All-Pairs: An Abstraction for Data Intensive Computing on Campus Grids

Christopher Moretti, Hoang Bui, Karen Hollingsworth, Brandon Rich, Patrick Flynn, and Douglas Thain
Department of Computer Science and Engineering, University of Notre Dame

Abstract— **Today, campus grids provide users with easy access to thousands of CPUs. However, it is not always easy for non-expert users to harness these 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 argue that campus grids 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 applications in biometrics, bioinformatics, and data mining. We demonstrate that an optimized All-Pairs abstraction is both easier to use than the underlying system, achieves performance orders of magnitude better than the obvious but naive approach, and is both faster and more efficient than a tuned conventional approach. This abstraction has been in production use for one year on a 500-CPU campus grid at the University of Notre Dame, and has been used to carry out a groundbreaking analysis of biometric data.**

I. Introduction

MANY FIELDS of science and engineering have the potential to use large numbers of CPUs to attack problems of enormous scale. Campus-scale computing grids are now a standard tool employed by many academic institutions to provide large scale computing power. Using middleware such as Condor [40] or Globus [21], many disparate clusters and standalone machines may be joined into a single computing system with many providers and consumers. Today, campus grids of about one thousand machines are commonplace [41], and are being grouped into larger structures, such as the 20,000-CPU Indiana Diagrid, and the 40,000-CPU Open Science Grid. [36]

Campus grids have the unique property that consumers of the system must always defer to the needs of the resource providers. For example, if a desktop computer is donated to the campus grid, then a visiting job may use it during idle times, but will be preempted when the owner is busy at the keyboard. If a research cluster is donated to the campus grid, visiting jobs may use it, but might be preempted by higher priority batch jobs submitted by the owner of the cluster. In short, the user of the system has access to an enormous number of CPUs, but must expect to be preempted from many of them as a normal condition.

Because of this property, scaling up an application to a campus grid is a non-trivial undertaking. Parallel libraries and languages such as MPI [18], OpenMP [14], and Cilk [8] are not usable in this context, because they do not explicitly address preemption and failure as a normal case. Instead, large workloads must be specified as a set of sequential processes connected by files. End users must carefully arrange the I/O behavior of their workloads. Bad configurations can result in poor performance, outright failure of the application, and abuse of physical resources shared by others. All too often, an end user composes a workload that runs correctly on one machine, then on ten machines, but fails disastrously on a thousand machines.

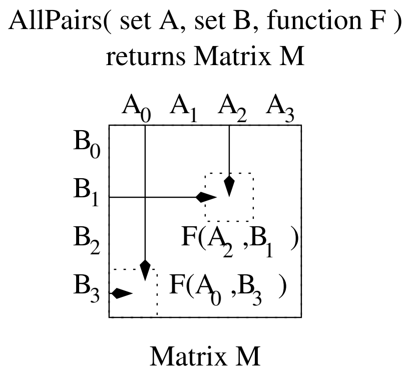
Providing an *abstraction* is one approach to avoiding these problems. An abstraction allows a user to declare a workload composed of multiple sequential programs and the data that they process, while hiding the details of how the workload will be realized in the system. Abstracting away details hides complications that are not apparent or important to a novice, limiting the opportunity for disasters. Because an abstraction states a workload in a declarative way, it can be realized within the grid in whatever way satisfies cost, policy, and performance constraints. Abstractions could also be implemented in other kinds of systems, such as dedicated clusters or multicore CPUs, but we do not address those here.

We have implemented one such abstraction – All-Pairs – for a class of problems found in many fields. All-Pairs is the Cartesian product of a large number of objects with a custom comparison function. While simple to state, it is non-trivial to carry out on large problems that require hundreds of nodes running for several days. All-Pairs is similar in spirit to other abstractions such as Dryad [28], Map-Reduce [16], Pegasus [17], and Swift [42], but it addresses a different category of applications.

Our implementation of All-Pairs is currently in production use on a 500 CPU campus grid at the University of Notre Dame, using Condor [40] to manage the CPUs and Chirp [39] to manage the storage. To demonstrate the performance benefits of using an abstraction, we compare two different implementations. The *conventional* implementation executes the specification by simply submitting a series of batch jobs that use a central file server to read data on demand and write outputs into files in the ordinary way. The *abstraction* implementation exploits the information found in the abstraction by efficiently distributing common data in advance, choosing an optimal granularity for decomposition, accessing local data copies, and storing the outputs in a custom data structure. By employing these techniques, large workloads run twice as fast using four to twelve times fewer resources than the conventional implementation. More importantly, users are prevented from making mistakes that harm other stakeholders.

We conclude with an example of a groundbreaking biometrics workload that employed All-Pairs to create the largest analysis of public biometric data to date, reducing the workload from an estimated sequential run time of 800 days to just over 10 days on campus grid. We conclude with a discussion of the techniques necessary to make All-Pairs robust for long-running jobs.

This work was first presented at the IEEE IPDPS 2008 conference [32]. This extended journal paper has new material on exploiting network topology for file distribution, output management using distributed data structures, experience and performance on a very large biometrics run, and a discussion of fault tolerance techniques.



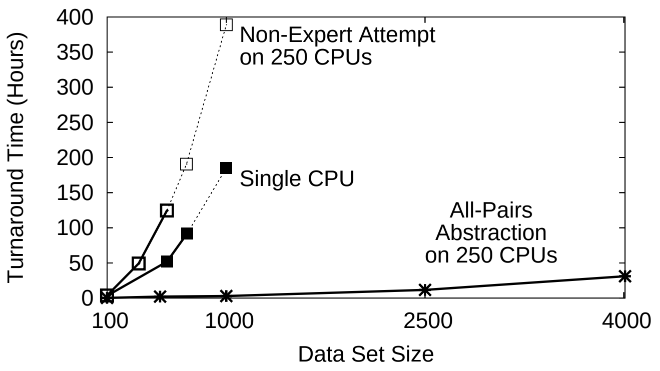


Fig. 1. The All-Pairs Problem
The All-Pairs abstraction compares all elements of sets A and B together using a custom function F, yielding a matrix M where each cell is the result of $F(A[i], B[j])$. The graph on the right shows the performance of an All-Pairs problem on a single machine, on 250 CPUs when attempted by a non-expert user, and on 250 CPUs when using the optimized All-Pairs abstraction.

II. 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, such that
M[i,j] = F(A[i],B[j]).

This abstraction is also known as the *Cartesian product* or *cross join* of sets A and B. Variations of All-Pairs 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. The function is sometimes, but not always, symmetric, so often it is enough to compute one half of the matrix. We are working with several different user communities that make use of All-Pairs computations.

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*. Multiple matrices generated on the same dataset can be used to quantitatively compare different comparison functions.

A typical All-Pairs problem in biometrics is to compare 4010 images of 1.25MB each from the Face Recognition Grand Challenge [35] to all others in the set, using functions that range from 1-20 seconds of compute time, depending on the algorithm in use. This workload requires 185 to 3700 CPU-days of computation, so it must be parallelized across a large number of CPUs in order to make it complete in reasonable time. Unfortunately, each CPU added to the system also needs access to the 5GB of source data. If run on 500 CPUs, the computation alone could be completed in 8.8 hours, but it would require 2.5TB of I/O. Assuming the filesystem could keep up, this would keep a gigabit (125MB/s) network saturated for 5.8 hours, rendering the grid completely unusable by anyone else. Addressing the CPU needs with massive parallelism simply creates a new bottleneck in I/O.

Data Mining is the study of extracting meaning from large datasets. One phase of knowledge discovery is reacting to bias or other noise within a set. In order to improve overall accuracy, 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.

