All-Pairs: An Abstraction for Data-Intensive Computing in Shared Distributed Systems

Christopher Moretti, Jared Bulosan, Douglas Thain, and Patrick J. Flynn
Department of Computer Science and Engineering, University of Notre Dame [*]

Abstract

Although modern parallel and distributed computing systems provide easy access to large amounts of computing power, it is not always easy for non-expert users to harness these large systems effectively. A large workload composed in what seems to be the obvious way by a naive user may accidentally abuse shared resources and achieve very poor performance. To address this problem, we propose that production systems should provide end users with high-level abstractions that allow for the easy expression and efficient execution of data intensive workloads. We present one example of an abstraction – All-Pairs – that fits the needs of several data-intensive scientific applications. We demonstrate that an optimized All-Pairs abstraction is both easier to use than the underlying system, and achieves performance orders of magnitude better than the obvious but naive approach, and twice as fast as a hand-optimized conventional approach.

1 Introduction

Many scientists have large problems that can make use of distributed computing, however most also are not distributed computing experts, and thus they can't be expected to get everything right. This is especially true considering the large number of factors involved in large distributed systems and the software that harnesses these resources. Inadvertent poor choices can result in outright failures, poor performance, ineffi cient use of resources, or abuse of the infrastructure.

Providing an abstraction for a class of problems is one approach to avoiding the pitfalls of distributed computing. The abstraction for a problem gives the user an interface to defi ne their problem in terms of data and computation requirements, while hiding the details of how the problem will be realized in the system. A well-designed system will consider the range of complications not apparent to a novice to make choices that avoid disastrous decisions. The goal is not to strip power from smart users, but rather to make distributed computing accessible to non-experts.

We have implemented one such abstraction – All-Pairs – for a class of problems found in several scientifi c fi elds. This implementation has several broad steps. First, we model the workflow so that we may predict execution based on grid and workload parameters, such as the number of hosts. We distribute the data, via spanning tree, to the compute nodes, being as flexible as possible

[*]This work was supported by National Science Foundation grants CCF-06-21434 and CNS-06-43229.

as to which nodes are chosen. We dispatch batch jobs that are structured based on the model to provide good results. Once the batch jobs have completed, we collect the output into a canonical form for the results matrix, and delete the data left on the compute nodes.

We examine the two algorithms for serving the workload's data requirement – demand paging and active storage. Active storage delivers higher throughput and efficiency for several large workloads on a shared distributed system. We evaluate these two algorithms on All-Pairs problems in biometrics and data mining on a 500-CPU shared computing system. The turnaround times for these workloads can be up to a factor of two faster than for demand paging, and orders of magnitude faster than the non-experts' choices.

2 The All-Pairs Problem

The All-Pairs problem is easily stated:

All-Pairs(set A, set B, function F) returns matrix M:
Compare all elements of set A to all elements of set B via function F,
yielding matrix M where M[i,j] = F(A[i],B[j]).

Variations of the All-Pairs problem occur in many branches of science and engineering, where the goal is either to understand the behavior of a newly created function F on sets A and B, or to learn the covariance of sets A and B on a standard inner product F. We are working with two user communities that make use of All-Pairs computations: biometrics and data mining.

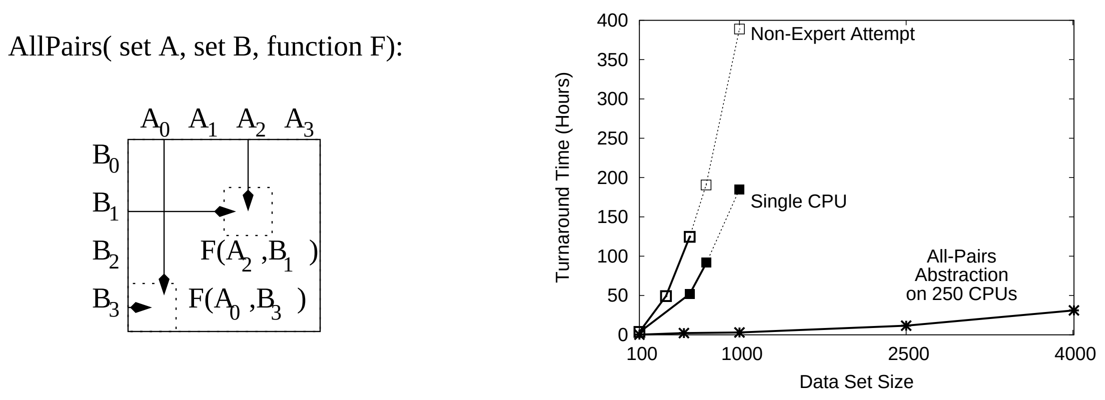
Biometrics is the study of identifying humans from measurements of the body, such as photos of the face, recordings of the voice, or measurements of body structure. A recognition algorithm may be thought of as a function that accepts e.g. two face images as input and outputs a number between zero and one reflecting the similarity of the faces.

Suppose that a researcher invents a new algorithm for face recognition, and writes the code for a comparison function. To evaluate this new algorithm, the accepted procedure in biometrics is to acquire a known set of images and compare all of them to each other using the function, yielding a *similarity matrix*. Because it is known which set images are from the same person, the matrix represents the accuracy of the function and can be compared quantitatively to other algorithms.

A typical All-Pairs problem in this domain consists of comparing 4010 1.25MB images from the Face Recognition Grand Challenge [17] all to each other, using functions that range from 1-20 seconds of compute time, depending on the algorithm in use. Future needs include the All-Pairs comparison of 60,000 iris images of similar size.

