Challenges in Executing Data Intensive Biometric Workloads on a Desktop Grid

Christopher Moretti[1], Timothy C. Faltemier[1], Douglas Thain[1], and Patrick J. Flynn[1]

[1]University of Notre Dame
Computer Science and Engineering Dept.
Notre Dame, IN 46556
{cmoretti, tfaltemi, dthain, flynn}@cse.nd.edu

Abstract

Desktop grids have traditionally focused on executing computation intensive workloads. Can they also be used to execute data-intensive workloads? To answer this question, we present a case study of a data intensive biometric application which is infeasible to process on a single machine. We evaluate the capacity of a desktop grid to store and deliver the data need to execute the workload, and compare several general techniques for data deployment. Selecting the most scalable technique, we execute and evaluate five large production workloads on a 350-CPU desktop grid. We observe that this technique is sensitive to many parameters, and propose that an ideal system should be responsible for choosing the proper decomposition of a workload.

1 Introduction

Can a desktop grid support data intensive workloads? Traditionally, desktop grids have focused on computation intensive workloads because of the requirement to evacuate a machine when its owner wishes to use it. However, modern machines have large amount of unused storage that can be used to serve data when the owner is away. It may not be necessary to delete the stored data when the owner returns.

In this paper, we present a significant biometric application that may be able to take advantage of a desktop grid. Based on measurements of a particular desktop grid, we observe that the raw capacity is sufficient to serve such an application. We explore several data distribution methods, and measure their relative scalability. Choosing one data distribution method, we execute five production biometric workloads. We observe that the nature of the desktop grid results in extreme I/O bursts varying by several orders of magnitude and a wide range of failure modes. Through this exercise, we demonstrate that a desktop grid can in fact support data-intensive workloads. However, we do not address all open issues in this problem. Significant open problems remaining include local storage evacuation, distributed data management, and automatic system tuning. We propose these problems for future research.

The Biometric Problem. Biometrics is the science of identifying people by their physical traits such as fingerprints, iris scans, and facial images. In this paper, we consider the problem of identifying a person from a 3D image generated by a stereoscopic camera. In the field, such images would be collected from people passing through a checkpoint. Each image would then be compared to a potentially very large watch list. (Currently the US has a terrorism watch list of nearly 325,000 individuals [26].) To do this, each new image must be compared to each image in the watch list by executing a *matching function* that compares each to the other and returns a value between zero and one, indicating the quality of each pairwise match.

The scientific problem is: *How do we design and evaluate an accurate matching function?* A good matching function must identify like images, reject unlike images, avoid false positives and false negatives, and handle all of the range of human moods and appearances. The design of a matching function is currently an open problem [28, 9]. Whatever the design, it must be evaluated on a large number of images before it can be trusted for real use.

To evaluate the quality of a matching function, we must obtain a large array of images and compare all of them to each other. Figure 1 shows this procedure: the matching function is executed for all pairs of images, and the result is a *similarity matrix* where each cell represents the quality of match between each image. For each proposed matching function and image data set, a similarity matrix must be generated. Now, the quality of several functions may be compared and evaluated by examining their matrices.

A typical workload of this kind consists of 4000 im-

Biometrics research at the University of Notre Dame is supported by the National Science Foundation under grant CNS01-30839, by the Central Intelligence Agency, by the US Department of Justice/ National Institute for Justice under grants 2005-DD-CX-K078 and 2006-IJ-CX-K041, by the National Geo-spatial Intelligence Agency, and by UNISYS Corp.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

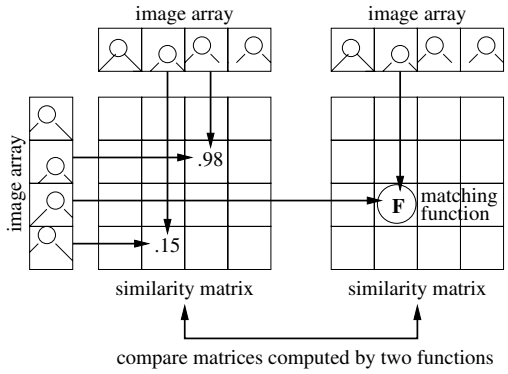

Figure 1. Biometric Similarity Matrices

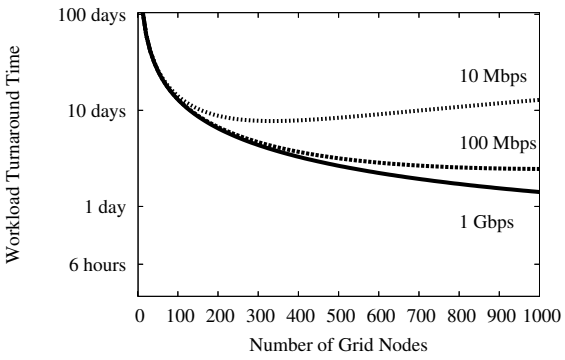

Figure 2. Ideal Model for Completion Time

ages of about 256 KB each taken from the Face Recognition Grand Challenge (FRGC) [21] data set, all compared to each other with the ICP [8] algorithm. Each match compares 14 regions of the face, each region requiring about 1 second of computation. Although 1 GB of total data may seem small, the polynomial complexity of the problem makes this a challenging workload. If an efficient method of executing this workload can be found, biometric researchers would like to scale up to workloads of arbitrary size.