Bioinformatics is the use of computational science methods to model and analyze biological systems. Genome assembly [25], [26] remains one of the most challenging computational problems in this field. A sequencing device can analyze a biological tissue and output its DNA sequence, a string on the set [AGTC]. However, due to physical constraints in the sequencing process, it is not produced in one long string, but in tens of thousands of overlapping substrings of hundreds to thousands of symbols. An assembler must then attempt to align all of the pieces with each other to reconstruct the complete string. All-Pairs is a natural way of performing the first step of assembly. Each string must be compared to all others at both ends, producing a very large matrix of possible overlaps, which can then be analyzed to propose a complete assembly.

Other Problems. Of course, All-Pairs is by no means a universal solution to distributed computing, but it is a common pattern that appears throughout science and engineering. The reader may note that some problems that *appear* to be All-Pairs may be algorithmically reducible to a smaller problem via techniques such as data clustering or filtering. [3], [6], [20], [27] In these 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 have outlined above, it is 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 requires brute force, and the challenge lies in providing interfaces to execute it effectively.

III. Why is All-Pairs Challenging?

Solving an All-Pairs problem 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 small datasets on a workstation, he or she connects to the campus grid, 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, this is a perfectly rational way to construct a large workload, because one would do exactly the same thing in a sequential or parallel programming language on a single machine. Unfortunately, it will likely result in very poor performance for the user, and worse, may result in the abuse of shared resources. 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 one user on a switched network, efficient use of resources might not be a concern. However, in a large campus grid shared by many users, a poorly configured workload will consume resources that might otherwise be assigned to waiting jobs. If the workload makes excessive use of the network, it may even halt other unrelated tasks. 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.

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 in a non-dedicated environment the data cannot easily be split into disjoint partitions: to complete the workload, each node needs all of the data. 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. For many I/O intensive problems, it may only make sense to harness ten CPUs, even though hundreds are available.

Data Distribution. After choosing the proper number of servers, we must then ask how to get the data to each computation. A campus grid usually makes use of an institutional file server or the submitting machine as a file server, as this makes it easy for programs to access data on demand. However, it is much easier to scale up the CPUs of a campus grid than it is to scale the capacity of a central file server. If the same input data will be re-used many times, then it makes sense simply to store the inputs on each local disk, getting better performance and scalability. Many dedicated clusters provide fixed 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.

Dispatch Latency. The cost to dispatching a job within a campus 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 system is under heavy load, dispatch latency can easily be measured in seconds. 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 job granularity 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 campus grid also has the possibility that a job can be preempted for a higher priority task, usually resulting in a rollback to the beginning of the job on another CPU. Short runs provide a kind of checkpointing, as a small result that is completed need not be regenerated. Long runs also magnify heterogeneity in the pool. For instance, jobs that should take 10 seconds on a typical machine but take 30 seconds on the slowest aren'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, the workload will incur a long delay waiting for the final job to complete on the slowest. Another downside to jobs that run for many hours is that it is difficult to discriminate between a healthy long-running job and a job that is stuck and not making progress. An abstraction has to determine the appropriate job granularity, noting that this depends on numerous factors of the job and of the grid itself.

Resource Limitations. Campus grids 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 made known to end users. However, systems also have more prosaic resources. Examples are the maximum number of open files 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 define. If a program exits with an error code, who is at fault? Does the program have a bug, or did the user give it bad inputs, or is the executing machine faulty? Is the problem transient or reproducible? Without any context about the workload and the execution environment, it is almost impossible for the system to take the appropriate recovery action. But, when using an abstraction that regulates the semantics of each job and the overall dataflow, correct recovery

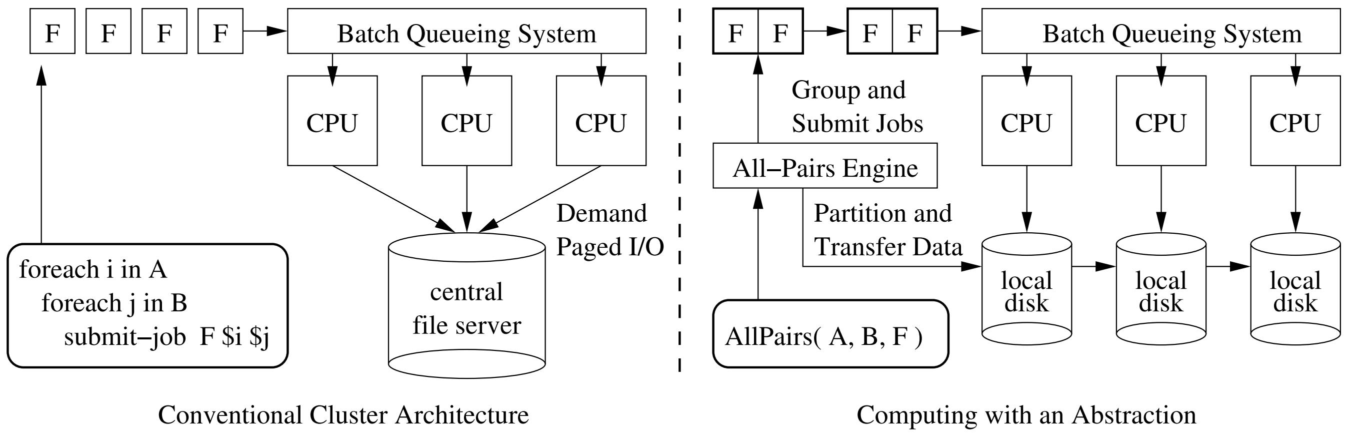


Fig. 2. Cluster Architectures Compared
When using a conventional computing cluster, the user partitions the workload into jobs, then a batch queue distributes jobs to CPUs where they access data from a central file server on demand. When using an abstraction like All-Pairs, the user states the high level 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.

is straightforward, as we show in Section VI below.

IV. An All-Pairs Implementation

To avoid the problems listed above, we propose that users of campus grids 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 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 jobs actually begin to issue system calls. When using an abstraction like All-Pairs, the user specifies both the data and computation needs, allowing the system to partition and distribute the data in a structured way, then dispatch the computation according to the data distribution.

We have constructed a prototype All-Pairs engine that runs on top of the Condor [40] distributed computing system and exploits the local storage connected to each CPU. The user invokes All-Pairs as follows:

```
AllPairs SetA SetB Function Matrix
```

where `SetA` and `SetB` are text files that list the set of files to process, `Function` is the function to perform each comparison, and `Matrix` is the name of a matrix where the results are to be stored. `Function` is provided by the end user, and may be an executable written in any language, provided that is has the following calling convention:

```
Function SetX SetY
```

where `SetX` and `SetY` are text files that list a set of files to process, resulting in a list of results on the standard output that name each element compared along with the comparison score:

```
A1 B1 0.53623
A1 B2 2.30568
A1 B3 9.19736
...
```

Note that we require the user's function be essentially a single-CPU implementation of All-Pairs. This is good for performance, because the actual execution time of of a single comparison could be significantly faster than the time needed to invoke an external program. It also improves usability, because the user may easily transition from a small job run on a workstation to a large job run on the campus grid.

Our implementation of All-Pairs operates in four stages: Modeling the system, distributing input data, managing batch jobs, and collecting the results. We describe each stage in turn.

A. Modeling the System

In order to decide how many CPUs to use and how to partition the work, we must have an approximate model of system performance. 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 access instead of remote network access.

We hasten to note that our goal in modeling is *not* to achieve optimal performance. This is essentially impossible in a large dynamic heterogeneous system where nothing is under our direct control. Rather, the best we can hope for is to avoid disasters by choosing the configuration that is optimal within the model.

The engine measures the input data to determine the size **s** of each input element and the number of elements **n** in each set (for simplicity, we assume here that the sets have the same number and size elements). The provided function is 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.