Data Mining is the study of extracting meaning from large data sets. One part of this is reacting to bias or other noise within a set. In order to improve at doing this, the researchers must determine which classifiers work on which types of noise. To do this, they use a distribution representative of the data set as one input to the function, and a type of noise (also defined as a distribution) as the other. The function returns a set of results for each classifier, allowing researchers to determine which classifier is best for that type of noise on that distribution of the validation set.

A typical All-Pairs problem in this domain consists of comparing 250 700 KB files, each of which either contains a validation set distribution or a bias distribution; each individual function instance in this application takes .25 seconds to rank the classifiers they are interested in examining.



(A) **(B)**

Figure 1: The All-Pairs Problem

(A) An All-Pairs problems compares all elements of sets A and B together using function F, yielding a matrix of results. (B) The performance of All-Pairs problems on a single machine, when attempted by a non-expert user, and when using an optimized All-Pairs abstraction.

Note that some problems that *appear* to be All-Pairs may be algorithmically reducible to a smaller problem via techniques such as data clustering (Google [5]), or early filtering (Diamond [13]). In these three cases, the user's intent is *not* All-Pairs, but sorting or searching, and thus other kinds of optimizations apply. In the All-Pairs problems that we consider, it is actually necessary to obtain all of the output values. For example, in the biometric application, it is necessary to verify that like images yield a good score and unlike images yield a bad score. The problem is brute force, and the challenge lies in providing interfaces to execute it effectively.

3 Why is All-Pairs Challenging?

Solving an All-Pairs seems simple at first glance. The typical user constructs a standalone program F that accepts two files as input, and performs one comparison. After testing F on his or her workstation, the user connects to the campus batch computing center, and runs a script like this:

```
foreach $i in A
    foreach $j in B
        submit_job F $i $j
```

From the perspective of a non-expert user, this seems like a perfectly rational composition of a large workload. Unfortunately, it will likely result in very poor performance for the user, and worse, may result in the abuse of shared computing resources at the campus center. Figure 1 shows a real example of the performance achieved by a user that attempted exactly this procedure at our institution, in which 250 CPUs yielded *worse* than serial performance.

If these workloads were to be completed on a dedicated cluster owned by the researcher and isolated on a single switched network, conservation and efficient use of resources might only be a performance consideration. However, this is not the case; large scientific workloads often require resources shared among many people, such as a desktop cycle-scavenging system, a campus computing center, or a wide area computing grid. Improperly configured workloads that abuse computation and storage resources, attached networks, or even the resource management software will cause trouble with other users and administrators. These issues require special consideration beyond simply performance.

Let us consider some of the obstacles to efficient execution. These problems should be no surprise to the distributed computing expert, but are far from obvious to end users.

Dispatch Latency. The cost to dispatching a job within a conventional batch system (not to mention a large scale grid) is surprisingly high. Dispatching a job from a queue to a remote CPU requires many network operations to authenticate the user and negotiate access to the resource, synchronous disk operations at both sides to log the transaction, data transfers to move the executable and other details, not to mention the unpredictable delays associated with contention for each of these resources. When a wide area system is under heavy load, dispatch latency can easily be measured in minutes. For batch jobs that intend to run for hours, this is of little concern. But, for many short running jobs, this can be a serious performance problem. Even if we assume that a system has a relatively fast dispatch latency of one second, it would be foolish to run jobs lasting one second: one job would complete before the next can be dispatched, resulting in only one CPU being kept busy. Clearly, there is an incentive to keep jobs large in order to hide the worst case dispatch latencies and keep CPUs busy.

Failure Probability. On the other hand, there is an incentive not to make individual jobs too long. Any kind of computer system has the possibility of hardware failure, but a shared computing environment also has the possibility that a job can be preempted for a higher priority, usually resulting in a rollback to the beginning of the job on another CPU. Short runs provide a sort of checkpointing, as a small result that is completed need not be regenerated. Long runs also magnify heterogeneity in the pool. For instance, a job that should take 10 seconds on a typical machine but takes 30 on the slowest isn't a problem if batched in small sets. The other machines will just cycle through their sets faster. But, if jobs are chosen such that they run for hours even on the fastest machine, then there is a much higher probability of outright failure, or at least a long delay waiting for the final job to complete on the slowest. Thus, there are advantages to short runs, however they increase overhead and thus should be mitigated whenever possible. An abstraction has to determine where the borders are between too short, just right, and too long, noting that these depend on numerous factors of the job, the grid, and the particular execution.

Number of Compute Nodes. It is easy to assume that more compute nodes is automatically better. This is not always true. In any kind of parallel or distributed problem, each additional compute node presents some overhead in exchange for extra parallelism. All-Pairs is particularly bad, because the data is not partitionable: each node needs *all* of the data in order to compute any significant subset of the problem. Data must be transferred to that node by some means, which places extra load on the data access system, whether it is a shared filesystem or a data transfer service. More parallelism means more concurrently running jobs for both the engine and the batch system to manage, and a greater likelihood of a node failing, or worse, concurrent failures of several nodes, which consume the attention (and increase the dispatch latency) of the queuing system.