The Computing Problem. Decomposing the computation and the data is critical to attacking this problem. A single machine would complete the 4000 image workload in about 90 days. Adding a second machine would cut the computation time in half, while adding some time to transfer the images; let's assume 10 seconds to transfer 1GB on a 1Gbps network. Adding more grid nodes to the job continues to improve performance until the point where the additional deployment takes longer than the time saved by adding another grid node. Figure 2 shows the ideal performance curves for this application on several networks.

However, there are many obstacles to achieving the ideal performance. Assuming data is deployed from a central server on a 1 Gb/s network, 100 nodes would require 15 minutes to deploy to the last node, if done sequentially. If done in parallel, each of the nodes must wait the full 15 minutes. These are wasted minutes in which no CPUs can be harnessed, as well as wasted effort if a job or transfer should fail. Things get worse on a slower network.

In order to distribute the data, we must find storage from which the data can be served. Each node in the desktop grid may have limited storage space or policies that make it infeasible to permanently keep the entire data set on each node (or even smaller subsets of the data to be worked on by that node). This increases the importance of the data servers, and forces us to break up data sets into smaller chunks to utilize the largest possible portion of our grid.

A benefit of the problem's structure is that it allows both partitioning and reuse of data. Not all of the grid nodes need all of the data in order to complete their portion of the computation. In fact, because this workload can be partitioned into very small portions, not even all of the servers need all of the data to serve, if the serving algorithm is clever. This can allow us to make use of cached data at the servers, by keeping portions of a data set hot on various servers.

Such an I/O intensive workload would normally require a supercomputing center. In this paper, we ask: *Can a desktop grid execute this demanding biometric workload?*

2 Observations of a Desktop Grid

To answer this question, we begin by measuring the unused storage and I/O capacity of a desktop grid based at the University of Notre Dame. This particular grid consists of approximately 250 machines containing 350 CPUs, all running Condor [25] to manage the CPUs and the Tactical Storage System [24] to manage the storage. The machines involved serve a mix of purposes: there are office workstations, classroom workstations, and research clusters. Machines vary in age from brand-new to five years old. This variety means that the grid nodes span several different positions in the interpretation of what exactly is a desktop grid. This grid fits within a continuum between an unreliable, diverse, worldwide (or at least campus-wide) scavenging pool and a well-monitored, homogenous, geographically local, tightly coupled cluster. We cannot prove that all of the properties here are universal, but we believe the preceding description applies to many computing environments.

Figure 3 shows both static and dynamic properties of this system. The graphs on the left show the distribution of system resources on October 16, 2006. The upper-left graph shows the CPU performance and memory capacity of each CPU. The lower-left graph shows the total disk size and the free space available on each host in the system. Note that the number of CPUs does not correspond to the number of hosts, because each host may have multiple CPUs.