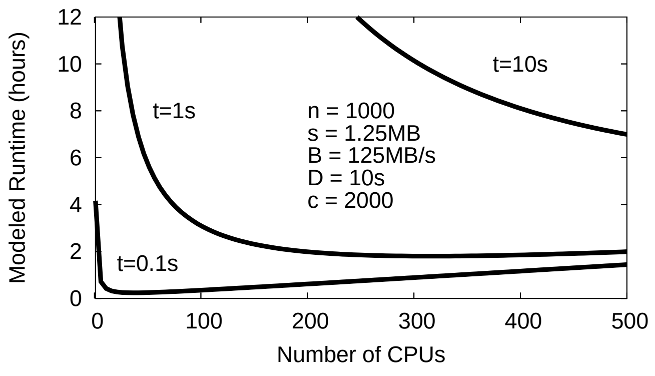


Fig. 3. Three Workloads Modeled
This graph compares the modeled runtime of three workloads that differ only in the time (t) to execute each function. In some configurations, additional parallelism has no benefit.

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

$$T_{distribute} = \frac{2ns}{B} log_2(h)$$

The total number of batch jobs is n^2/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:

$$T_{compute} = \frac{\frac{n^2}{c}(D + ct)}{h}$$

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 turnaround time is:

$$T_{turnaround} = \frac{2ns}{B} log_2(h) + \frac{n^2}{ch}(D + ct) + D(h - 1)$$

Now, we may choose the free variables c and h to minimize the turnaround time. 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 must be either a multiple or even divisor of a result row, to simplify job partitioning. 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.

Figure 3 shows the importance of modeling the orders of magnitude within the abstraction. Suppose that All-Pairs is invoked for a biometric face comparison of 1000x1000 objects of 1.25MB each, on a gigabit ethernet (125MB/s) network. Depending on the algorithm in use, the comparison function could have a runtime anywhere between 0.1s and 10s. If the function takes 0.1 seconds, the optimal number of CPUs is 38, because the expense of moving data and dispatching jobs outweighs the benefit of any additional parallelism. If the function takes one second, then the system should harness several hundred CPUs, and if it takes ten, all available CPUs.

B. Distributing the Data

Each partition of the workload must have some method for getting its input data. Batch systems are usually coupled with a traditional file system, such that when a job issues I/O system calls, the execution node is responsible for pulling data from the storage nodes into the compute node. Because the abstraction is given the data needs of the workload in advance, it can implement I/O much more efficiently. To deliver all of the data to every node, we can build a spanning tree which performs streaming data transfers and completes in logarithmic time. By exploiting the local storage on each node, we avoid the unpredictable effects of network contention for small operations at runtime.

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. Figure 4 shows the algorithm, which is a greedy work list. The data is pushed to one node, which then is directed to transfer it to a second. Two nodes can then transfer in parallel to two more, and so on. Each node in the system runs a lightweight fileserver called Chirp [39] which provides the access interface, performs access control, and executes directed transfers.

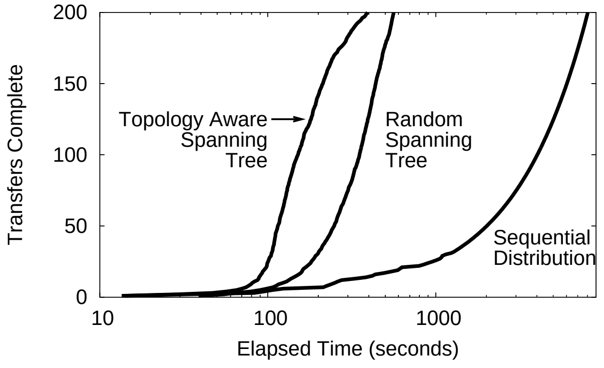
Dealing with failures is a significant concern for pushing data. Failures are quite common and impossible to predict. A transfer might fail outright if the target node is disconnected, misconfigured, has different access controls or is out of free space. A transfer might be significantly delayed due to competing traffic for the shared network, or unexpected loads on the target system that occupy the CPU, virtual memory, and filesystem. Delays are particularly troublesome, because we cannot tell whether a problem will be briefly resolved or delay forever.

A greedy work list is naturally fault tolerant. If any one transfer fails outright or is delayed, the remaining parallel branches of the spanning tree will reach other parts of the campus grid. We have found it most effective to simply give the distributor a goal of h hosts, and then let it operate until that number is reached, cancel any outstanding transfers, and then list the hosts actually reached.

Of course, a campus grid does not have a uniform network topology. Transfers may be fast between machines on one switch, but become slower as transfers reach across routers and other network elements. In the worst case, the file distributor might randomly arrange a large number of transfers that saturate a shared network link, rendering the system unusable to others.

To prevent this situation, we allow the system administrator to provide the abstraction with a simplified topology in the form of a "network map", which simply states which machines are connected to the same switch. The file distributor algorithm is slightly refined in two ways. First, the distributor will prefer to transfer data between clusters before transferring within clusters, assuming that the former are slower and thus should be performed sooner so as to minimize the makespan. Second, the distributor will not allow more than one transfer in or out of a given cluster at once, so as to avoid overloading shared network links.

The performance of file distribution is shown in Figure 4. Here, a 500MB dataset is transferred to the first 200 available hosts in our campus grid, recording the elapsed time at which each single transfer is complete. Each of three distribution techniques is performed ten times, and the average at each host is shown. Sequential distribution takes 8482 seconds to complete. A fully random spanning tree takes 585 seconds, while a topology aware tree takes 420 seconds.



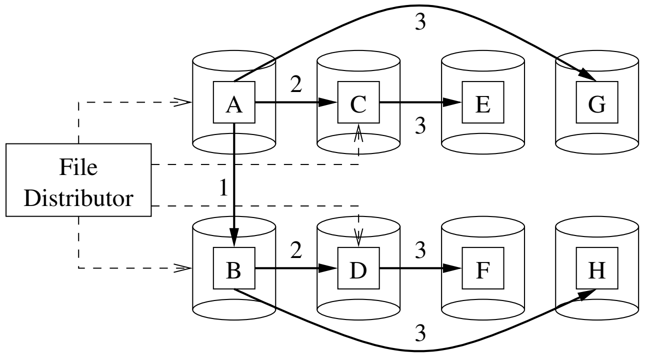


Fig. 4. File Distribution via Spanning Tree

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 left compares the performance of transferring data to the first 200 available hosts using sequential transfers, a random spanning tree, and a topology aware spanning tree.

Is it really necessary to distribute *all* of the data to every node? We could simply distribute the minimum amount of data to allow each node to run its job. As we will show, distributing all of the data via spanning tree is *faster* than distributing the minimum fragment of data from a central server, and it also improves the fault tolerance of the system. Table I summarizes the result.

Proof: Consider a cluster of h reliable hosts with no possibility of preemption or failure. The *fragment method* minimizes the amount of data sent to each host by assigning each host a square subproblem of the All-Pairs problem. Each subproblem requires only a fragment of data from each set to complete. So, both data sets are divided into into f fragments, where $f = \sqrt{h}$. Each host then needs n/f data items of size s from each set delivered from the central file server. Dividing by the bandwidth B yields the total time to distribute the data fragments:

$$T_{fragment} = \frac{2nsh}{Bf} = \frac{2ns}{B}\sqrt{h}$$

Compare this to the *spanning tree method* described above:

$$T_{distribute} = \frac{2ns}{B}log_2(h)$$

Because $log_2(h) << \sqrt{h}$, the spanning tree method is faster than the minimum fragment method for any number of hosts in a reliable cluster without preemption or failure. Of course, the total amount of data transferred is higher, and the dataset must fit entirely on a sufficient number of disks. However, as commodity workstation disks now commonly exceed a terabyte and are typically underutilized [19], this has not been a significant problem.

As we have noted above, a campus grid is a highly unreliable environment. The fragment method is even worse when we consider failures. Because it delivers the minimum amount of data to any one host, there is no other location a job can run if the data is not available. With the spanning tree method, any job can run on any node with the data, so the solution is more naturally fault tolerant.

