Abstractions for Cloud Computing with Condor

Draft Manuscript

7 August 2009

Douglas Thain and Christopher Moretti
Department of Computer Science and Engineering
384 Fitzpatrick Hall
University of Notre Dame
Notre Dame, IN, 46556
dthain@nd.edu
cmoretti@nd.edu

to appear in:
Syed Ahson and Mohammad Ilyas, editors,
Cloud Computing and Software Services, CRC Press / Taylor and Francis Books, 2009.

1.1 Introduction

A **cloud computer** provides a simple interface that allows end users to allocate large amounts of computing power and storage space at the touch of a button. However, many potential users of cloud computers have needs much more complex than simply the ability to allocate resources. In scientific domains, it is easy to find examples of workloads that consist of hundreds or thousands of interacting processes. A user that wishes to run such a workload on a cloud computer faces the daunting task of deciding how many resources to allocate, where to dispatch each process, when and where to move data, and how to deal with the inevitable failures. For this reason, many users with large workloads are reluctant to move away from the predicable environment of a single workstation or multicore server.

Abstractions are an effective way of harnessing large cloud computers while insulating the user from technical complexities. An **abstraction** is a structure that allows one to specify a workload in a way that is natural to the end user. It is then up to the system to determine how best to realize the workload given the available resources. This also allows the user to move a workload from one machine to another without rewriting it from scratch. The concept of abstraction is fundamental to computer science, and examples can be found in other software systems such as compilers, databases, and filesystems.

Map-Reduce [9] is a well known abstraction for cloud computing. The Map-Reduce abstraction allows the user to specify two functions that transform and summarize data, respectively. If the desired computation can be expressed in this form, then the computation can be scaled up to thousands of nodes. The Map-Reduce abstraction is well suited for analyzing and summarizing large amounts of data, and has a number of implementations of which the open source **Hadoop** [6] is the most widely deployed.

But are there other useful abstractions? In our work with several scientific application communities at the University of Notre Dame, we have encountered a number of large workloads that are regularly structured, but do not fit the Map-Reduce paradigm. In each case, we found workloads that were easy to write on the chalkboard, possible to run on one machine, but very challenging to scale up to hundreds or thousands of nodes. In each case, our research group worked to design an

abstraction that could represent a large class of applications, and was able to execute reliably on a cloud computer.

In this chapter, we will describe the following set of abstractions, in roughly increasing order of complexity:

- **Map** - Applies a single program to a large set of data.
- **All-Pairs** - Computes a Cartesian product on two large sets of data.
- **Sparse-Pairs** - Computes a function on selected pairs of two large sets of data.
- **Wavefront** - Carries out a large dynamic programming problem.
- **Directed Graph** - Runs a large graph of processes with multiple dependencies.

We have implemented these abstractions on the Condor distributed processing system. We will begin with a short overview of Condor as a cloud computer, and then explain each abstraction in turn. For each, we will present a formal model, describe how the abstraction is implemented, and give an example of a community that has used the abstraction to scale up an application to hundreds of CPUs. We conclude the chapter by discussing the relative power of each abstraction.

1.2 Condor as a Cloud Computer

Our foundation for this work is the **Condor** distributed processing system. Condor was first created in 1987 at the University of Wisconsin, and has remained in continuous development and deployment ever since [15, 27]. At the time of writing, it was deployed at several thousand institutions around the world, managing several hundreds thousand CPU cores [25, 5]. At a typical university, the Condor software is deployed to all available machines, including desktop workstations, classroom machines, and server clusters, all of which are typically idle 90% of the day. Users queue jobs to run in the system, and Condor matches the jobs to run on machines that would otherwise go unused.

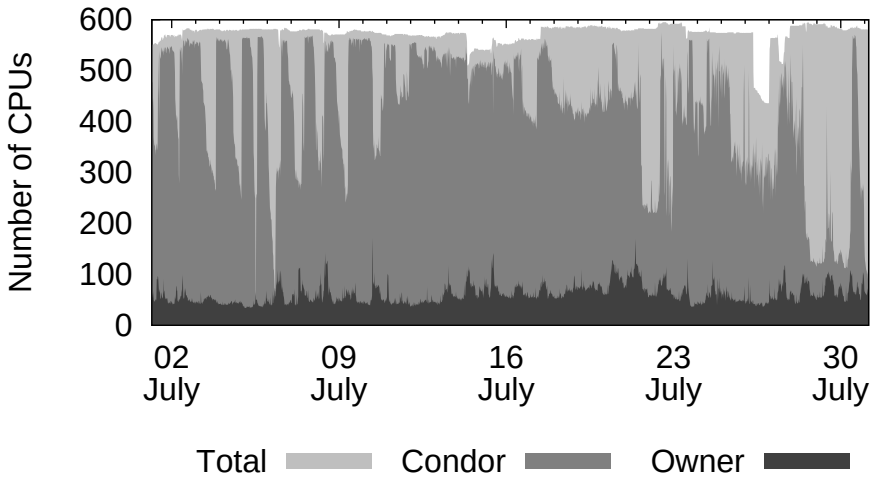

Figure 1.1: Time Variations in a Condor Pool

A large Condor pool can be considered a **cloud computer**. Like other cloud computing systems, users request service from Condor, but don't care (and cannot control) exactly which resources are used to service that request. A job submitted to Condor could run on a desktop machine down the hall, or in a machine room at another institution. However, Condor is unlike other cloud computing systems in that it employs **preemption** [22]. A job running on a machine may be preempted if the machine's owner returns to type on the keyboard or otherwise uses the CPU.