The two graphs on the right show the system availability

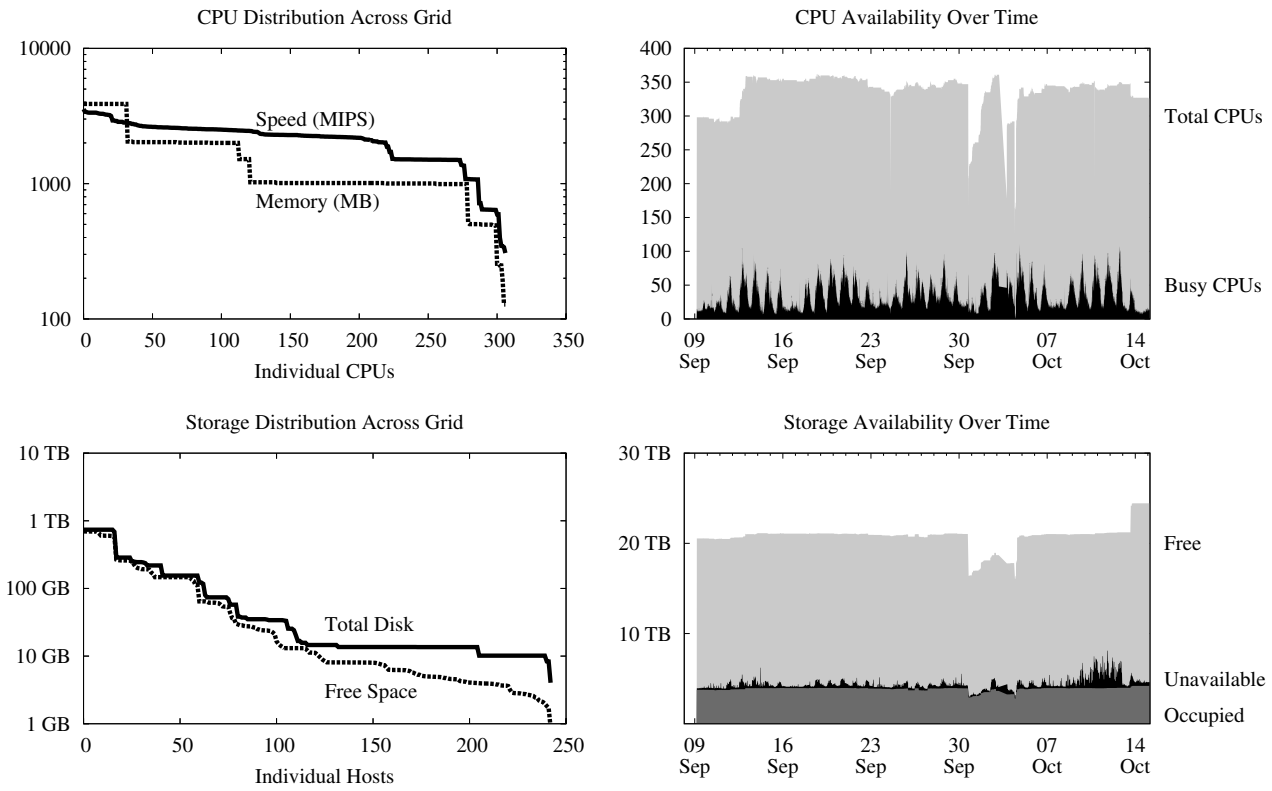

Figure 3. Performance Heterogeneity and Time Variance in a Desktop Grid

over five weeks in the fall of 2006. The upper-right graph shows CPU availability. A CPU is "busy" if the CPU load average is above 0.5 or the keyboard has been touched in the last fifteen minutes. As might be expected, CPU usage is cyclic with respect to days and weeks. Overall CPU utilization varies between five and thirty percent. The lower-right graph shows storage availability. We assume that disks should have the same constraints as CPUs: if the machine is in use by the owner, the disk should not be harnessed. Accordingly, we define three storage states: "Occupied" space actually contains data; "Unavailable" space is empty, but the corresponding CPU is busy; and "Free" space contains no data, and the corresponding CPU is idle.

Several important claims may be made from this data:

Large amounts of unused storage space. Across the entire system, there is consistently over 15 TB of total space available for use by a desktop grid. This is because most users (quite rationally) place their most critical data in external distributed file systems or network attached storage, where they can exercise greater control over performance, access control, and archive status. However, this desktop space exceeds the size of many users' archives, so it appears quite feasible to use the distributed storage as a scratch space for staging very large data sets in and out of the grid.

Large amounts of unused I/O bandwidth. Bandwidth is equally as important as capacity. A 15TB archive is not very useful if it cannot deliver data at the speed required by an application. The large number of individual disks in the system presents the possibility of very high I/O rates. If a data set can be delivered and partitioned appropriately across many (possibly slow) disks, an enormous amount of throughput can be delivered to an application. If we assume a modest disk bandwidth of 10 MB/s across 250 devices, this system has the *potential* to deliver 2.5 GB/s of data.

High degree of performance heterogeneity. Nodes vary considerably in CPU performance and memory capacity, and even more in storage capacity. Several orders of magnitude separate the best and worst machines in each metric. A computing system must take this into account. A highly synchronous parallel algorithm would always be waiting for the slowest machine, so asynchrony must be pursued wherever possible. When partitioning data across nodes, the size of the data chunk constrains the set of nodes that can store the data. To maximize I/O bandwidth and storage utilization, data partitioning must be non-uniform.

Major periodic disruptions. Even in a one-month timeline, we may see several major changes to the system. Between September 9th and 16th, a new cluster was turned on

and added 50 CPUs to the system. Between September 30th and October 7th, a major power outage and then a hardware failure at the data center brought about half of the system offline. Across the entire month, we may see a slow "decay" in the number of CPUs as hardware fails and software crashes. (Typically, such failed systems are fixed in batches at the end of a semester.) To harness a desktop system, we cannot wait for a perfect moment when the system is completely operational. Instead, we must organize workloads and system structures to accommodate these problems.