Data Distribution. After choosing the proper number of servers, we must then ask how to get the data to each computation. A traditional cluster makes use of a central file server, as this makes it easy for programs to access data on demand. However, it is much easier to scale the CPUs of a cluster than it is to scale the capacity of the file server. If the same input data will be re-used many times, then it makes sense to simply store the inputs on each local disk, getting better performance

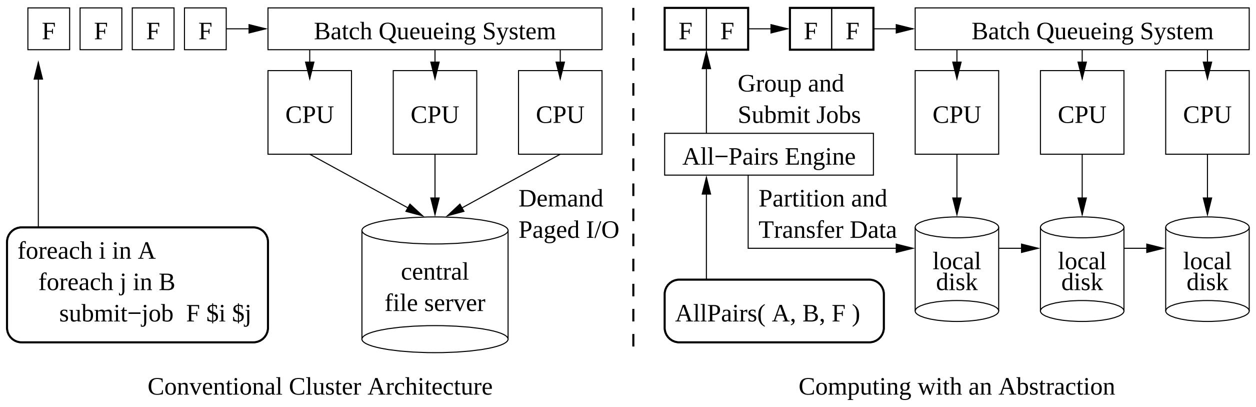


Figure 2: Cluster Architectures Compared
When using a conventional computing cluster, the user partitions the workload into jobs, a batch queue distributes jobs to CPUs, where they access data from a central file server on demand. When using an abstraction like All-Pairs, the user states the high level structure of the workload, the abstraction engine partitions both the computation and the data access, transfers data to disks in the cluster, and then dispatches batch jobs to execute on the data in place.

and scalability. Many dedicated clusters provide fi xed local data for common applications (e.g. genomic databases for BLAST [2]). However, in a shared computing environment, there are many different kinds of applications and competition for local disk space, so the system must be capable of adjusting the system to serve new workloads as they are submitted. Doing this requires signifi cant network traffi c, which if poorly planned can make distributed solving worse than solving locally.

Hidden Resource Limitations. Distributed systems are full of unexpected resource limitations that can trip up the unwary user. The major resources of processing, memory, and storage are all managed by high level systems, reported to system administrators, and well known to end users. However, systems also have more prosaic resources. Examples are the maximum number of open fi les in a kernel, the number of open TCP connections in a network translation device, the number of licenses available for an application, or the maximum number of jobs allowed in a batch queue. In addition to navigating the well-known resources, an execution engine must also be capable of recognizing and adjusting to secondary resource limitations.

Semantics of Failure. In any kind of distributed system, failures are not only common, but also hard to defi ne. It isn't always clear what constitutes failure, whether the failure is permanent or temporary, and whether a localized failure will affect other running pieces of the workload. If the engine managing the workflow doesn't know the structure of the problem, the partitioning, and the specifi cs about the job that failed, it will be almost impossible to recover cleanly. This is problematic, because expanding system size dramatically increases our chances of running into failures; no set of any signifi cant number of machines has all of them online all the time.

It should be clear that our concern in this work is *not* how to fi nd the optimal parallelization of a workload under ideal conditions. Rather, our goal is to design robust abstractions that result in reasonable performance, even under some variation in the environment.

4 An All-Pairs Implementation

To avoid the problems listed above, we propose that heavy users of shared computing systems should be given an *abstraction* that accepts a specification of the problem to be executed, and an *engine* that chooses how to implement the specification within the available resources. In particular, an abstraction must convey the data needs of a workload to the execution engine.

Figure 2 shows the difference between using a conventional computing cluster and computing with an abstraction. In a conventional cluster, the user specifies what jobs to run *by name*. Each job is assigned a CPU, and does I/O calls on demand with a shared file system. The system has no idea what data a job will access until runtime. When using an abstraction like All-Pairs, the user specifies both the data and computation needs, allowing the system to intelligently partition and distribute the data, then dispatch the computation.

We have constructed a prototype All-Pairs engine that runs on top of a conventional batch system and exploits the local storage connected to each CPU. This executes in four stages:
(1) Model the system. (2) Distribute the data. (3) Dispatch batch jobs. (4) Clean up the system.

Stage 1: Model the System. In a conventional system, it is difficult to predict the performance of a workload, because it depends on factors invisible to the system, such as the detailed I/O behavior of each job, and contention for the network. Both of these factors are minimized when using an abstraction that exploits initial efficient distribution followed by local storage instead of run time network access.

The engine measures the input data to determine the size **s** of each input element and the number of elements **n** and **m** in each set. (For simplicity, we assume here that the sets have the same size elements.) The provided function may be tested on a small set of data to determine the typical runtime **t** of each function call. Several fixed parameters are coded into the abstraction by the system operators: the bandwidth **B** of the network and the dispatch latency **D** of the batch software. Finally, the abstraction must choose the number of function calls **c** to group into each batch job, and the number of hosts **h** to harness.