C. Dispatching Batch Jobs

After transferring the input data to a suitable 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 those nodes where the data is available. Each batch job consists of the user's function and the All-Pairs *wrapper*, shown in Figure 5. The wrapper is responsible for executing the user's function on the appropriate partition of the job and writing the results in batch to the output matrix, described in detail below.

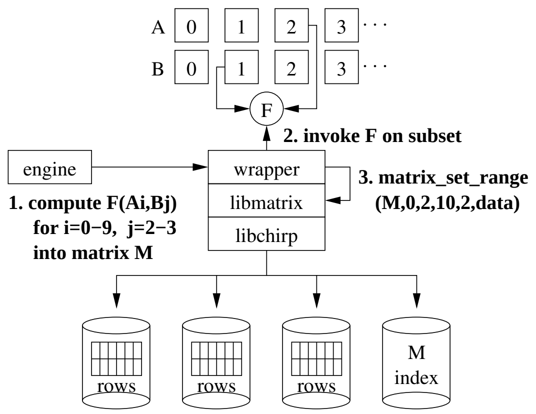
TABLE I
COMPARISON OF DATA DISTRIBUTION TECHNIQUES

Distribution Method	**Total Time**	**Data Transferred**	**Fault Tolerant?**
Fragment	$(ns/B)\sqrt{h}$	$ns\sqrt{h}$	No
Spanning Tree	$(ns/B)log_2(h)$	nsh	Yes

Although we rely heavily on the batch system to manage the workload at this stage, the framework still has two important responsibilities: local resource management and error handling.

The All-Pairs engine is responsible for managing local resources on the submitting machine. If a workload consists of hundreds of thousands of partitions, it may not be a good idea to instantly materialize all of them as batch jobs for submission. Each materialized job requires the creation of several files in the local filesystem, and consumes space in the local batch queue. Although Condor is capable of queueing hundreds of thousands of jobs reliably, each additional job slows down queue management and thus scheduling performance. When jobs complete, there is the possibility that they produce some large error output or return a core dump after a crash. Instead of materializing all jobs simultaneously, the engine throttles the creation of batch jobs so that they queue only has twice as many jobs as CPUs. As jobs complete, the engine deletes the output and ancillary files to manage the local filesystem.

The engine and the wrapper together are responsible for handling a large number of error conditions. Again, Condor itself can silently handle problems such as the preemption of a job for a higher priority task or the crash of a machine. However, because it has no knowledge of the underlying task, it cannot help when a job fails because the placed input files have been removed, the execution machine does not have the dynamic libraries needed by


A
0
1
2
3
B
0
1
2
3
F
2. invoke F on subset
engine
wrapper
libmatrix
libchirp
3. matrix_set_range (M,0,2,10,2,data)
1. compute F(Ai,Bj) for i=0−9, j=2−3 into matrix M
rows
rows
rows
M index

```
matrix_create     ( host, path, width,
                    height, itemsize, nhosts );
matrix_open       ( host, path );
matrix_set_cell   ( matrix, i, j, data );
matrix_set_row    ( matrix, i, data );
matrix_set_col    ( matrix, j, data );
matrix_set_range  ( matrix, i, j,
                    width, height, data );
matrix_close      ( matrix );
matrix_delete     ( host, path );
```

Fig. 5. Detail of Local Job Execution
The engine *runs on the submitting node, directing each working node to compute a subset of the All-Pairs problem. On each node, a* wrapper *invokes the users function, buffers the results in memory, and then updates the distributed data structure with the results.*

the function, or a brief network outage prevents writing results to the matrix. Although events like this sound very odd, they are all too common in workloads that run for days on hundreds of machines. To address this, the wrapper itself is responsible for checking a number of error conditions and verifying that the output of the function is well formed. If the execution fails, the wrapper informs the engine through its exit code, and the engine can resubmit the job to run on another machine.

The engine can also handle the problem of jobs that run for too long. This may happen because the execution machine has some hardware failure or competing load that turns a 30 minute job into a 24 hour job. Although the runtime of an arbitrary batch job is impossible to predict in the general case, the engine has access to a model of the workload, as well as a distribution of runtimes, so it can cancel and resubmit jobs that fall far out of the range of normal execution times. To improve the "long tail" of jobs at the end of an execution, it could also submit duplicate jobs as in Map-Reduce [16], although we have not yet implemented this feature.

D. Collecting the Output

The output produced by an All-Pairs run can be very large. In our largest biometrics workload, a 60,000 by 60,000 comparison will produce 3.6 billion results. Each result must, at a minimum, contain an eight-byte floating point value that reflects the similarity of two images, for a total of 28.8GB of unformatted data. If we store additional data such as troubleshooting information for each comparison, the results may balloon to several hundred gigabytes. Our users run many variations of All-Pairs, so the system must be prepared to store many such results.

Although current workstation disks are one terabyte and larger, and enterprise storage units are much larger, several hundred gigabytes is still a significant amount of data that must be handled with care. It is not likely to fit in memory on a workstation and applying improper access patterns will result in performance many orders to magnitude slower than necessary. A user that issues many All-Pairs runs will still fill up a disk quite quickly.

The non-expert user who applies existing interfaces in the most familiar way will run into serious difficulties. We have encountered several non-expert users whose first inclination is to store each result as a separate file. Of course, this is a disastrous way to use a filesystem, because each eight byte result will consume one disk block, one inode, and one directory entry. If we store each row of results in a separate file in the function's text output format, we are still storing several hundred gigabytes of data, and still have a sufficiently large number of files that directory operations are painfully slow. Such a large amount of data would require the user to invest in a large storage and memory intensive machine in order to manipulate the data in real time.

Instead, the abstraction must guide users toward an appropriate storage mechanism for the workload. Output from All-Pairs jobs goes to a *distributed data structure* provided by the system. The data structure is a matrix whose contents are partitioned across a cluster of reliable storage nodes maintained separately from the campus grid. Data in the matrix is not replicated for safety, because the cluster is considered an intermediate storage location in which results are analyzed and then moved elsewhere for archival. In the event of failure leading to data loss, the All-Pairs run can easily be repeated.

The matrix is named by the location of an *index file* that indicates the dimensions of the array and the location of each of the data files. It is accessed through a library interface shown in Figure 5. Each chunk of the matrix is a subset of complete rows, stored in row-major order. Clearly, row-major access is most efficient: A row read or write results in a single sequential I/O request to one host. Column-major access is still possible: A column read or write results in a strided read or write performed on all hosts in parallel. This is unavoidably inefficient on a single disk, because the underlying filesystem will only access a few bytes in each disk block, but can take advantage of the hardware parallelism. Individual cell reads are also possible, but are inefficient for the same reason. Regardless, I/O performance of all access methods is improved by access to parallel I/O and memory capacity.