3 Data Distribution

We have shown that a desktop grid has the raw capacity to execute a data intensive workload, but that doesn't mean the data management required to do so is trivial. How can we lay out the data within the grid so that it can be accessed efficiently? We consider four methods to attack this problem, which vary in terms of data partitioning and access.

Single Package Single Server (SPSS) In a desktop grid, not all workstations have the storage resources needed for all workloads. Even smaller data sets often consume too many resources to allocate on every grid host long-term. The simplest solution is to ship data directly to each host as it is chosen in the queue for computation. This naïve approach sends each available machine the packaged experiment (a single file archive of all the gallery images, a probe image, the executable, and the required library files) with instructions as to which image to process. On completion, the machine sends the results back to the source.

This approach has limitations. A single server may not be able to support the continuous bandwidth needed by a workload. Even if it can support the aggregate, a batch system may start many jobs at once, resulting in simultaneous requests for terabytes of data over thousands of connections. This burstiness slows the response to all of the requests, due to finite memory and bandwidth on the data server.

Single Package Multiple Server (SPMS) To solve the problem of efficiently responding to a large batch of simultaneous matches, we can rely on prestaging data at various points across the network. Having sufficiently large numbers of prestaged data servers ensures that data transfer can occur upon fetching/matching a job from the queue to the grid node without overburdening the submitting machine with the large overhead requirements. Once the data is transferred to our prestaged data servers, the same after-match process is completed: transferring the packaged experiment to its local machine, unpacking it, and then completing the matching for each of the images in the gallery. On completion, the result is sent back to the original submitting machine. If a single machine is both client and server, the server should always choose its local copy, rather than going to a random server on the network.

Though this method solves the problem of overloading the central server, it does not address the fundamental issues of the storage requirement. We must account for the case that many grid nodes will have less disk space than is consumed by the data set. Requiring the entire data set to fit on a grid node results in poor utilization of the grid pool. We suggest storage sufficiency is another of the factors for consideration in the binary classification of host availability summarized by Kondo et al. [16].

Single File Multiple Server (SFMS) Instead of sending the entire data set in advance, which requires it to fit on the grid node, we can send each image individually as needed.

This method of approach is similar to a conventional distributed file-system in that it activates accesses at a file level (as opposed to a data set level), but differs in that it creates a copy of the remote data on the local grid node (like our previous methods), rather than conducting exclusively remote operations. This is a palatable solution for read-only data.

The prestaging still solves the initial throughput issue presented against the naïve approach. The image-by-image on-demand access allows us to restrict our grid node space usage to that necessary for the current computation.

This model over-corrects from the last observation, however. Whereas all-at-once access is too restrictive from the standpoint of grid node space requirements (resulting in low resource utilization), the problem with this approach is one of performance. The entirety of the gallery will eventually (by the end of the job) be compared to the image, but this model does not make use of that. It requires a new connection to download each gallery image to the grid node individually. The number of network connections scales linearly with the number of comparison images in the gallery, increasing overhead from previous models.

A variant of this method that makes use of prefetching is feasible. Future images are buffered while current computation is occurring, storing only a fraction of the total gallery space at any one time (but more than the 2 image requirement of the non-prefetching case described above). Adding prefetching, however, still does not alleviate the basic problem of connections scaling with comparisons (for either the computing node, or the data server). It also doesn't account for the detriment to the computation from sharing CPU resources with the prefetching process.

Multiple Package Multiple Server (MPMS) Instead of creating a single file containing the the entire data set, it may be broken down into pieces that can fit onto the large majority of the grid nodes. If we use the same technique as in SPMS, but on chunks of the data set instead of the entire package, we can balance the low overhead from SPMS with the ability to harness many more nodes.

An additional advantage of MPMS is a minimum of repeated work. With SPMS, a single failure requires resubmitting the entire set of comparisons; here, only the specific

comparison that failed must be resubmitted. On the other hand MPMS is more complicated and requires networking connections between many hosts in the system.

Network Feasibility and Scale. We hypothesize that the various methods will result in different levels of I/O throughput, as measured by completed tasks over a given time period, and we want to find whether any single method will deliver the highest throughput under increasing loads. The metric of how many tasks can be completed within a time limit is used for two reasons. The first is that grid operations have timeouts to protect against hosts that go offline, or otherwise become dysfunctional. The second is to give a sense of how a method performs in bursty traffic. If a server cannot deliver files in parallel in a timely manner, it leaves CPUs unutilized, and is at risk of falling progressively farther behind within a burst.