The time to perform one transfer is simply the total amount of data divided by the bandwidth. Distribution by a spanning tree is $O(log_2(h))$, so the total time to transfer data is:

$$T_{data} = \frac{(n+m)s}{B} log_2(h)$$

The total number of batch jobs is nm/c, the runtime for each batch job is $D + ct$, and the total number of hosts is h, so the total time needed to compute on the staged data is:

$$T_{compute} = \frac{\frac{nm}{c}(D+ct)}{h}$$

However, because the batch scheduler can only dispatch one job every D seconds, each job start will be staggered by that amount, and the last host will complete $D(h-1)$ seconds after the first host to complete. Thus, the total estimated turnaround time is:

$$T_{turnaround} = \frac{(n+m)s}{B} log_2(h) + \frac{nm}{c}(D+ct)h + D(h-1)$$

Now, we may choose the free variables c and h to minimize the turnaround time. We employ a simple hill-climbing optimizer that starts from an initial value of c and h and chooses adjacent values until a minimum T is reached. Some constraints on c and h are necessary. Clearly, h cannot be greater than the total number of batch jobs or the available hosts. To bound the cost of eviction in long running jobs, c is further constrained to ensure that no batch job runs longer than one hour. This is also helpful to enable a greater degree of scheduling flexibility in a shared system where preemption is undesirable.

The reader may note that we choose the word *minimize* rather than *optimize*. No model captures a system perfectly. In this case, we do not take into account performance variations in functions, or the hazards of network transfers. Instead, our goal is to avoid pathologically bad cases.

Table 1 shows sample values for h and c picked by the model for several sample problems, assuming that 400 is the maximum possible value for h. In each case, we modify one parameter (shown in bold) to demonstrate that it is not always best to use a fi xed number of hosts.

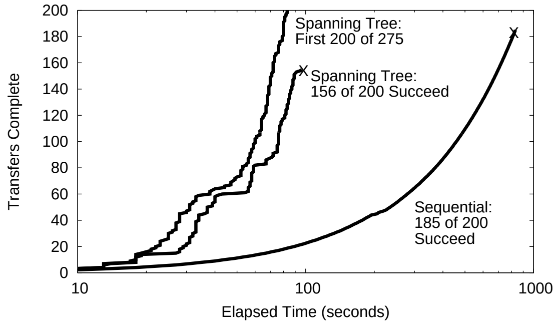
n, m	t	s	B	D	**h**	**c**
4000	1	10	100	10	400	2000
4000	1	10	**1**	10	138	2000
1000	1	10	100	10	303	3000
1000	1	**1000**	100	10	34	3000
100	1	10	100	10	31	300
100	**100**	10	100	10	315	25

Table 1: **h** and **c** Choices for Selected Confi gurations

Stage 2: Distribute the Data. For a large computation workload running in parallel, fi nding a place for the computation is only one of the considerations. Large data sets are usually not replicated to every compute node, so we must either prestage the data on these nodes or serve the data to them on demand. Figure 3 shows our technique for distributing the input data. A *file distributor* component is responsible for pushing all data out to a selection of nodes by a series of directed transfers forming a spanning tree with transfers done in parallel. The algorithm is a greedy work list. As each transfer completes, the target node becomes a source for a new transfer, until all are complete.

However, in a shared computing environment, these data transfers are never guaranteed to succeed. A transfer might fail outright if the target node is disconnected, misconfi gured, has different access controls or is out of free space. A transfer might be signifi cantly delayed due to competing traffi c for the shared network, or unexpected loads on the target system that occupy the CPU, virtual memory, and fi lesystem. Delays are particularly troublesome, because we must choose some timeout for each transfer operation, but cannot tell whether a problem will be briefly resolved or delay forever. A better solution is to set timeouts to a short value, request more transfers than needed, and stop when then target value is reached.

Figure 3 shows results for distribution of a 50 MB fi le to 200 nodes of our shared computing system. We show two particularly bad (but not uncommon) examples. A sequential transfer to 200 nodes takes linear time, but only succeeds on 185 nodes within a fi xed timeout of 20 seconds per transfer. A spanning tree transfer to 200 nodes completes much faster, but only succeeds on 156 nodes, with the same timeout. But, a spanning tree transfer instructed to accept the fi rst 200 of 275 possible transfers completes even faster with the same timeout. A “fi rst N of M” technique yields better throughput and better effi ciency in the case of a highly available pool, and allows



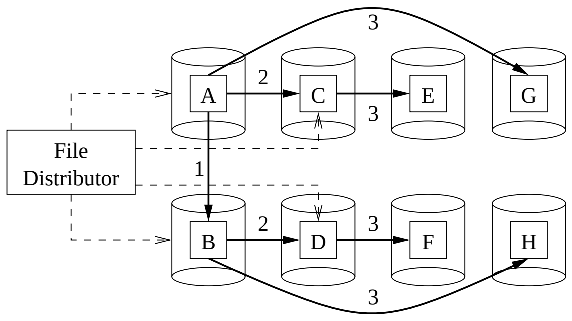


Figure 3: File Distribution via Spanning Tree

An efficient way to distribute data to many nodes of a system is to build a spanning tree. In the example on the right, a file distributor initiates transfers as follows: (1) A transfers to B, (2) AB transfer to CD, (3) ABCD transfer to EFGH. The graph on the right compares the performance transferring data to 200 hosts using sequential transfers and a spanning tree. In a shared system, failures are highly likely, so better performance is achieved by accepting the first 200 of 275 hosts.

more leeway than in sub-ideal environments.

Stage 3: Dispatch batch jobs. After transferring the input data to a selection of nodes, the All-Pairs engine then constructs batch submit scripts for each of the grouped jobs, and queues them in the batch system, with instructions to run on nodes where the data is available. Although the batch system has primarily responsibility at this stage, there are two ways in which the higher level abstraction provides added functionality.

An abstraction can monitor the environment and determine whether local resources are overloaded. An important example of this is the robustness of batch queues. As noted above, it may not be a good idea to queue 10,000 batch jobs at once, even if that much work is necessary. In this case, the abstraction can limit the number of jobs in the queue at a time to $2h$, which allows for all hosts to be harnessed without overloading the queue. In addition, if `submit_job` should fail under heavy load, the engine can back off and resubmit.