Figure 1.1 shows the natural variations found in our campus Condor pool over the course of July 2009. The dark "Owner" curve shows the number of CPUs currently in use by their owners, who are either typing at the keyboard or making extensive use of the CPU. The lighter "Condor" curve shows the number of CPUs currently harnessed by Condor. The lightest "Total" curve shows the total number of CPUs in the pool, which varies between 500 and 600. As can be seen, all of these values fluctuate considerably as machines are powered on and off, used during the work day, and harvested for batch workloads.

Condor has been widely used to run large numbers of long-running computations, typically scientific simulations. However, it is not as well suited for large numbers of tasks that are short running, data intensive, or both. Even in an unloaded system, it takes about thirty seconds from the time a job is submitted until it actually begins running on a machine. This is because Condor must mediate the needs of many different stakeholders, including the machine owner, the job owner, and

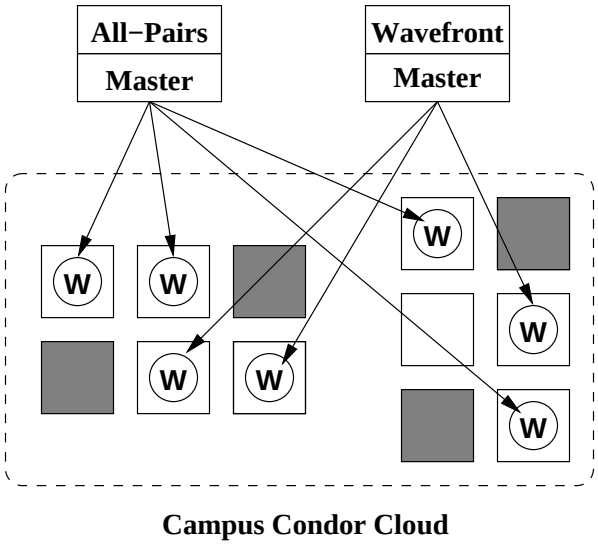

Figure 1.2: Multiple Abstractions Sharing a Condor Pool

the pool manager. (Other cloud computing systems have similar latencies for resource allocation.) Because Condor is careful to clean up thoroughly after a job completes, there is no easy way to maintain state on a machine across multiple jobs.

To compensate for these properties, we have built an intermediate layer of software called **Work Queue** that provides fast execution and data persistence on top of Condor. Work Queue consists of two pieces: a **Master** and a **Worker**. A Worker is a simple process that is submitted to the Condor pool like an ordinary batch job. Once running, it makes a network connection back to a Master process. The Master can send files to the worker, execute programs, and retrieve outputs.

In this way, the Master can start a new program in milliseconds rather than thirty seconds. Further, it can take advantage of a semi-persistent filesystem: if two consecutive tasks require the same input data, it only needs to be sent to the Worker once. Of course, if Condor decides to evict the Worker process, it will kill any running processes and delete the local storage. The Master is able to detect these evictions, and re-assign tasks to other Workers as needed.

Figure 1.2 shows how all of these pieces fit together. The end user is not exposed to any details of the cloud. Instead, he or she runs a command such as `All-Pairs` or `Wavefront` corresponding to the desired abstraction. The abstraction examines the workload, decides how many Workers it can use, and submits them as jobs to Condor. Condor decides what resources to

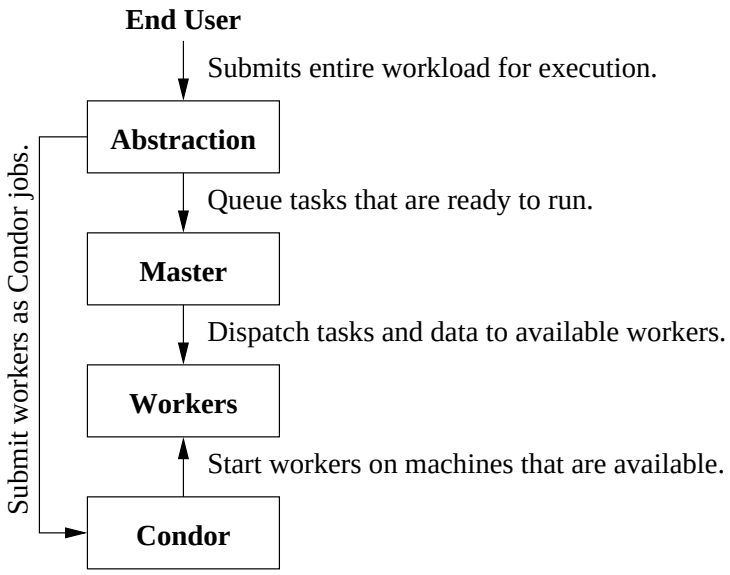

Figure 1.3: A Layered System for Cloud Computing

allocate to each user, and each abstraction schedules tasks on whatever Workers become available. The result is a layered system, where each component has a distinct responsibility, as shown in Figure 1.3.