Figure 6 shows the performance of this distributed data structure on a cluster of 32 nodes with 2GB of RAM and 24 GB of disk, connected by gigabit ethernet. A benchmark program creates a matrix on a set of nodes, fills it with data, and then times 1000 row reads, followed by row writes, column reads and writes, and single cell reads and writes. Write operations are flushed to disk before the clock is stopped. The results show the data throughput for all access modes for matrices of double precision floating point values ranging from 1K square (8MB of data total) to 64K

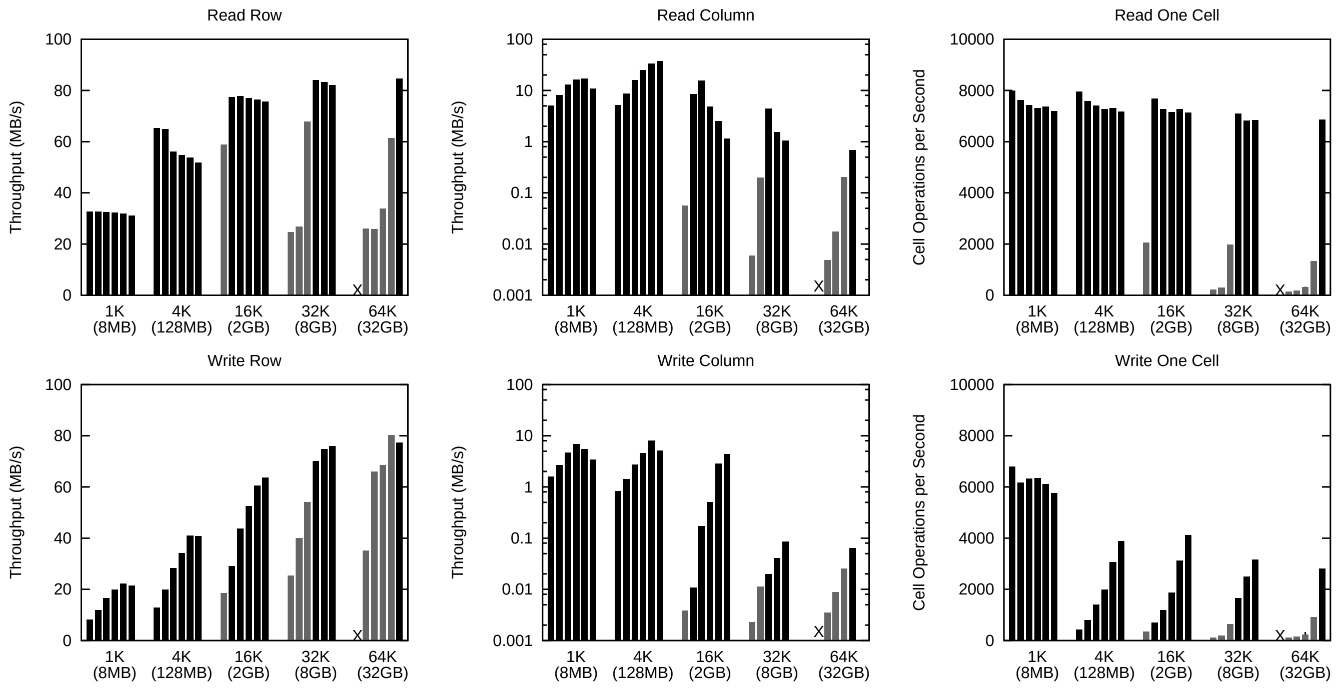


Fig. 6. Scalability and Performance of a Distributed Results Matrix.
These graphs show the throughput a single client can achieve while accessing a distributed matrix in varying configurations on a cluster of 32 nodes each with 2GB of RAM and 24GB of disk. A benchmark program creates a matrix of a given size and then measures the time to read and write rows, columns, and single cells at random. Each cluster of bars shows the performance where the matrix is partitioned across 1, 2, 4, 8, 16, and 32 nodes. Bars in gray indicate configurations where the matrix is sufficiently large that it does not fit in main memory. For example, the marked example shows the performance of reading rows from a 64K by 64K matrix of double-precision values (a total of 32GB of data) configured on 1-32 hosts. By distributing the matrix across a large number of hosts, the data structure can be kept in memory rather than disk, improving performance by several orders of magnitude.

square (32GB of data total). Each cluster of bars shows how the performance varies on 1, 2, 4, 8, 16, and 32 nodes. The bars shown in gray indicate cases where the matrix does not fit in aggregate memory. (Note that the case of a 64Kx64K array on a single node is marked with an 'X', because it cannot fit on a single disk in this system.)

If we consider row reads, we observe that harnessing multiple hosts improves performance dramatically when it allows the matrix to reside entirely in memory. Increasing the number of hosts beyond the memory threshold results in no benefit or a slight decrease because each read must block until the requested data is available. Row writes see some benefit from increasing beyond the memory limit because multiple writes can be pipelined into each local buffer cache, so additional hosts add I/O bandwidth. Column and single cell reads and writes naturally see much worse performance than row operations, but the pattern is the same: multiple hosts improve read performance up to the memory threshold, and improve write performance beyond.

By requiring the use of distributed data structure, several goals are accomplished at once: the user is forced to employ an efficient storage representation, the results may scale to sizes much larger than possible on any one disk, and performance is improved to the point where real-time data analysis becomes feasible.

V. Evaluation

Environment. To evaluate the concept of using abstractions for data intensive computing on a campus grid, we have constructed an All-Pairs engine and employed it with biometric, data mining, and bioinformatics applications at the University of Notre Dame. Our campus grid consists of about 500 CPUs connected to 300 disks. We use Condor [40] to manage the CPUs and the Chirp [39] user level filesystem 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 gigabit Ethernet that serves many more machines for general Internet access. Figure 7 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, both CPU and disk performance vary widely, with no simple correlation.

Configurations. We evaluate two implementations of All-Pairs. The first is labelled *abstraction*, which is exactly the implementation described above. In this mode, the implementation takes advantage of its knowledge of the I/O structure of the workload to model the system, distribute the data, and collect the outputs. As a tradeoff, this mode can only take advantage of nodes where the entire input data set will fit on disk. As the data sizes become larger, fewer CPUs become available.

We compare this to *conventional* mode, which is the traditional cluster architecture with a central filesystem. In this mode, the abstraction simply partitions the workload into the same sized jobs as above, but then just submits them all to the batch system, where they access data on demand from a central shared fileserver. For this configuration, the central file server was a dedicated 2.4GHz dual-core Opteron machine with 2GB of RAM, also running a Chirp fileserver.

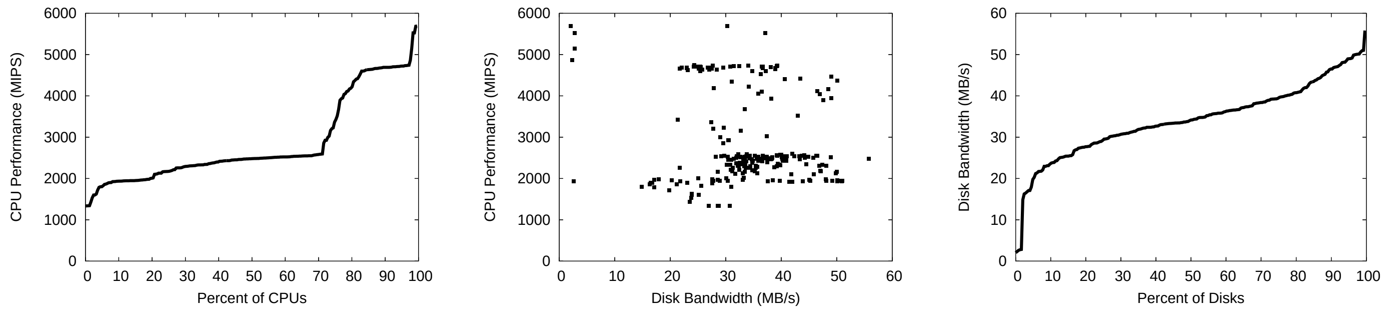


Fig. 7. Performance Variance in a Campus Grid

A campus grid has a high degree of heterogeneity in available resources. (a) shows the distribution of CPU speed, ranging from 1334 to 5692 MIPS. (c) shows the distribution of disk bandwidth, as measured by a large sequential write, ranging from 2 MB/s (misconfigured controller) to 55 MB/s. (b) shows the weak relationship between CPU speed and disk bandwidth. A fast CPU is not necessarily the best choice for an I/O bound job.