To determine this, we measured the sustainable throughput of each method by completing that method's I/O pattern over a variable number of clients. Each of the multiple-server methods served data from five servers. For SPSS and SPMS, each host read a series of 500MB files; for MPMS each host read a series of 50MB files; and for SFMS each host read a series of 250KB files. The number of completed tasks can be multiplied by the file size and divided over the time period to achieve throughput. Figure 4 shows the I/O capacity provided by our four methods running on the production grid. The fifth plot is the MPMS method with the replica selection algorithm adjusted from random server (the default choice for MPMS) to one that recognizes cluster locality and will pick the appropriate intra-cluster replica.

The inherent limitation of a single-server setup (SPSS) is evident. Its peak throughput is lower than the multiple server version (SPMS), at just under 100MB/s, and it can maintain this only up to approximately a load factor of 25, after which it drops off to less than 5MB/s. SPMS reaches a peak throughput of over 120MB/s around a load factor of 35, and can sustain throughput of 80MB/s beyond a load factor of 80. MPMS reaches a peak throughput above 190MB/s, which doesn't occur until a load factor between 75 and 80. Utilizing cluster locality, we can push the peak throughput of MPMS beyond 230MB/s. We reach the point at which both of these MPMS curves have levelled off, but we have not reached the point at which they begin to drop off significantly. The SPSS method peaks at only 13MB/s, around a load factor of 55. It maintains up to half of that peak through a load factor of 75.

The data demonstrate that MPMS scales to both the highest peak throughput, and the highest load factor before levelling off. If greater throughput is needed, more servers can be added. In terms of peak throughput, SPMS fails to get even double the single server's value, and reaches the peak throughput within a factor of two of the corresponding load factor. The early precipitous decline in SPSS and the early levelling of SPMS are likely due to the larger file size. The large file cannot be transferred completely to each node within the time limit when under heavy load.

The SFMS method cannot serve sufficiently many files to supply an entire data set to a large number of servers on demand. This is due to an implementation detail: the job execution script creates a new TCP connection to fetch each file. If a single TCP connection could be re-used, a single server can support a large number of operations: The right graph in Figure 4 shows that a single serve can serve 25,000 RPC/s under heavy load. However, re-using the same connection multiple times within a script is not trivial with conventional software technologies.

Three conclusions about I/O access in desktop grids:

Single servers do not scale. A single server cannot serve sufficient throughput over its network link to sustain a large desktop grid. Even though a single high performance server can accommodate a large load with its storage and memory, it is limited by the bandwidth of the network connection. Further, multiple small servers are more cost effective.

Single file per connection causes significant overhead. While accessing files one by one allows for minimal space requirements on grid nodes, it comes at the cost of large overhead to create separate connections for each file. To manage data efficiently, even for relatively large files, the system must maintain connection state wherever possible.

Cluster locality improves scalability. A node within a fast cluster suffers moderately when it must fetch data from a slow cluster. Choosing a replica that is many network hops away, even over a fast connection, introduces unnecessary latency. Thus, we should utilize intra-cluster replicas when possible to maximize performance of the fast networks.

4 Experience with Real Workloads

Using this understanding of data distribution, we constructed a a prototype system for executing data intensive biometric workloads on our desktop grid described above. Here, we describe experience with five large workloads used to produce results that are described in reference [11].

The table in Figure 5 summarizes the five runs. The first two runs correspond to the scenario described at the beginning of the paper: 4007 images must be compared to each other. Each comparison involves 14 sub-comparisons of regions of the face, each taking about one second on a 1-GHz CPU. A total of 1300 CPU-days (on a 1-GHz CPU) is needed to execute the computation. Each image is about 256KB, and the entire image data set is 1.2GB. The third run uses a different collection of 3851 images, but only compares one region per image, and so runs much faster with a higher I/O-CPU ratio. Run 4 is again like Run 1 with a larger collection of smaller images. Run 5 uses the same data set as run 4, but processes 5 columns in each batch job.