1.3 The Map Abstraction

We will begin by describing the simplest abstraction – Map – and then work our way up to more complex abstractions. For each, we will give a formal definition, describe an example, and then explain a significant result achieved using the abstraction.

Map(**data** $D[i]$, **function** $F($ **data** $x))$
returns array R **such that** $R[i] = F(D[i])$

Map applies a function F to all elements of a dataset D, yielding a result dataset R. Of course, Map and similar operations have been available in functional programming languages such as LISP [24] for many years, and has long been recognized as a suitable primitive for parallel programming [7, 13]. Map is a natural starting point for exploring parallelism.

In practice, our users invoke a standalone program called `Map` that accepts two arguments: the

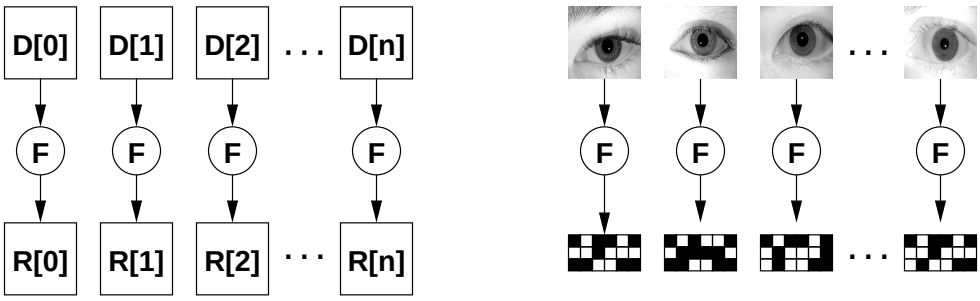

Figure 1.4: The Map Abstraction Applied to Biometrics

function is the name of a conventional program that transforms one file, and the array is a file listing the names of files to be mapped. In contrast to Map-Reduce [9], which interfaces with C++, and Hadoop [6], which interfaces with Java, Map and the rest of our abstractions use ordinary executable programs as "functions". This allows end users to use whatever language they are most comfortable with, and often are able to plug in existing tools without recoding.

Figure 1.4 shows an application of Map used extensively in biometrics. A common task is to convert a large set of iris images of about 300 KB each into iris codes of about 20 KB each. (An **iris code** is a compressed binary representation of an iris actually used for archival and comparison [8].) A program named `ConvertIrisToCode` can carry out one conversion in about 19s.

To execute this workload, the user runs:

```
Map IrisListing ConvertIrisToCode
```

Logically, this means to run `ConvertIrisToCode` once for each entry in `IrisListing`:

```
ConvertIrisToCode iris001.jpg iris001.code
ConvertIrisToCode iris005.jpg iris005.code
ConvertIrisToCode iris008.jpg iris008.code
...
```

Although one could accomplish a Map by simply submitting batch jobs, our implementation of the abstractions solves a number of technical challenges that would otherwise make using the

system very challenging. It caches the executable and other required libraries on the execution nodes, detects failed or evicted Workers, detects compatibility failures with various machines, aborts straggling Workers, preferentially assigns tasks to faster nodes, and deals with network outages and other failures. In this way, the user can focus on their desired work instead of the details of distributed computing.

A typical example of an unoptimized production run of Map on our cloud converted 58,639 iris images to codes in 2.4 hours, using anywhere between 100 and 400 Workers at any given time. The same workload would have taken 309 hours on a single CPU, for an effective speedup of 125X. By making use of the Map abstraction on the cloud, the end user can accomplish in a few hours what previously took over a week.

1.4 The All-Pairs Abstraction

Building on the idea of applying a function to a one-dimensional array of single inputs, we move on to All-Pairs, an abstraction in which each function call is applied onto a pair of inputs.

All-Pairs(**data** A, **data** B, **function** F(**data** x, **data** y))
returns matrix R **such that** $R[i,j] = F(A[i], B[j])$

The **All-Pairs** abstraction applies a function F to each pair of elements in datasets A and B, yielding a result matrix R, where each cell is the result of comparing two items. A common variant of All-Pairs is to let $A = B$, in which case it is often only necessary to compute half of the result matrix. Previous researchers have studied All-Pairs theoretically [28] and on small clusters [4]. Our contribution is to scale the problem up to hundreds of nodes in the cloud [16].

As with the previous abstraction, the user provides a "function" in the form of a program that compares two input files. The data sets A and B are text files listing the remaining files to be compared. For small problems, the result matrix is emitted as a plain text file; for large problems, it is stored as a distributed data structure.

All-Pairs problems occur in several fields, such as biometrics, bioinformatics, and data mining.