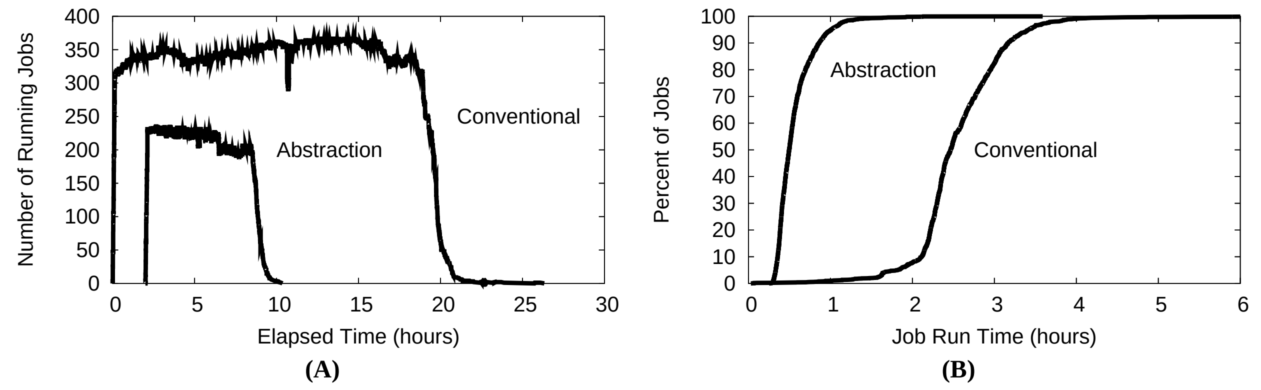


Fig. 8. Challenges in Evaluating Grid Workloads

Evaluating workloads in a campus grid is troublesome, because of the variance of available resources over time. 8(A) shows the CPUs assigned to two variants of the same 2500x2500 All-Pairs problem run in conventional and abstraction modes. 8(B) shows the distribution of job run times for each workload. In this case, the abstraction mode 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 resource efficiency *metric described in the text.*

Note that we cannot compare against a kernel-level distributed filesystem like NFS [37]. This campus grid spans multiple administrative domains and firewalls; gaining access to modify kernel level configurations is impossible in this kind of environment. Both configurations use the exact same software stack between the end user's application and the disk, differing only in the physical placement of jobs and data. In any case, the precise filesystem hardware and software is irrelevant, because the conventional configuration saturates the gigabit network link.

Metrics. Evaluating the performance of a large workload running in a campus grid has several challenges. In addition to the heterogeneity of resources, there is also significant time variance in the system. The number of CPUs actually plugged in and running changes over time, and the allocation of those CPUs to batch users changes according to local priorities. In addition, our two different modes (*abstraction* and *conventional*) will harness different numbers of nodes for the same problem. How do we quantitatively evaluate an algorithm in this environment?

Figure 8(A) shows this problem. Here we compare an All-Pairs run of 2500x2500 on a biometric workload. The *conventional* mode uses all available CPUs, while the *abstraction* mode chooses a smaller number. Both vary considerably over time, but it is clear that *abstraction* completes much faster using a smaller number of resources. Figure 8(B) shows the distribution of job run times, demonstrating that the average job run time in *abstraction* is much faster, but the long tail rivals that of *conventional.*

To accommodate this, we present two quantitative results. The *turnaround time* is simply the wall clock time from the invocation to completion. The *resource efficiency* is the total number of cells in the result (the number of function invocations), divided by the cumulative CPU-time (the area under the curve in Figure 8(A). For both metrics, smaller numbers are better.

Results. Figure 9 shows a comparison between the two implementations for a biometric comparing face images of 1.25MB each in about 1s each. For workload above 1000x1000, the abstraction is twice as fast, and four times more efficient. Figure 10 shows a data mining application comparing datasets of 700KB in about 0.25s each. Again, the execution time is almost twice as fast on large problems, and seven times more resource efficient on the largest configuration. We also constructed a synthetic application with a heavier I/O ratio: items of 12.5MB with 1s of computation per comparison. 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 11 shows for this third workload another

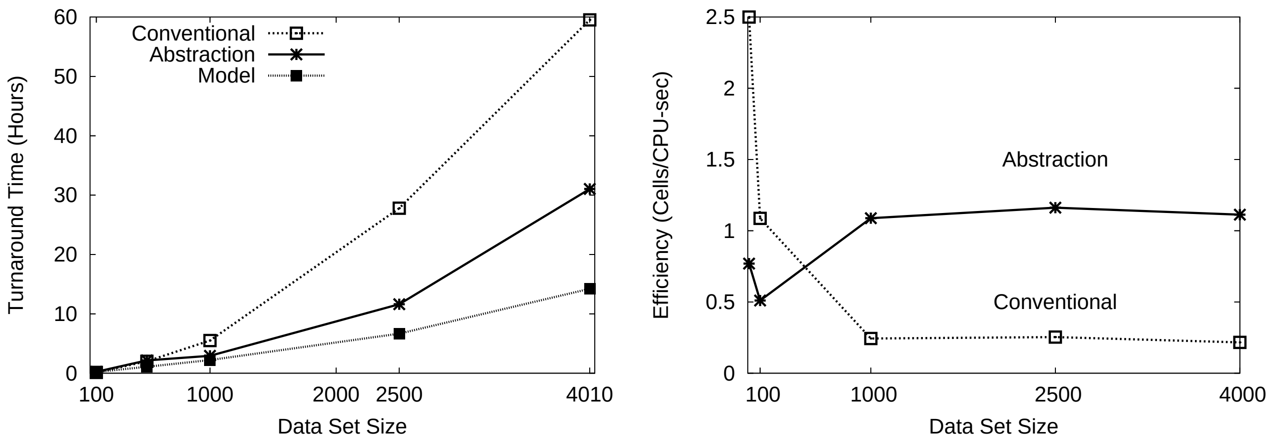


Fig. 9. Performance of a Biometric All-Pairs Workload
The biometric face comparison function takes 1s to compare two 1.25MB images.

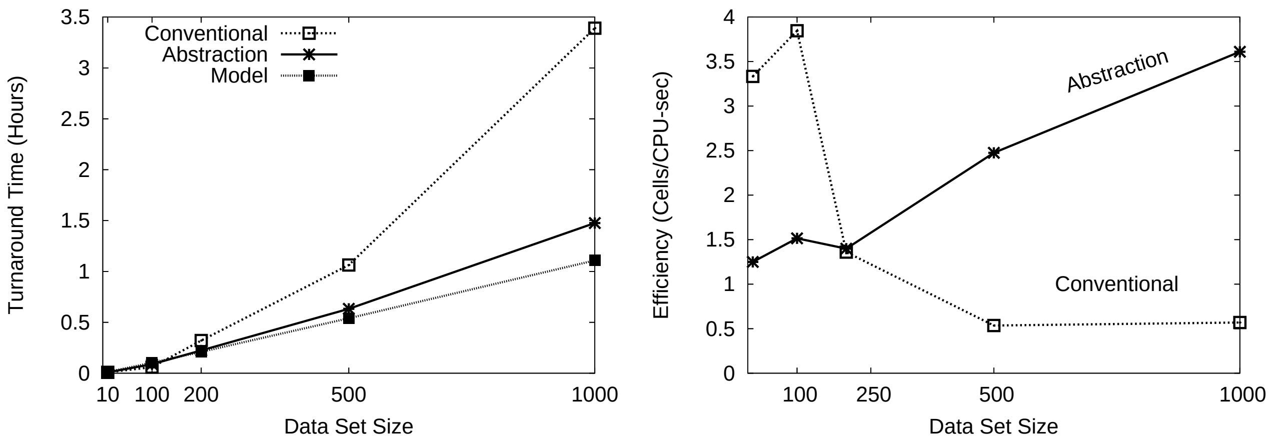


Fig. 10. Performance of a Data Mining All-Pairs Workload
The data mining function takes .25 seconds to compare two 700KB items.