Another facet of management that the abstraction's structure gives access to is estimating completion time. If the time per function instance is approximately correct, we have a general idea from the model how long something should take to run. This guide, combined with comparisons versus other similarly-sized jobs, give the abstraction the ability to determine when jobs are being unusually slow (indicative of an error on the underlying worker node, or with the grid itself). From here, the engine can submit duplicate jobs, or just kill and resubmit. In this way, the abstraction and the model can prevent delays that might impact the workload's turnaround.

Stage 4: Collect results and clean up. As the batch system completes jobs, the abstraction engine must collect the results and assemble them into an appropriate structure, typically a single fi le listing all results in a canonical order. This presents another opportunity to add fault-tolerance: if the output of any function does not match a template provided by the user, the job can be resubmitted and the error corrected before showing results to the user.

When a workload has completed, the engine is responsible for deleting all data placed in stage 2. This is critical for executing in a shared environment, as artifacts left on shared resources tend to be a disincentive for the owners' continued sharing of those resources.

5 Evaluation

Environment. To evaluate the concept of using abstractions for data intensive computing, we have constructed a prototype All-Pairs engine and employed it with biometric and data mining applications in a shared computing environment at the University of Notre Dame. We operate a shared computing system consisting of about 500 CPUs connected to 300 disks. We use Condor [20] to manage the CPUs and Tactical Storage [19] to manage and access all of the local disks.

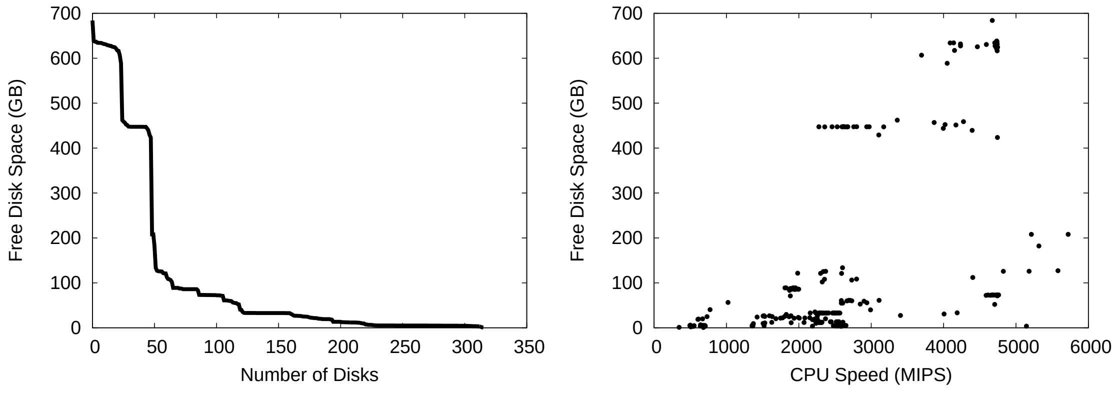
This is a highly shared computing environment consisting of a mix of research clusters, personal workstations, student lab workstations, and classroom display systems, all connected by shared networks that serve many more machines for general Internet access. Figure 4 also shows that it is highly heterogeneous. New machines are added to the system and old machines are removed on a daily basis. As a result, a small number of machines have a very large amount of free disk space, while a large number of machines have a few GB free. CPU speed ranges from 500-6000 MIPS, and is not strongly correlated with disk space.

Configurations. We evaluate two implementations of the All-Pairs abstraction. The first we call *active storage*, which is exactly the implementation described above. In this mode, the data is prestaged to local filesystems on each compute node. The computation references the local copy of the data, avoiding a long network trip for each access, and distributes the I/O load among the resources. As a tradeoff, the data must be prestaged to the compute nodes, and the resources must meet the disk requirements of the data set to be considered for inclusion in the computation pool. As the data sizes become larger, fewer CPUs become available.

We compare this to *demand paging*, which is the traditional cluster architecture with a central filesystem. In this mode, when a batch job issues a system call, the data is pulled on demand into the client's memory. This allows execution on nodes with limited disk space, since only two data set members must be available at any given time, and those will be accessed over the network, rather than on the local disk. In this mode, we hand-optimize the size of each batch job and other parameters to achieve the best comparison possible.

Metrics. Evaluating the performance of a large workload running in a shared computing system has several challenges. In addition to the heterogeneity of resources, there is also significant time variance in the system. In a shared computing environment, interactive users have first priority for CPU cycles. A number of batch users may compete for the remaining cycles in unpredictable ways. In our environment, anywhere between 200 and 400 CPUs were available at any given time. Figure 5(A) shows an example of this. The timeline of two workloads of size 2500x2500 is compared, one employing demand paging, and the other active storage. In this case, the overall performance difference is clearly dramatic: active storage is faster, even using fewer resources. However, the difference is a little tricky to quantify, because the number of CPUs varies over time, and there is a long tail of jobs that complete slowly, as shown in Figure 5(B).

To accommodate the variance, we present two results. The first is simply the turnaround time of the entire workload, which is of most interest to the end user. Because of the unavoidable variances, small differences in turnaround time should not be considered significant (although we do show very large differences for large problem sizes.) A more robust metric is *efficiency*, which we compute as the total number of cells in the result (i.e. the number of function evaluations), divided by the total CPU time (i.e. the area under the curve in Figure 5(A)). Higher efficiencies represent better use of the available resources.