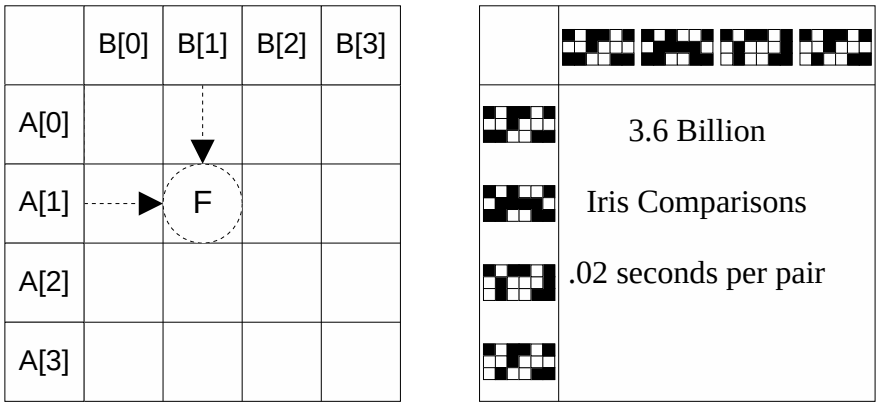

Figure 1.5: The All-Pairs Abstraction Applied to Biometrics

We will focus on biometrics here. A common problem in the field is evaluation of new algorithms developed to improve the state of the art in personal identification. One way to do this is to assemble a large corpus of images and compare all of them to each other using the new algorithm. Results obtained with different algorithms on the same set of images are directly comparable for overall effectiveness.

Figure 1.5 continues with the application from the previous example. Using Map, the user has already reduced 58,639 iris images into an equal number of compact iris codes. He has written a program `CalculateIrisSimilarity` which computes the masked Hamming distance between two iris codes. The program can complete approximately 50 such comparisons per second. An All-Pairs comparison of those images against each other would consist of 3.4 billion function executions, 795 days of of serial computation, and 6.8 TB of aggregate input requirements.

Such a workload is impractical to complete serially, so scaling up to the cloud is required. To invoke the All-Pairs abstraction, the user specifies the input sets and the comparison function:

```
AllPairs SetA SetB CalculateIrisSimilarity
```

The abstraction handles all of the computation and data management. Using a model that takes into account function computation time and data element sizes it calculates how many resources should be used for the workload and how much work they should be given at a time to balance queuing overhead and job runtime. It then distributes data to chosen resources and assigns computation to those resources. If the node has multiple cores, the access pattern is carefully chosen to

maximize the cache hit rate. The final results are stored in a large distributed array, which may be accessed directly, or downloaded to a local file.

Developing a model for the All-Pairs problem is a critical component for several reasons. First, it relieves the user of the responsibility of determining the number of resources. As problems scale up in size, the number of resources required to not necessarily scale up in kind, and thus users may make poor decisions – underprovisioning the system hurting performance, or overprovisioning the system increasing overhead and wasting resources. Second, the ability to predict very general approximate runtimes based on simple diagnostic benchmarks for work allows the system to manage running processes and detect jobs that are not making progress within a reasonable time (whether due to bugs, hardware misconfigurations, etc.) automatically instead of requiring a user to aggressively monitor his job.

Our largest production run of All-Pairs compared 58,639 iris codes generated from the Iris Challenge Evaluation 2006 [2] dataset all to each other. To our knowledge, this is the largest such result ever computed on a publicly available dataset. The abstraction ran in 10 days on a varying set of 100-200 nodes in the cloud, for an effective speedup of about 80X [16].

1.5 The Sparse-Pairs Abstraction

There are many workloads that involve the comparison of large sets of objects, but do not require *all possible* comparisons. These workloads require the Sparse-Pairs abstraction.

Sparse-Pairs(**data** A, **data** B, **function** F(**data** x, **data** y), **pairs** P)
returns array R **such that** $F(A[P[i].x], B[P[i].y])$

The **Sparse-Pairs** abstraction applies a function F to pairs of elements in sets A and B given by the set P, yielding a result set R. Sparse-Pairs fits between the one-dimensional array abstraction of Map, and the two-dimensional array abstraction of All-Pairs. In this way it is a bit like superimposing Bag-of-Tasks [3, 23] on top of the one-dimensional structure of Map.

Sparse-Pairs problems occur frequently in the field of bioinformatics, particularly in the prob-

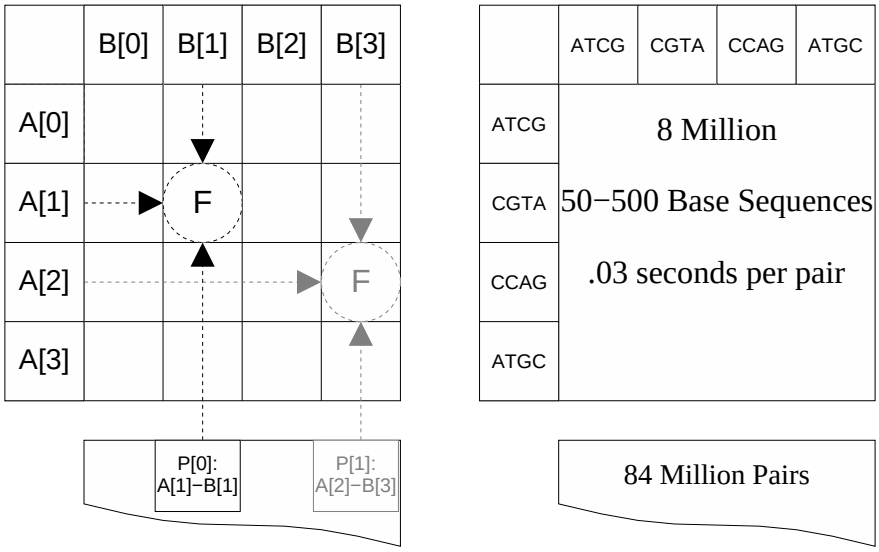

Figure 1.6: The Sparse-Pairs Abstraction Applied to Bioinformatics

lem of **genome assembly**. Very briefly, genome assembly is the problem of assembling many small fragments of DNA (hundreds of bytes each) into one long string (billions of bytes) that represents the entire genomic code of an organism. This is much like putting together a jigsaw puzzle: one must compare many pieces to each other in order to determine which should be adjacent.