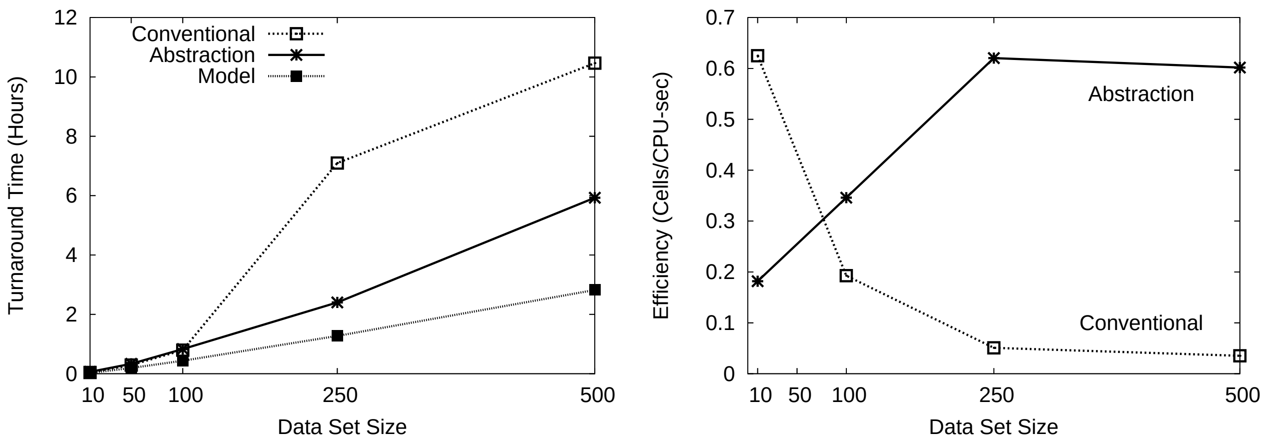


Fig. 11. Performance of a Synthetic All-Pairs Workload
This function takes 1s for two 12.5MB data items, 10 times the data rate of the biometric workload.

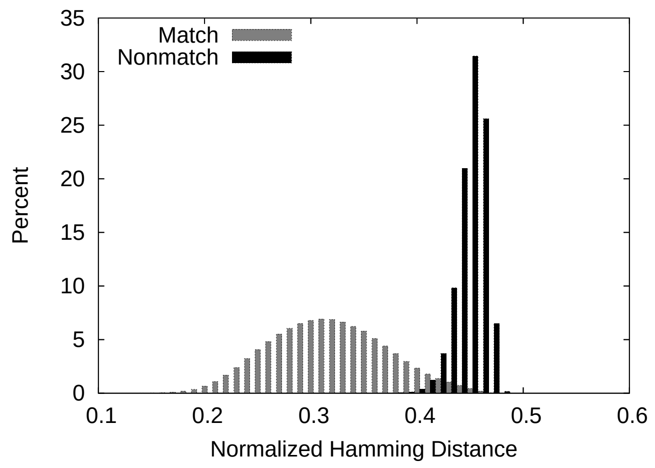


Fig. 12. Results From Production Run
The results of All-Pairs on 58,639 iris codes. The gray indicates comparison of irises from the same person (match). The black indicates comparison of irises from different people (nonmatch).

example of the abstraction performing better than the conventional mode on all non-trivial data sizes.

For comparison, we also show the execution time predicted by the model for the abstraction. As expected, the actual implementation is often much slower than the modeled time, because it does not take into account failures, preemptions, competition for resources, and the heterogeneity of the system.

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 sufficiently to match the data staging step in the application.

For larger problem sizes, however, the conventional algorithm is not as efficient because the aggregate I/O rate (hs/t) exceeds the capacity of the network link to the central file server, which has a theoretical maximum of 125 MB/s. If we assume that exactly 300 CPUs are in use at once, the aggregate I/O rate is 375 MB/s in Figure 9, 820 MB/s in Figure 10, and 3750 GB/s in Figure 11. To support these data rates in a single file server would require a massively parallel storage array connected to the cluster by a high speed interconnect such as Infiniband. Such changes would dramatically increase the acquisition cost of the system. The use of an abstraction allows us to exploit the aggregate I/O capacity of local storage, thereby achieving the same performance at a much lower cost.

VI. Production Workload

Our implementation of All-Pairs has been used in a production mode for about one year to run a variety of workloads in biometrics, which is the study of identifying humans from measurements such as fingerprints, face images, and iris scans. Our tool has been used to explore matching algorithms for 2-D face images, 3-D face meshes, and iris images. Our largest production run so far explored the problem of matching a large body of iris images. To motivate the problem, we will briefly describe how iris biometrics are performed. A more detailed overview is given by Daugman [15].

Failure Type	Observer	Count
Job killed with signal 15.	engine	4161
Job killed with signal 9.	engine	372
Inputs not accessible.	wrapper	5344
Failed to store output.	wrapper	17
Dynamic linking failed.	wrapper	45
Function returned 255.	wrapper	20
Function returned 127.	wrapper	300
Job preempted.	batch system	14560

TABLE II
Summary of Failures in Production Run

A conventional iris biometric system will take a grayscale iris image and extract a small binary representation of the texture in the iris, called an *iris code* [9]. The iris code is a small (20KB) black and white bitmap designed to make comparisons as fast as possible. To compare two iris codes, we compute the normalized Hamming distance, which is simply the fraction of the bits that differ. Two random binary strings would likely differ in about half of their bits, and would therefore have a Hamming distance score around 0.5. Two iris codes corresponding to two different images of the same person's eye would not differ in as many bits, and thus have a Hamming distance closer to 0. A comparison between two different images of the same iris is called a *match*, and comparison between images of two different eyes is called a *nonmatch*. Ideally, all match comparisons should yield lower Hamming distance scores than all nonmatch comparisons.

Our largest run computed Hamming distances between all pairs of 58,639 20KB iris codes from the ICE 2006 [33] data set, which is to be released to the public shortly. The next largest publically available iris data set is CASIA 3 [10], about three times smaller, on which no results have been published on complete comparisons. To our knowledge, this will be the largest such result ever computed on a publically available dataset.

Figure 12 show the end result of this workload. A histogram shows the frequency of Hamming distances for matching irises (different images from the same person) and non-matching irises (images from different people.) As can be seen, the bulk of each curve is distinct, so an online matching system might use a threshold of about 0.4 to determine whether two irises represent different people. However, these results also indicate a group of non-matching comparisons that significantly overlap the matches. In examining these comparisons, they discovered that these low scores occur when one of the images in the comparison is partially occluded by eyelids so that only a small amount of iris is visible and available for the comparison. This observation will drive future work that must identify and account for occluded irises specifically. Without the ability to easily perform large scale comparisons, such an observation could not have been made.

Our fastest single machine can perform 50 comparisons per second, and would take about 800 days to run the entire workload sequentially. Our All-Pairs implementation ran in 10 days on a varying set of 100-200 machines, for a parallel speedup of about 80. The speedup is imperfect because one cannot maintain ideal conditions over the course of ten days. Table II summarizes all of the failures that occurred over that period, grouped by the component that observed and responded to the failure.

As discussed above, the Condor batch system handles a large fraction of the failures, which are preemptions that force the job to run elsewhere. However, the number of failures handled by the rest of the system is still large enough that they cannot be ignored. All are cases that are not supposed to happen in a well regulated system, but creep in anyhow. Despite extensive debugging and development, the user's function still crashes when exposed to unexpected conditions on slightly different machines. A number of machines were wiped and re-installed during the run, so input files were not always found where expected. There are a surprisingly large number of instances where the job was forcibly killed with a signal; only a local system administrator would have permission to do so. These data emphasize the point that anything can and will happen in a campus grid, so every layer of system is responsible for checking errors and ensuring fault tolerance – this task cannot be delegated to any one component.

Despite these challenges, we have demonstrated that the All-Pairs abstraction takes a computation that was previously infeasible to run, and makes it easy to execute in a matter of days, even in a uncooperative environment. Using this abstraction, a new graduate student can break new ground in biometrics research without becoming an expert in parallel computing.