(A) **(B)**

Figure 4: Resource Variance in a Shared Computing System
A shared computing system has a high degree of heterogeneity in available resources. 4(A) shows the distribution of free disk space across our system, ranging from a few MB to hundreds of GB per disk. 4(B) shows the weak relationship between CPU speed and available space. As a result, the selection of storage for a workload may have a significant effect on the available computation power, and vice versa.

Results. Figure 6 shows a comparison between the active storage and demand paging implementations for the biometric workload described above in the list of applicable problems. For the 4010x4010 standard biometric workload, the completion time is faster for active storage by a factor of two, and its effi ciency is much higher as well. Figure 7 shows the performance of the data mining application. For the largest problem size, 1000x1000, this application also shows a factor of two better turnaround time and more than that in effi ciency for the active storage implementation.

Finally, we consider an application with a very high data requirement – over 20 MB of data per 1 second of computation on that data. Although this application is synthetic, chosen to have ten times the biometric data rate, it is relevant as processor speed is increasing faster than disk or network speed, so applications will continue to be increasingly data hungry. Figure 8 shows for this third workload another example of active storage performing better than demand paging on all non-trivial data sizes.

For small problem sizes on each of these three applications, the completion times are similar for the two data distribution algorithms. The central server is able to serve the requests from the limited number of compute nodes for data suffi ciently to match the cost of the prestaging plus the local access for the active storage implementation.

For larger problem sizes, however, the demand paging algorithm is not as effi cient because the server cannot handle the load of hundreds of compute nodes and the network is overloaded with sustained requirements to keep all the compute nodes running at full speed. Also, the data set size passes the memory of the server, and because all the compute nodes will be in different stages, the server will thrash pieces of the data set in and out of memory to serve competing jobs. For active storage, however, the load factor is only 1 and there is no thrashing. Thus, the initial cost of data prestaging is made up for by effi cient access to the data over the course of thousands to millions of comparisons. The completion times diverge at approximately the same point where the active storage effi ciency surpasses the demand paging effi ciency.

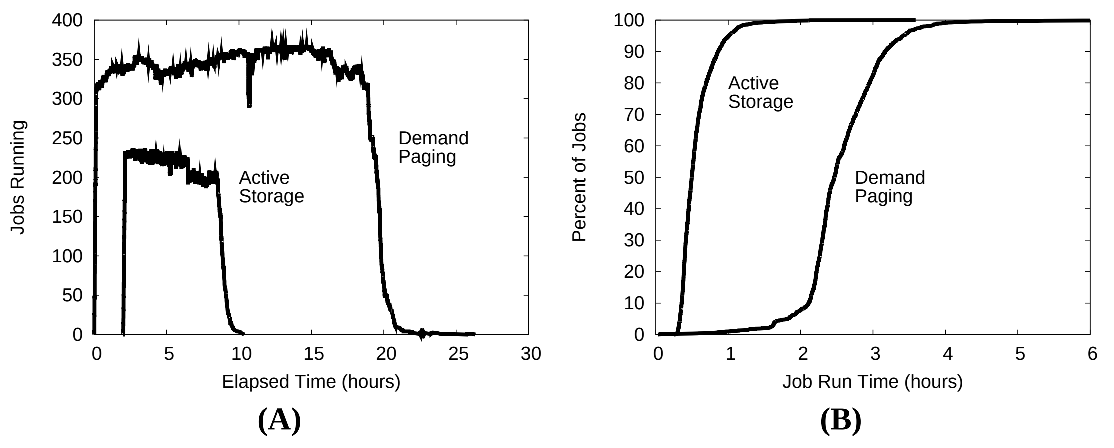


Figure 5: Time Variance in a Shared Computing System
Evaluating workloads in a shared computing system is troublesome, because of the variance of available resources over time. 5(A) shows the CPUs assigned to two variants of the same 2500x2500 All-Pairs problem run in demand paging and active storage modes. 5(B) shows the distribution of job run times for each workload. In this case, the active storage implementation is significantly better, but a quantitative evaluation is complicated by the variance in number of CPUs and the long tail of runtimes that occurs in a distributed system. To accommodate this variance, we also compute the efficiency *metric described in the text.*

In each of the graphs, the actual turnaround time for the active storage algorithm is longer than the modeled time. The model does not consider suspensions and evictions on batch systems, so the existence of these in actual systems causes higher turnaround times than predicted. Additionally, average execution times may not match the predicted execution time when there is heterogeneity in the system so many machines will complete a single function instance slower than the benchmarked (or user-given) time. Finally, the model assumes that h compute nodes are always available, which is not always the case due to disk availability and competition from other users.

6 Related Work

We have used the example of the All-Pairs abstraction to show how a high level interface to a distributed system improves both performance and usability dramatically. All-Pairs is not a universal abstraction; there exist other abstractions that satisfy other kinds of applications, however, a system will only have robust performance if the available abstraction maps naturally to the application.

For example, Bag-of-Tasks is a simple and widely used abstraction found in many systems, of which Linda [1], Condor [14], and Seti@Home [18] are just a few well-known examples. In this abstraction, the user simply states a set of unordered computation tasks and allows the system to assign them to processors. This permits the system to retry failures [4] and attempt other performance enhancements. [8]. Bulk-Synchronous-Parallel (BSP) [7] is a variation on this theme.

Bag-of-Tasks is a powerful abstraction for computation intensive tasks, but ill-suited for All-Pairs problems. If we attempt to map an All-Pairs problem into a Bag-of-Tasks abstraction by e.g.