In principle, one could run an All-Pairs abstraction to compare all fragments to each other, and then match up the pieces with the best scores. However, for a sufficiently large problem, this is computationally infeasible. Fortunately, there exist various heuristics that can be used to filter out the vast majority of these comparisons [19], leaving only a list of "candidate" sequences to compare. This candidate list becomes the P set for a Sparse-Pairs workload, as shown in Figure 1.6

The principal complication for Sparse-Pairs is that it is not generally feasible to optimize a bulk transfer of data files to many nodes because while each data item is used multiple times, the number of repetitive uses may be far less than the number of nodes. Thus, the Master must be active in transferring data, which potentially creates a single bottleneck at the Master's outgoing network link. Additionally, for fast-finishing functions, even if the Master has sufficient bandwidth the network latency may be too great to keep a sufficient number of Workers satiated.

The first issue can be alleviated with compressed data – in bioinformatics, the language $\{ACGT\}$ can easily be compressed to two bits per basepair – or multiple Masters. The second can be im-

proved by grouping together many functions into a single “task” sent to the Worker in order to prevent numerous high-latency round-trips in sending data for potentially thousands of functions.

Two data-related factors differentiate Sparse-Pairs from both Map and All-Pairs. First, although the pairs are sparse, each sequence is still used many times throughout the workload. Thus, while the pairs to be computed could be written in full to files in which every pair was a single element, and Map could then be run using that input, this is inefficient. Instead, if the set of sequences is not too large for main memory, the sequences can be stored only once in their datafile and are read into the Master’s memory to construct the tasks for the Workers on the fly as the workload advances.

A Sparse-Pairs result is a subset of a corresponding All-Pairs result. All-Pairs can be optimized to take advantage (via data transfer and assignment of computation to resources) of the fact that every single computation pair will be completed. However, it is unnecessary to complete an entire All-Pairs problem for every case of Sparse-Pairs, and for particularly sparse sets of pairs, it may be very inefficient to do so even if the All-Pairs abstraction is highly optimized. The regular structure of All-Pairs also allows the interface to the abstraction to require only the function and the names of the full sets. For Sparse-Pairs the usage is less uniform even for the same input set size, thus the design in which each pair should be transfered to the host on which its computation will be completed, rather than allowing computation to be arbitrarily assigned to pre-staged identical hosts.

Our Sparse-Pairs implementation is in regular use with a bioinformatics research group at Notre Dame. Our largest assembly so far used 8 million sequences extracted from a completed *Sorghum bicolor* genome and completed alignments for 84 million candidate pairs. (The equivalent All-Pairs would have required 64 *trillion* comparisons.) Using 512 CPUs, the assembly completed in just under two hours, with an effective speedup of 425X [17].

1.6 The Wavefront Abstraction

So far, each of the abstractions discussed has allowed computation to be completed in an arbitrary order. However, more complex abstractions such as Wavefront have dependencies, requiring one stage of the computation to complete before another can proceed.

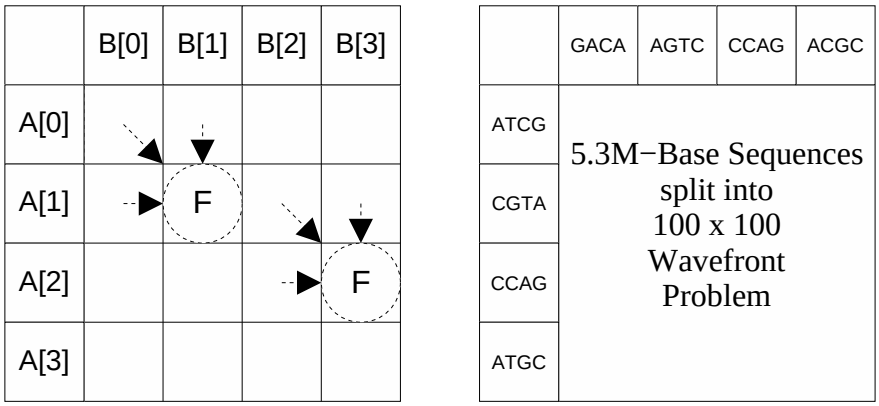

Figure 1.7: The Wavefront Problem Applied to Bioinformatics

Wavefront(**data** X, **data** Y, **function** F(**data** x, **data** y, **data** d))

returns matrix $R[i,j] = \begin{cases} X[i] & \textbf{if } j = 0 \\ Y[j] & \textbf{if } i = 0 \\ F(R[i-1,j], R[i,j-1], R[i-1,j-1]) & \textbf{otherwise} \end{cases}$

Figure 1.7 shows the **Wavefront** abstraction, which is a recurrence relation in two dimensions. Given two data sets as original input, and a function that takes three inputs and returns a single output, calculate the function at each of n^2 possible states of the system, where each state is defined by the results of its predecessor states. A state's predecessors are its neighbors in a matrix, whose values have been computed by previous function executions. The problem can be generalized to multiple dimensions. Wavefront has previously been studied in the context of multicore processors [1], which our work has extended to clusters and clouds of multicore machines [29].