VII. 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 [29], and Seti@Home [38] 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 [5] and attempt other performance enhancements. [13]. Bulk-Synchronous-Parallel (BSP) [12] 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. 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 [16] 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.

Hadoop [24] is a widely-used open source implementation of Map-Reduce. Although Hadoop has significant fault-tolerance capabilities, it assumes that it is the primary controller of a dedicated cluster, so it does not have the necessary policy and preemption mechanisms necessary for a campus grid.

That said, we may ask whether we can express an All-Pairs problem using the Map-Reduce abstraction. We can do so, but an efficient mapping is neither trivial nor obvious. A pure Map can only draw input from one partitioned data set, so we might itemize the Cartesian product into a set like $S = ((A_1, B_1), (A_1, B_2)...)$ then invoke $Map(F, S)$ Obviously, this would turn a dataset of n elements into one of n^2 elements, which would not be a good use of space. If set A is smaller, we might package A with F and define $F^+ = Map(F, A)$ and then compute $Map(F^+, B)$, relying on the system to partition B. However, this would result in the sequential distribution of one set to every node, which would be highly inefficient as shown above. A more efficient method might be to add our mechanism for data distribution alongside Hadoop, and then use the Map-Reduce to simply invoke partitions of the data by name.

As this discussion shows, there are many ways to express one problem in terms of the other. In order to guide users to an appropriate implementation, it is best to provide them with an abstraction that closely matches the problem domain.

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 multi-dimensional arrays [7], storing hashtables [23] and B-trees [31] and semi-relational data [11], searching images [27], and sorting record-oriented data [4].

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 [17] 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 [40] 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 [22] 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 [42] and GridDB [30] build on this idea by providing languages for describing the relationships between complex data sets.

VIII. Conclusion

We have shown how an abstraction like All-Pairs can be used to improve the usability, performance, and efficiency of a campus grid. By expressing the high level structure of a workload, an abstraction enables the runtime to choose an appropriate partitioning strategy, exploit data re-use, and recovery cleanly from failures. An abstraction also serves to guide the user toward an appropriate mechanism for the task at hand, such as a distributed array for storing results. Patterson [34] has suggested that abstractions are the assembly language that will be used to program large distributed systems. All-Pairs is one new instruction to add to that set, and we suspect that there are others. Future work may identify common patterns in large workloads, and create other abstractions that enable new forms of discovery.

Acknowledgments

This work was supported in part by National Science Foundation grants CCF-06-21434, CNS-06-43229, and CNS-01-30839. We thank David Cieslak, Tim Faltemier, Tanya Peters, and Robert McKeon for testing early versions of this work.

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. Andoni and P. Andyk. Near optimal hashing algorithms for approximate nearest neighbor in high dimensions. *CACM*, 51(1).

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

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

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

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

[8] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *ACM SIGPLAN Notices*, volume 30, August 1995.

[9] K. Bowyer, K. Hollingsworth, and P. Flynn. Image understanding for iris biometrics: A survey. *Computer Vision and Image Understanding*, 110(2):281–307, 2007.

[10] Center for Biometrics and Security Research. CASIA iris image database http://www.cbsr.ia.ac.cn/english/Databases.asp, accessed Apr 2008.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, , and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Operating Systems Design and Implementation*, 2006.

[12] 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.

[13] 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.

[14] L. Dagum and R. Menon. OpenMP: An industry standard api for shared memory programming. *IEEE Computational Science and Engineering*, 1998.

[15] J. Daugman. How iris recognition works. *IEEE Transactions on Circuits and Systems for Video Technology*, 14(1):21–30, 2004.

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

[17] 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 scientific workflows onto distributed systems. *Scientifi c Programming Journal*, 13(3), 2005.

[18] J. J. Dongarra and D. W. Walker. MPI: A standard message passing interface. *Supercomputer*, pages 56–68, January 1996.

[19] J. Douceur and W. Bolovsky. A large scale study of file-system contents. In *Measurement and Modeling of Computer Systems (SIGMETRICS)*, Atlanta, Georgia, May 1999.

[20] T. Elsayed, J. Lin, and D. Oard. Pairwise document similarity in large collections with mapreduce. In *48th Annual Meeting of the Association for Computational Linguistics*, 2008.

[21] I. Foster and C. Kesselman. Globus: A metacomputing intrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[22] 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.

[23] 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.

[24] Hadoop. http://hadoop.apache.org/, 2007.

[25] P. Havlak, R. Chen, K. J. Durbin, A. Egan, Y. Ren, X.-Z. Song, G. M. Weinstock, and R. Gibbs. The Atlas genome assembly system. *Genom Research*, 14:721–732, 2004.

[26] X. Huang, J. Wang, S. ALuru, S.-P. Yang, and L. Hillier. PCAP: a whole-genome assembly program. *Genome Research*, 13:2164–2170, 2003.

[27] 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.

[28] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data parallel programs from sequential building blocks. In *Proceedings of EuroSys*, March 2007.

[29] 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.

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

[31] 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.

[32] C. Moretti, J. Bulosan, P. Flynn, and D. Thain. All-pairs: An abstraction for data intensive cloud computing. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2008.

[33] National Insitute of Standards and Technology. Iris challenge evaluation data http://iris.nist.gov/ice/, accessed Apr 2008.

[34] D. Patterson. The data center is the computer. *Communications of the ACM*, 51, January 2008.

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

[36] R. Pordes and et al. The open science grid. *Journal of Physics: Conference Series*, 78, 2007.

[37] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *USENIX Summer Technical Conference*, pages 119–130, 1985.

[38] 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.

[39] D. Thain, C. Moretti, and J. Hemmes. Chirp: A practical global file system for cluster and grid computing. *Journal of Grid Computing*, to appear in 2008.

[40] 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.

[41] D. Thain, T. Tannenbaum, and M. Livny. How to measure a large open source distributed system. *Concurrency and Computation: Practice and Experience*, 18:1989–2019, 2006.

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

Christopher Moretti graduated from the College of William and Mary in 2004 with a B.S. in Computer Science. He received a Master of Science in Computer Science and Engineering from the University of Notre Dame in 2007. He is currently a Ph.D. candidate at Notre Dame, researching distributed computing abstractions for scientific computing.

Hoang Bui received the B.S. and M.S. in Computer Science in 2004 and 2007 from Midwestern State University. He is currently a PhD student of Computer Science and Engineering at the University of Notre Dame, where his research focuses on scientific repositories and workflows utilizing distributed computing systems.

Karen Hollingsworth is a graduate student studying iris biometrics at the University of Notre Dame. She graduated as valedictorian of the College of Science from Utah State University in 2004, with B.S. degrees in computational math and math education. She has taught algebra and trigonometry at both high school and college levels. She completed the M.S. degree in computer science and engineering from the University of Notre Dame in May 2008.

Brandon Rich received the B.S. in Computer Science in 2004 from Baylor University. He is currently an information technology professional at the University of Notre Dame.

Patrick Flynn is Professor of Computer Science and Engineering and Concurrent Professor of Electrical Engineering at the University of Notre Dame. He received the B.S. in Electrical Engineering (1985), the M.S. in Computer Science (1986), and the Ph.D. in Computer Science (1990) from Michigan State University, East Lansing. He has held faculty positions at Notre Dame (1990-1991, 2001-present), Washington State University (1991-1998), and Ohio State University (1998-2001). His research interests include computer vision, biometrics, and image processing. Dr. Flynn is a Senior Member of IEEE, a Fellow of IAPR, and an Associate Editor of IEEE Trans. on Information Forensics and Security.

Douglas Thain received the B.S. in Physics in 1997 from the University of Minnesota and the M.S. and Ph.D. in Computer Sciences in 1999 and 2004 from the University of Wisconsin. He is currently an Assistant Professor of Computer Science and Engineering at the University of Notre Dame, where his research focuses on scientific applications of distributed computing systems.