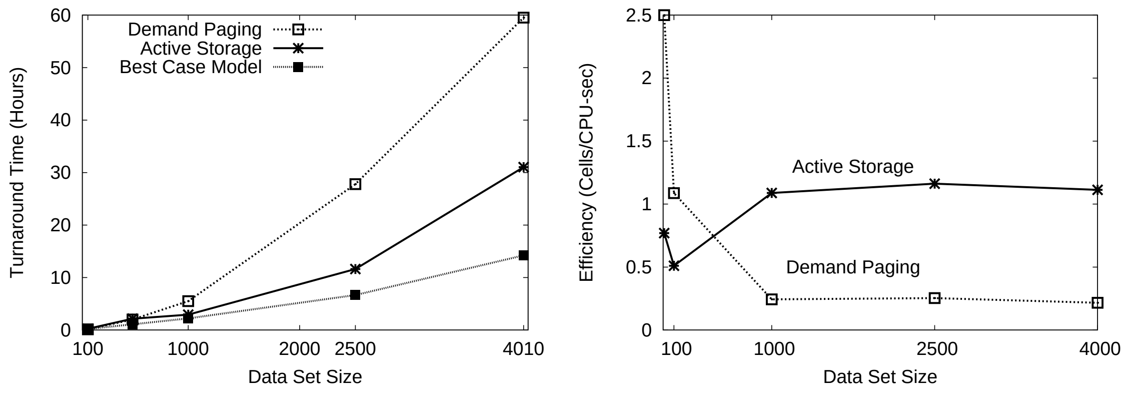


Figure 6: Performance of a Biometric All-Pairs Workload
The biometric face comparison function takes 1s to compare two 1.25MB images.

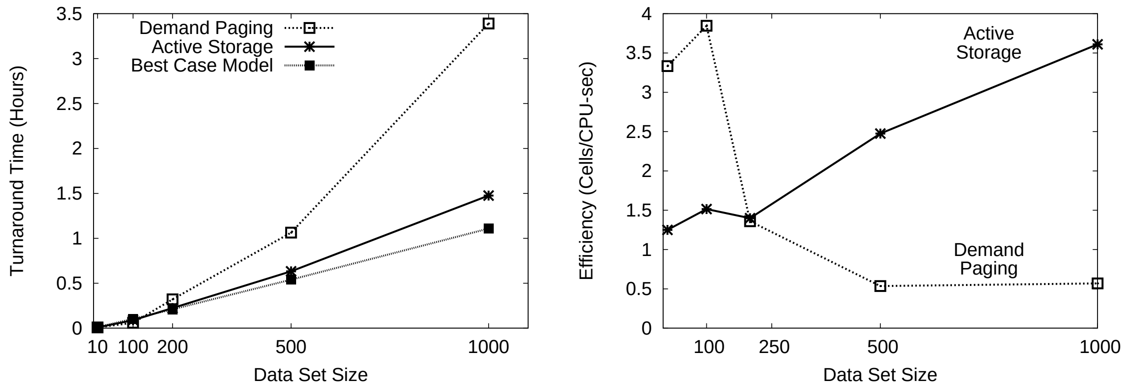


Figure 7: Performance of a Data Mining All-Pairs Workload
The data mining function takes .25 seconds to compare two 700KB items.

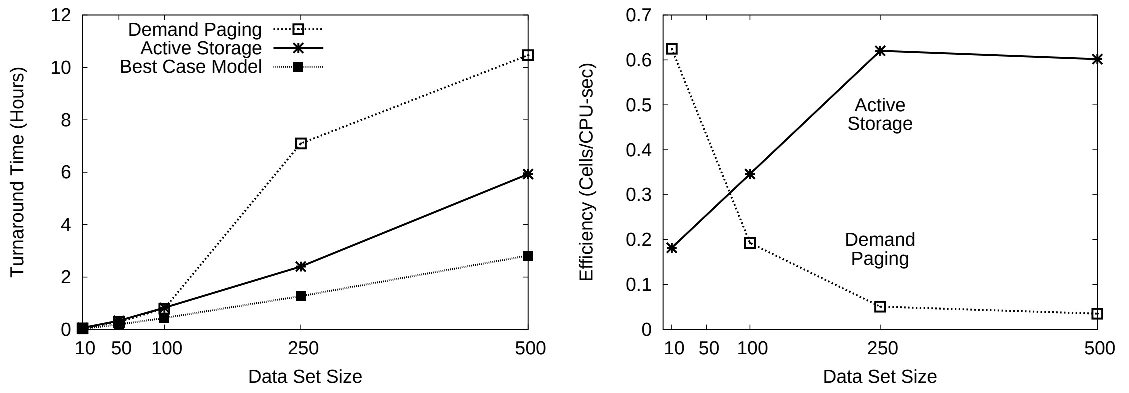


Figure 8: Performance of a Synthetic All-Pairs Workload
This function takes 1s for two 12MB data items, 10 times the data rate of the biometric workload.

making each comparison into a task to be scheduled, we end up with all of the problems described in the introduction. Bag-of-Tasks is insufficient for a data-intensive problem.

Map-Reduce [9] is closer to All-Pairs in that it encapsulates both the data and computation needs of a workload. This abstraction allows the user to apply a `map` operator to a set of name-value pairs to generate several intermediate sets, then apply a `reduce` operator to summarize the intermediates into one or more final sets. Map-Reduce allows the user to specify a very large computation in a simple manner, while exploiting system knowledge of data locality.

We may ask the same question about All-Pairs and Map-Reduce: *Can Map-Reduce solve an All-Pairs problem?* The answer is: *Not efficiently.* We might attempt to do this by executing $AllPairs(A, B, F)$ as $Map(F, S)$ where $S = ((A_1, B_1), (A_1, B_2)...)$. This would result in a large workload where each individual job refers to one element from both sets, with no particular effort in advance to distribute data appropriately and co-location programs with the needed data. Again, when the workload does not match the available abstraction, poor performance results.