In practice, the user invokes Wavefront by specifying the input data sets and the recurrence function. As before, the "function" is an arbitrary program that accepts files on the command line:

```
Wavefront XData YData RecurrenceFunction
```

Examples of Wavefront problems occur in game theory, economics, bioinformatics, and any problem that involves dynamic programming. In game theory, a recurrence table can be constructed to enumerate all possible states of a simulation with given inputs. Each cell in the table is dependent on its previous neighbor states. With a completed table, economists can see the start

states, all possible final states, and all possible paths within the simulated context.

A common use of Wavefront in bioinformatics is the alignment of two very large DNA strings. This is done by constructing a dynamic programming table, where each cell gives the "score" of the alignment with each string offset by the coordinates of that cell. The alignment of two complete genomes (billions of bytes) is intractable serially. However, the entire problem can be broken up into a number of smaller sub-alignment problems. Each sub-problem computes the dynamic programming table for a fragment of the genome, and then passes the boundary value to its neighbor.

In previous abstractions, the ability to predict runtimes of work units was used primarily to provision resources. Determining which processes have run too long is useful for detecting misconfigured nodes, but a slow node at the beginning or middle of the workload does relatively little damage to overall performance because there is still a high degree of concurrency. In Wavefront, however, predicting runtimes takes on extra importance. A slow-finishing work unit in Wavefront propagates its delay through to all of its dependents. This is especially harmful early in a workload, when most or all of the remaining computations are dependents, and there is already limited concurrency available in the problem. To combat this, Wavefront makes use of the Work Queue's ability to remove, reschedule, and restart tasks that have run significantly beyond their predicted completion time.

Using the Wavefront abstraction, we were able to complete the alignment of two variants of the Anthrax genome measuring 5.4 million bytes. Each genome was split into 100 fragments of about 54,000 bytes, yielding a 100x100 Wavefront problem. Using the cloud, the problem completed in 8.3 hours instead of 13 days, achieving an effective speedup of 38X.

1.7 The Directed Graph Abstraction

The abstractions that we have presented so far have a highly regular structure. However, many users have applications that can only be described as a **directed graph** of programs. There exist a number of **workflow languages** that are capable of expressing arbitrary graphs of tasks, such as

```
a.dat b.dat: all.dat
   split all.dat

a.out: a.dat
   process a.dat

b.out: b.dat
   process b.dat
```

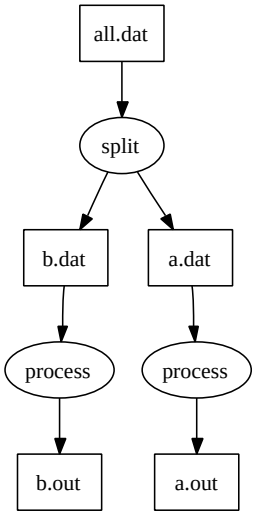

Figure 1.8: Small Example of the Makeflow Language

Dagman [26], Pegasus [10], Taverna [18], Swift [30], BPEL [14], and Dryad [12], to name a few. Each of these languages has its own syntax, and is capable of connecting to a number of remote systems.

However, we often find that end users are reluctant to learn an entirely new language simply to run some programs in a particular distributed system. Fortunately, many are already using a coordination language that easily expresses parallelism. The traditional **Make** [11] tool is typically used to drive the compilation and linking of complex programs, but it can be used to drive any arrangement of programs and files.

To this end, we designed a tool called **Makeflow** that implements the Directed Graph abstraction using the same basic language as Make. In many cases, users can take their existing Makefiles and use them unmodified with Makeflow. The Makeflow program reads in a directed graph, and then submits jobs individually to be executed. By changing command-line options, the same directed graph can be run on a single multicore computer, on a Condor pool, or on the Work Queue system. Makeflow keeps a transaction log, so that in the event of failure, the entire workload can be picked up where it left off without losing or duplicating jobs.

Figure 1.8 shows a very small example of a Makeflow. The user gives a set of rules, each one stating a program to run, along with the files that it requires and the files that it uses. In the example, the program `split` accepts the file `all.dat` as input, and produces the files `a.dat`