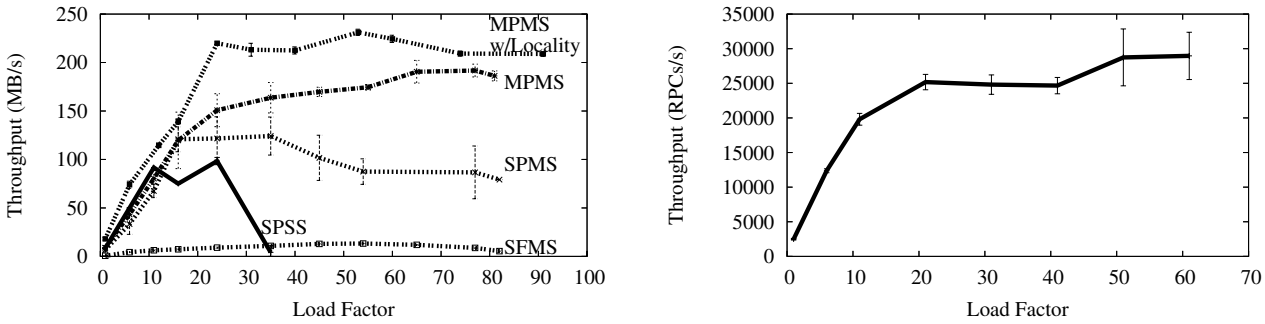

Figure 4. I/O Scalability of Data Partitioning and Access

To support these workloads, we chose the following fixed (conservative) MPMS data configuration: For each run, the input data set was broken in 50MB chunks and replicated to 32 disks across the grid, each large enough to hold all chunks at once. These 32 nodes served as runtime I/O servers for the remaining grid nodes. A global directory of I/O servers was established. Each workload was partitioned into jobs, each processing one column of the result matrix. Each job obtained the necessary chunks one at a time to process its portion of the workload. To choose an I/O server, each job consulted the global directory and picked the host with the most similar name, assuming that it would be the "closest" server.

Figure 5 also shows timelines of resources consumed by each of the five workloads over the course of a month. The upper graph shows the number of jobs running at any given time. Three shades of gray indicate jobs that have been matched to a CPU, but are not yet running, jobs that are running on a CPU, and jobs that are suspended because the owner of the machine is using the keyboard or CPU.

The lower graph shows the total cluster I/O rate on the same time scale as the upper graph. This is obtained by summing the total amount of data read from each of the 32 I/O servers over one minute intervals. (Note that we do not include the I/O implicitly performed by the batch system to stage executables, write outputs, etc, which is far less.)

A few anomalies about the graph should be noted. Run 1 was limited to 60 reliable cluster nodes, in order to test the system before expanding to the full desktop grid. These machines all had fast 3GHz processors, and thus completed one workload in 9 days. Run 2 was run in competition with several other workloads on the desktop grid. As other workloads finished, groups of CPUs became available at once. In addition, a off-by-one bug in the I/O server selection code would cause a fraction of the jobs started in each burst to fail immediately, resulting in the fast fall-off of each job starting burst. (This was fixed in later runs.) Run 3 completed quickly with a very high I/O rate due to the single comparison per image employed. In addition, it was interrupted by the system failures observed in Figure 3 above.

Several more general observations may be made:

Very Bursty I/O Rates. Because jobs are submitted (and other workloads complete) in large batches, cluster-wide I/O vary by several orders of magnitude. Several regular bursts exceed the steady state by a factor of ten or even one hundred times. The two highest I/O bursts seen are 1.42 GB/s and 1.02 GB/s, both during start-up bursts in the more I/O intensive workloads. To support such bursts, it is necessary to significantly over-provision the I/O system; given the large amount of available disk space, this is cheap to do.

High Failure Rate. Although we have observed that desktop grids offer a large number of unused cycles, there are still many events that can interfere with the execution of a job. For each workload, a significant number of jobs are evicted from the executing machine. In the absence of checkpointing, this results in the (potentially large) data transfer at the beginning of the job going to waste. Thus, it is important to minimize start-up I/O overhead (as MPMS does), because it must be repeated for each restart.

Silent Failures. Each workload suffered a number of failures that were not detected (and thus not retried) by the batch system. Such jobs appeared to exit with a normal exit status, while others did not print out correct results. Not all of these failures could be diagnosed, but some were due to system-level issues such as missing or incompatible shared libraries. To address these problems, each workload ended with a "catch up" run of jobs that were manually determined to have failed. Evidence of this can been seen in I/O bursts at the end of several runs.

Despite these challenges, we have demonstrated that real, production biometric workloads with very high data rates can be successfully executed on a desktop grid. However, several open problems remain.

5 Related Work

Many large scale cycle scavenging systems are designed on the assumption that codes are very computation inten-

	Workload Properties					Runtime Totals			
Run	Images	Jobs	Rgns	CPU Hours	Data Size	Max CPUs	Susp.	Evicts	Total I/O
1	4007	4007	14	1300	1.2 GB	60	555	113	0.68 TB
2	4007	4007	14	1300	1.2 GB	226	3653	494	2.29 TB
3	3851	3851	1	85	850 MB	293	483	71	2.00 TB
4	7263	7263	1	305	1.8 GB	210	8607	658	5.31 TB
5	7263	1453	1	305	1.8 GB	255	4383	331	1.11 TB