Clusters can also be used to construct more traditional abstractions, albeit at much larger scales. For example, data intensive clusters can be equipped with abstractions for querying multidimensional arrays [6], storing hashtables [12] and B-trees [16], searching images [13], and sorting record-oriented data [3].

The field of grid computing has produced a variety of abstractions for executing large workloads. Computation intensive tasks are typically represented by a directed acyclic graph (DAG) of tasks. A system such as Pegasus [10] converts an abstract DAG into a concrete DAG by locating the various data dependencies and inserting operations to stage input and output data. This DAG is then given to an execution service such as Condor's DAGMan [20] for execution. All-Pairs might be considered a special case of a large DAG with a regular grid structure. Alternatively, an All-Pairs job might be treated as a single atomic node in a DAG with a specialized implementation.

Other abstractions place a greater focus on the management of data. Chimera [11] presents the abstraction of *virtual data* in which the user requests a member of a large data set which may already exist in local storage, be staged from a remote archive, or be created on demand by running a computation. Swift [21] and GridDB [15] build on this idea by providing languages for describing the relationships between complex data sets.

7 Future Work

Several avenues of work remain open.

Very Large Data Sets. The All-Pairs engine described above requires that at least one of the data sets fits in its entirety on each compute node. This limits the number of compute nodes, perhaps to zero, available to a problem for large data sets. If we can only distribute partial data sets to the compute nodes, they are not equal in capability. Thus, the data prestaging must be done with more care, and there must be more structure in the abstraction to monitor which resources have which data. This is required to correctly direct the batch system's scheduling.

Data Persistence. It is common for a single data set to be the subject of many different workloads in the same community. If the same data set will be used many times, why transfer and delete it for each workload? Instead, we could use the cluster with embedded storage as both a data repository and computing system. In this scenario, the challenge becomes managing competing data needs. Every user will want their data on all nodes, and the system must enforce a resource sharing policy by increasing and decreasing the replication of each data set.

8 Conclusions

The problems described above are indicative of the ease for non-expert users to make naive choices that inadvertently result in poor performance and abuse of the shared environment. We proposed that an abstraction lets the user specify only what he knows about his workload: the input sets and the function. This optimized abstraction adjusts for grid and workload parameters to execute the workload effi ciently. We have shown an implementation of one such abstraction, All-Pairs. Using an active storage technique to serve data to the computation, the All-Pairs implementation avoids pathological cases and can signifi cantly outperform hand-optimized demand paging attempts as well. Abstractions such as All-Pairs can make grid computing more accessible to non-expert users, while preventing the shared consequences of mistakes that non-experts might otherwise make.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[2] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 3(215):403–410, Oct 1990.

[3] A. Arpaci-Dussea, R. Arpaci-Dusseau, and D. Culler. High performance sorting on networks of workstations. In *SIGMOD*, May 1997.

[4] D. Bakken and R. Schlichting. Tolerating failures in the bag-of-tasks programming paradigm. In *IEEE International Symposium on Fault Tolerant Computing*, June 1991.

[5] R. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *World Wide Web Conference*, May 2007.

[6] M. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. Middleware for fi ltering very large scientifi c datasets on archival storage systems. In *IEEE Symposium on Mass Storage Systems*, 2000.

[7] T. Cheatham, A. Fahmy, D. Siefanescu, and L. Valiani. Bulk synchronous parallel computing-a paradigm for transportable software. In *Hawaii International Conference on Systems Sciences*, 2005.

[8] D. da Silva, W. Cirne, and F. Brasilero. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Euro-Par*, 2003.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplifi ed data processing on large cluster. In *Operating Systems Design and Implementation*, 2004.

[10] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, B. Berriman, J. Good, A. Laity, J. Jacob, and D. Katz. Pegasus: A framework for mapping complex scientifi c workfbws onto distributed systems. *Scientifi c Programming Journal*, 13(3), 2005.

[11] I. Foster, J. Voeckler, M. Wilde, and Y. Zhou. Chimera: A virtual data system for representing, querying, and automating data derivation. In *14th Conference on Scientifi c and Statistical Database Management*, Edinburgh, Scotland, July 2002.

[12] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *USENIX Operating Systems Design and Implementation*, October 2000.

[13] L. Huston, R. Sukthankar, R. Wickremesinghe, M.Satyanarayanan, G. R.Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *USENIX File and Storage Technologies (FAST)*, 2004.

[14] J. Linderoth, S. Kulkarni, J.-P. Goux, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *IEEE High Performance Distributed Computing*, pages 43–50, Pittsburgh, Pennsylvania, August 2000.

[15] D. Lui and M. Franklin. GridDB a data centric overlay for scientifi c grids. In *Very Large Databases (VLDB)*, 2004.

[16] J. MacCormick, N. Murphy, M. Najork, C. Thekkath, and L. Zhou. Boxwood: Abstractions as a foundation for storage infrastructure. In *Operating System Design and Implementation*, 2004.

[17] P. Phillips and et al. Overview of the face recognition grand challenge. In *IEEE Computer Vision and Pattern Recognition*, 2005.

[18] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A new major SETI project based on project serendip data and 100,000 personal computers. In *5th International Conference on Bioastronomy*, 1997.

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

[20] 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 Infrastructure a Reality*. John Wiley, 2003.

[21] Y. Zhao, J. Dobson, L. Moreau, I. Foster, and M. Wilde. A notation and system for expressing and executing cleanly typed workfbws on messy scientifi c data. In *SIGMOD*, 2005.