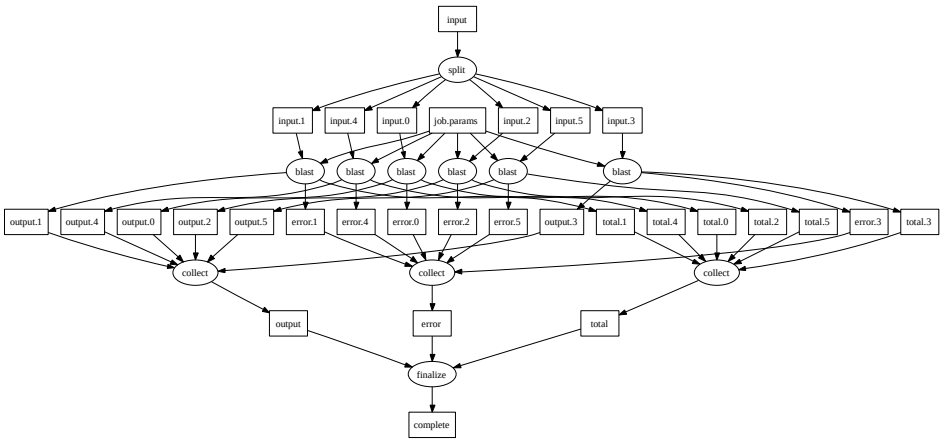

Figure 1.9: An Example Makeflow Used in Bioinformatics

and `b.dat` as output. Each of those is then consumed by the `process` program.

Figure 1.9 shows a larger example of a real Makeflow written to support a bioinformatics application. In the figure, circles represent programs, and squares represent the files that they read and write. In this particular example, the topmost program reads a large input file and splits it into many pieces. Each piece is then processed by a genomic search tool, which creates three different outputs per piece. The results must be joined together and analyzed in order to produce a final result. The system is capable of running workloads consisting of hundreds of thousands of nodes.

Makeflow is currently used as the primary execution engine for a bioinformatics research portal at the University of Notre Dame. A typical Makeflow job executed via the portal consisted of 704 tasks dispatched to the Condor pool and ran on between 25-56 cores on a designated cluster. The overall workflow consumed 686 CPU-hours in 17 hours of wall clock time, reducing the runtime from nearly a month down to less than a day.

1.8 Choosing the Right Abstraction

As we have mentioned above, some abstractions can be interchanged with each other, with some loss of efficiency. The formal relationship between different abstractions, and how to choose amongst them, remains an open problem in our field. How, then, can a user choose which one

to use for a given problem? So far, we have worked closely with our potential users to choose the appropriate abstraction for their needs. With the growing suite of abstractions, though, it is becoming important that users in various fields can select the right abstraction from the suite based on their knowledge of their own problem.

The intent of providing abstractions is for the user to define a large workload in a simple manner. The user should be able to use codes that are very similar or identical to their serial implementations. The user should be able to garner good performance without having to separately implement complicated resource management, data management, and fault tolerance mechanisms into each application.

Abstractions on the whole shield the user from difficult details about executing a workload in a distributed environment. However, it is often the case that the abstraction that fits the problem best – either due to the design of the abstraction or the way a user has defined the problem – will be more efficient due to less transformation required to scale up to the cloud and because of greater possibilities for problem-specific solution optimizations.

It is our general suggestion that a user should choose the abstraction that fits the way he already thinks about his problem. This most easily fulfills the intent of running a workload as-is, and simply scaling up to a cloud while abstracting away the messier details of the larger scale. This also usually requires the least amount of user overhead to handle the details of transforming his serial application into an entirely different problem before scaling it up.

An example of additional work required to transform the problem is seen when comparing Wavefront to a general DAG. A particular piece of a Wavefront computation can be referenced simply by coordinates in the results table. That ordered pair, when combined with the problem definition, is sufficient to enumerate all incoming and outgoing edges in the DAG. The more general DAG abstraction would need to define the problem in a less efficient manner, costing execution time to complete the transcription into the more general definition and also the disk/memory resources to store it. Even then, when executing, a general DAG abstraction would still not have the advantages of automatically being able to optimize disk and memory management to the rigid patterns of a Wavefront problem. Thus, it only makes sense for a user who is already looking at his workload as a Wavefront problem to use the abstraction that is most specific for that problem –

	Application	Problem Size	Runtime on One CPU	Runtime in Cloud
Map	Transform to Iris Code	58,639 irises	12.8 days	2.4 hours
All-Pairs	Compare Iris Codes	58,639 irises	2 years	10 days
Sparse-Pairs	Sequence Overlapping	84 million pairs	35 days	2 hours
Wavefront	Long Sequence Alignment	5.3 million bytes	13 days	8 hours
Directed Graph	Parallel Genome Search	704 nodes	686 hours	17 hours

Table 1.1: Summary of Typical Workloads

because it fits with how he has already designed his approach.

This is, however, only a general suggestion, and must be reevaluated even when scaling up the same workload. An example of a case in which this is important was shown above when discussing the Sparse-Pairs problem. A scientist may start with a fairly dense set of pairs to compute between two sets, and decide to use the All-Pairs problem. However, as the problem is scaled up and the set of pairs becomes sparser, even though the All-Pairs abstraction is still available and will still solve the problem, it no longer is the appropriate choice. Generalizing an arbitrary set of computation pairs into the superset of computation pairs will increase the amount of work he requires significantly. Not only will it require much more time to compute all the extraneous pairs that he isn't interested in, but the abstraction solving that problem will provision more remote resources (data and worker nodes, for instance) to solve the larger version.

1.9 Conclusion

In this chapter, we have demonstrated several abstractions for cloud computing. An abstraction allows an end user to express a very large workload in a compact form, allowing the underlying system to handle the complexity of allocating resources, dispatching tasks, managing data, and dealing with failures. For each abstraction, we have shown a scientific application that gains significant benefit from the cloud.

