.

We can also consider the cost of the time spent copying data between S3 and EC2. Amazon does not specify the usable bandwidth between S3 and EC2, but if we assume that it is a very low 2 Gb/s, it costs \$41 to keep 47 EC2 nodes active while copying 1 TB of data from S3 to EC2. If our 25% utilization rate comes from working on 1 TB of data for one day every four days, the annual transfer cost is less than \$4000 per cluster. This cost is even lower if the available bandwidth is more than 2 Gb/s, which is likely. Therefore, in most cases we expect this transfer cost to be negligible compared to the cost of computation time and storage space.

At these rates, our two clusters will take three years to break even with an Amazon EC2 and S3 implementation. The discrepancy between these figures and those given in section 2.1 is a reflection of the lower number of CPU cores per node in our older hardware, as well as our low hosting costs. Even with this older hardware, a utilization level of 25% is enough to bring the break-even point down to two years.

5 Future Directions

For our projects – and we believe this is true for others too – it is difficult to budget for computation and storage needs. How much we want depends on how much it costs. A platform for universal personalized medicine should permit individuals to form small communities that suit their own needs, while retaining much of the economy of scale available to much larger communities. We believe that this can be achieved by building a highly decentralized global network of Free Factories that allocate underutilized resources through market mechanisms.

Others have explored the possibility of capturing market signals from users and we believe this is an attractive way to allocate resources for our applications in the long term [4, 10, 26]. Since we have control of our architecture from the hardware up, we hope that implementation and experimentation with such mechanisms will be provide an opportunity for fruitful future research.

Finally, in the spirit of free and open source software, we hope others will deploy Free Factories of their own for applications we have never imagined.

6 Acknowledgements

This work was supported by a Center for Excellence in Genome Sciences grant from the National Human Genome Research Institute. We would like to thank our anonymous reviewers for providing numerous insightful comments. The process of making revisions was greatly facilitated by our shepherd, John Wilkes. Art Mann at Silicon Mechanics helped us with our hardware related requests both large and small. Finally, we are in debt to many people who provided data and suggestions, and developed applications for Free Factories.

References

- [1] Amazon.com, Inc. <http://aws.amazon.com/>.
- [2] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. In *SIGMETRICS '07*, pages 289–300. ACM Press, 2007.
- [3] G. Bell, J. Gray, and A. Szalay. Petascale computational systems. *IEEE Computer*, 39(1):110–112, 2006.
- [4] John Brunelle, Peter Hurst, John Huth, Laura Kang, Chaki Ng, David C. Parkes, Margo Seltzer, Jim Shank, and Saul Youssef. Egg: An extensible and economics-inspired open grid computing platform. In *GECON'06*, Singapore, 2006. World Scientific Publishing.

- [5] Canonical Ltd. <http://ubuntu.com>.
- [6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI'06*, pages 205–218, Seattle, WA, 2006. USENIX Association.
- [7] G. M. Church. The personal genome project. *Molecular Systems Biology*, 1:2005.0030, 2005. <http://personalgenomes.org>.
- [8] Coreboot. <http://coreboot.org>.
- [9] Danga Interactive. <http://danga.com>.
- [10] R.K. Dash, N.R. Jennings, and D.C. Parkes. Computational-mechanism design: a call to arms. *Intelligent Systems*, 18(6):40–47, 2003.
- [11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04*, pages 137–150, San Francisco, CA, 2004. USENIX Association.
- [12] Jon G. Elerath and Michael Pecht. Enhanced reliability modeling of RAID storage systems. In *DSN'07*, pages 175–184, 25-28 June 2007.
- [13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP'03*, pages 29–43, New York, NY, 2003. ACM Press.
- [14] GNU Project. <http://gnu.org>.
- [15] Jim Gray, David T. Liu, Maria Nieto-Santisteban, Alex Szalay, David J. DeWitt, and Gerd Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34:34–41, 2005.
- [16] Harvard University. Free Factories source code, 2008. <http://factories.freelogy.org>.
- [17] Jae Bum Kim, Gregory J Porreca, Lei Song, Steven C Greenway, Joshua M Gorham, George M Church, Christine E Seidman, and J. G. Seidman. Polony multiplex analysis of gene expression (PMAGE) in mouse hypertrophic cardiomyopathy. *Science*, 316(5830):1481–1484, Jun 2007.
- [18] Lawrence Livermore National Laboratory. <https://computing.llnl.gov/linux/slurm/>.
- [19] Jeantine E Lunshof, Ruth Chadwick, Daniel B Vorhaus, and George M Church. From genetic privacy to open consent. *Nature Reviews Genetics*, 9:406–411, May 2008.
- [20] Nature Publishing Group. Positively disruptive. *Nature Genetics*, 40(2):119, Feb 2008.
- [21] Opendgear, Inc. <http://opengear.com>.
- [22] OpenVPN, Inc. <http://openvpn.net>.
- [23] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andre Barroso. Failure trends in a large disk drive population. In *FAST'07*, San Jose, CA, 2007. USENIX Association.
- [24] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *FAST'07*. USENIX Association, 2007.
- [25] Jay Shendure, Gregory J Porreca, Nikos B Reppas, Xiaoxia Lin, John P McCutcheon, Abraham M Rosenbaum, Michael D Wang, Kun Zhang, Robi D Mitra, and George M Church. Accurate multiplex polony sequencing of an evolved bacterial genome. *Science*, 309(5741):1728–1732, Sep 2005.
- [26] Jeffrey Shneidman, Chaki Ng, David C. Parkes, Alvin AuYoung, Alex C. Snoeren, Amin Vahdat, and Brent Chun. Why markets could (but don't currently) solve resource allocation problems in systems. In *HotOS-X*, Santa Fe, NM, 2005.
- [27] Software in the Public Interest. <http://debian.org>.
- [28] Alexander Szalay and Jim Gray. 2020 computing: science in an exponential world. *Nature*, 440(7083):413–414, Mar 2006.
- [29] X-Prize Foundation. Archon x-prize for genomics. <http://genomics.xprize.org/>.
- [30] Kun Zhang, Adam C Martiny, Nikos B Reppas, Kerrie W Barry, Joel Malek, Sallie W Chisholm, and George M Church. Sequencing genomes from single cells by polymerase cloning. *Nature Biotechnology*, 24(6):680–686, Jun 2006.
- [31] Kun Zhang, Jun Zhu, Jay Shendure, Gregory J Porreca, John D Aach, Robi D Mitra, and George M Church. Long-range polony haplotyping of individual human chromosome molecules. *Nature Genetics*, 38(3):382–387, Mar 2006.