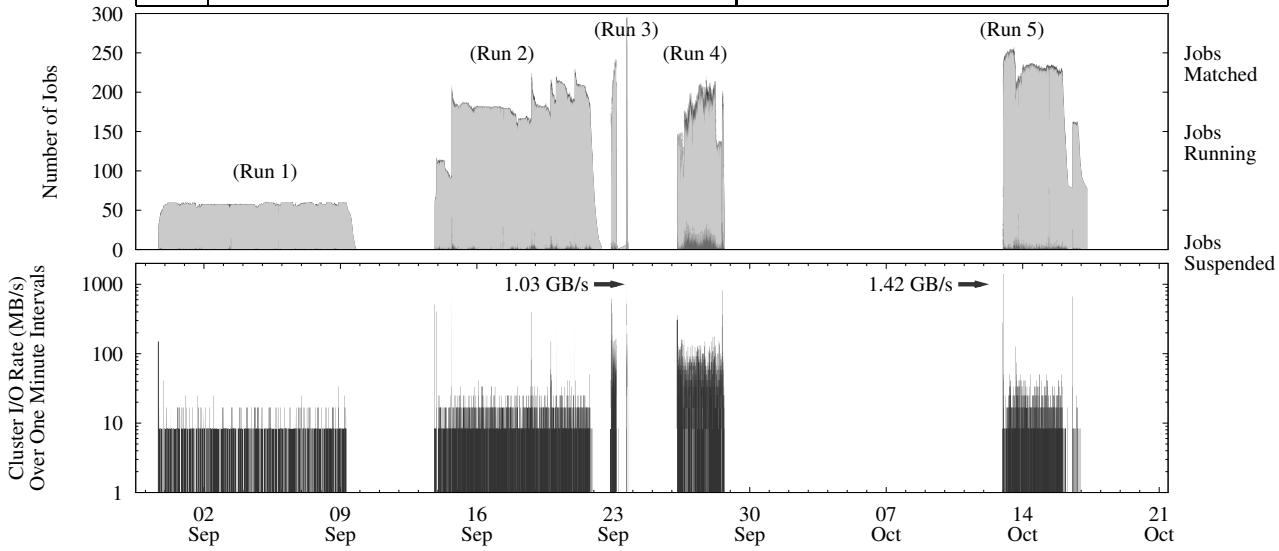

Figure 5. Summary of Five Production Biometric Workloads

sive and have low data rates. For example, the Folding@Home [18] protein modeling system requires about 100KB of data to seed a simulation that will run for several days. SETI@Home requires only 35KB of data per CPU hour [3], The Great Internet Mersenne Prime Search transfers a few hundred bytes per week [27]. The Master-Worker [19] and XtremWeb [12] systems have attacked several important optimization problems, which typically require a few bytes of input and then many CPU hours to explore many possible solutions.

Anderson and Fedak [4] have shown that large scale desktop scavengers have the capacity to run codes with moderate data rates. They define a ratio R as MB of I/O per hour of computation time on a 1 GFLOPS machine, and observe that current systems have the potential to handle applications with R as high as 1 or even 10 while using a central server. Our biometric application has a much higher data rate, consuming MB of data in seconds, not hours.

The need for *data* intensive desktop computing comes from scientific applications. Typically, these data intensive applications are developed and tested in environments with a high I/O capacity. Once tested, users wish to scale these applications up by degrees: first on tens of CPUs in a tightly coupled cluster, then hundreds of CPUs in a campus grid, and then thousands of CPUs in a national-scale grid or desktop computing environment. Foster and Iamnitchi describe several such applications, including an analysis of astrophysics data that averages 660MB per CPU hour [13]. Thain et al. [23] describe five data intensive grid applications with complex data sharing behavior.

Many systems aim to support a wide area "data grid" [1] that facilitates data sharing between institutions. For example, GridFTP [2], SRB [6], Nest [7], Stork [17], and IBP [22] all provide facilities for storing, transferring, managing over the wide area, but do not particularly address the issue of getting data close to CPUs.

Many filesystems harness the storage capacity and throughput of an entire cluster. xFS [5] and Zebra [15] stripe small blocks across cluster nodes, Google-FS [14] and Freeloader replicate large chunks, while object-based filesystems [20, 10] distribute individual files. All these systems allow the aggregate performance of a cluster to be harnessed, but do not address the problem of tailoring data layout to the intended computational workload.

6 Conclusion and Future Work

In this paper, we have discussed options for selecting the optimal distribution and processing method for biometrics on a desktop grid. Neither single-file-on-demand, nor a full-data-on-placement is ideal; the former suffers in per-

formance, and the latter suffers in scalability. The MPMS method is a scalable compromise between these extremes.

Several open problems remain:

Local Data Management. We have assumed that local storage is so plentiful that the user will never notice if space is consumed indefinitely. This is certainly not the case over the long term. A desktop grid harnessing storage must have some ability to evict local data when necessary. This is non-trivial, because file deletion is a relatively slow activity that must be done well before the space is needed.