Our suite of abstractions is not necessarily complete. Our experience so far suggests that a given community of researchers is likely to engage in the same kinds of computations, albeit with different underlying functions and different scales of data. This is only natural, because both collaborating and competing researchers may use the same underlying techniques and must

compare their work to one another. For this reason, one or two abstractions may be sufficient to serve a very large community of people in a given field of study. At our institution, Map and All-Pairs are common tasks in biometrics research, while Sparse-Pairs and Wavefront are useful for bioinformatics. We have found that Makeflow has broad application.

We have implemented these abstractions in the Condor distributed system because it is widely used to share computing power in academic settings. However, the same concepts can be applied to to other systems. For example, the Work Queue system can be deployed on any kind of cloud computer in order to run the same set of abstractions. Further, abstractions need not be implemented with plain programs and files as we have done, but could also be implemented in dynamic languages such as Java or C♯. using formal functions and datatypes. Such implementations would be more strongly typed and have less invocation overhead, but would of course be restricted to the given language.

For more information about these abstractions, the reader may consult our research publications [21, 16, 17, 29]. Code implementing these abstractions can be downloaded from the Cooperative Computing Lab at the University of Notre Dame at `http://ccl.cse.nd.edu`. The Condor distributed computing software is available from the University of Wisconsin at `http://www.cs.wisc.edu/condor`.

Acknowledgments

This work was supported in part by National Science Foundation grants CCF-0621434 and CNS-0643229. We thank Prof. Patrick Flynn, Karen Hollingsworth, Robert Mckeon, and Tanya Peters for their collaboration on biometrics applications. We thank Prof. Scott Emrich, Michael Olson, Ben Drda, and Rory Carmichael for their collaboration on bioinformatics applications. We thank Ryan Jansen, Joey Rich, Kameron Srimoungchanh, and Rachel Witty for testing early versions of our software.

References

[1] J. Anvik, S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling, and K. Tan. Generating Parallel Programs from the Wavefront Design Pattern. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, page 165, 2002.

[2] Iris Challenge Evaluation 2006, *National Institute of Standards and Technology* `http://iris.nist.gov/ice/ice2006.htm`, July 2009.

[3] D. Bakken and R. Schlichting. Tolerating Failures in the Bag-of-Tasks Programming Paradigm. In *IEEE International Symposium on Fault Tolerant Computing*, 1991.

[4] W. Chen, A. Radenski, and B. Norris. A Generic All-Pairs Cluster-Computing Pipeline and its Applications. In *Parallel Computing: Fundamentals & Applications*, 366–374, 2000.

[5] Condor World Map. http://www.cs.wisc.edu/condor/map, July 2009.

[6] The Hadoop Project. http://hadoop.apache.org, July 2009.

[7] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11), November 1989.

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

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large cluster. In *Operating Systems Design and Implementation (OSDI)*, 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 scientific workflows onto distributed systems. *Scientific Programming Journal*, 13(3), 2005.

[11] S. Feldman. Make – A Program for Maintaining Computer Programs. *Software: Practice and Experience*, 9:255–265, November 1978.

[12] 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.

[13] S. L. P. Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32:175–186, April 1989.

[14] D. Jordan and J. Evdemon. Web services business process execution language version 2.0. OASIS Standard, April 2007.

[15] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *International Conference on Distributed Computing Systems (ICDCS)*, June 1988.

[16] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, D. Thain, All-Pairs: An Abstraction for Data Intensive Computing on Campus Grids. *IEEE Transactions on Parallel and Distributed Systems*, accepted for publication in 2009.

[17] C. Moretti, M. Olson, S. Emrich, and D. Thain. Scalable Module Genome Assembly on Campus Grids. Technical Report 2009-04, University of Notre Dame, Computer Science and Engineering Department, 2009.

[18] T. Oinn and et al. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[19] M. Pop, S. L. Salzberg, and M. Shumway. Genome sequence assembly:algorithms and issues. *Computer*, 35(7):47–54, 2002.

[20] J. Reinganum Dynamic Games of Innovation. In *Journal of Economic Theory*, 25:21–41, 1981.

[21] B. Rich and D. Thain. DataLab: Transactional Data Parallel Computing on an Active Storage Cloud. In *IEEE/ACM High Performance Distributed Computing*, pages 233–234, 2008.

[22] A. Roy and M. Livny. Condor and preemptive resume scheduling. Kluwer Academic Publishers, 2004.

[23] 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.

[24] G. Steele. *Common LISP: The Language*. Digital Press, Woburn, MA, 1990.

[25] D. Thain and M. Livny. How to Measure a Large Open Source Distributed System. *Concurrency and Computation: Practice and Experience*, 18(15):1989–2019, 2006.

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

[27] D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[28] K. Theobald, and G. Gao. An Efficient Parallel Algorithm for All Pairs Examination. *ACM/IEEE conference on Supercomputing*, 742–753, 1991.

[29] L. Yu, C. Moretti, S. Emrich, K. Judd, and D. Thain. Harnessing Parallelism in Multicore Clusters with the All-Pairs and Wavefront Abstractions. In *IEEE High Performance Dis-*

tributed Computing, pages 1–10, 2009.

[30] 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.