Distributed Data Management. We have proposed that a desktop grid can handle I/O intensive workloads by replicating data out to the various nodes of the grid. For a single user, there is plenty of space. But, if many users recognize these untapped resources, there is a natural incentive to replicate data widely and quickly consume the space. To address this problem, a desktop grid must have a process that is responsible for managing space across the entire system. For example, a user might request space for 10 copies of a 10 GB file; the data manager would choose ten disks, allocate the space, and inform the user. A quota of time and space would be needed to limit consumption.

Automatic Configuration. This system has many different parameters to tune: the number of CPUs to harness, the number of data copies, the ratio of comparisons to jobs, and so forth. As Figure 2 shows, choosing the wrong parameter can lead to poor performance. A more user-friendly system would accept a work specification, and then choose the appropriate system parameters for execution.

References

[1] W. Allcock, A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *J Network and Computer Applications*, 2001.

[2] W. Allcock, A. Chervenak, I. Foster, C. Kesselman, and S. Tuecke. Protocols and services for distributed data-intensive science. In *Proc Advanced Computing and Analysis Techniques in Physics Research*, pages 161–163, 2000.

[3] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home an experiment in public-resource computing. *CACM*, 45(11):56–61, 2002.

[4] D. P. Anderson and G. Fedak. The computational and storage potential of volunteer computing. In *IEEE/ACM Symposium on Cluster Computing and the Grid*, May 2006.

[5] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *ACM Symposium on Operating System Principles*, Dec 1995.

[6] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *Proceedings of CASCON*, Toronto, Canada, 1998.

[7] J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Flexibility, manageability, and performance in a grid storage appliance. In *IEEE High Perf Dist Comp*, July 2002.

[8] P. Besl and N. McKay. A method for registration of 3-D shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14, 1992.

[9] K. Bowyer, K. Chang, and P. Flynn. A survey of approaches and challenges in 3d and multi-modal 3d + 2d face recognition. *Computer Vision and Image Understanding*, 2006.

[10] Cluster File Systems. Lustre: A scalable, high performance file system. white paper, November 2002.

[11] T. Faltemier, K. Bowyer, and P. Flynn. 3D face recognition with region committee voting. *3D Data Processing, Visualization, and Transmission*, 2006.

[12] G. Fedak, C. Germain, V. Neri, and F. Cappello. XtremWeb: A generic global computing system. In *IEEE Workshop on Cluster Computing and the Grid*, May 2001.

[13] I. Foster and A. Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In *2nd Intl Workshop on Peer-to-Peer Systems*, February 2003.

[14] S. Ghemawat, H. Gobioff, and S. Leung. The Google filesystem. In *ACM Symp on Operating Systems Principles*, 2003.

[15] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, 2001.

[16] D. Kondo, M. Taufer, C. Brooks, H. Casanova, and A. Chien. Characterizing and evaluating desktop grids: An empricial study. In *IEEE/ACM International Parallel and Distributed Processing Symposium*, Santa Fe, NM, 2004.

[17] T. Kosar and M. Livny. Stork: Making data placement a first class citizen in the grid. In *IEEE International Conference on Distributed Computing*, 2004.

[18] S. M. Larson, C. Snow, M. Shirts, and V. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. In R. Grant, editor, *Computational Genomics*. Horizon Press, 2002.

[19] J. Linderoth, S. Kulkarni, J.-P. Goux, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *IEEE High Perf Dist Comp*, August 2000.

[20] M. Mesnier, G. Ganger, and E. Riedel. Object based storage. *IEEE Communications*, 41(8), August 2003.

[21] P. J. Phillips, P. J. Flynn, T. Scruggs, K. W. Bowyer, J. Chang, K. Hoffman, J. Marques, J. Min, and W. Worek. Overview of the face recognition grand challenge. *Proceedings of IEEE Computer Vision and Pattern Recognition*, 2005.

[22] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the network. In *Network Storage Symposium*, 1999.

[23] D. Thain, J. Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Pipeline and batch sharing in grid workloads. In *IEEE High Perf Dist Comp*, June 2003.

[24] D. Thain, S. Klous, J. Wozniak, P. Brenner, A. Striegel, and J. Izaguirre. Separating abstractions from resources in a tactical storage system. In *Supercomputing*, Nov 2005.

[25] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infra. a Reality*. Wiley, 2003.

[26] The Washington Post. 325,000 Names on Terrorism List. http://www.washingtonpost.com/wp-dyn/content/article/2006/02/14/AR2006021402125.html, 2006.

[27] G. Woltman. The great internet mersenne prime search. Available from `http://www.mersenne.org`.

[28] W. Zhao, R. Chellappa, P. J. Phillips, and A. Rosenfeld. Face recognition: A literature survey. *ACM Comp. Surv.*, 2003.