Distributed Scientific Data Processing Using the DBC[1]

James Duff
Dept. of Computer Science
University of Maryland
College Park, MD 20742
USA
jimduff@cs.umd.edu

Kenneth Salem
Dept. of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1
Canada
kmsalem@uwaterloo.ca

Miron Livny
Computer Sciences Dept.
University of Wisconsin-Madison
Madison, WI 53706
USA
miron@cs.wisc.edu

Keywords: *distributed data processing, batch processing, atmospheric science, satellite data processing, Java, PVM, Condor.*

Abstract

The Distributed Batch Controller (DBC) supports scientific batch data processing. The DBC distributes batch jobs to one or more pools of workstations and monitors and controls their execution. The pools themselves may be geographically distributed, and need not be dedicated to processing batch jobs. We describe the use of the DBC in a large scientific data processing application, namely the generation of atmospheric temperature and humidity profiles from satellite data. This application shows that the DBC can be an effective platform for distributed batch processing. It also highlights several design and implementation issues that arise in distributed data processing systems.

1 Introduction

The Distributed Batch Controller (DBC) is a tool for scientific data processing. Its function is the execution of batch data processing jobs in pools of workstations. While it can be used to control jobs in a single pool, it is primarily intended for users who wish to combine more than one pool into a single computational engine. For example, a collaborative research team could use the DBC to combine the computational resources of individual team members.

The workstation pools used to process DBC jobs remain autonomous and are not assumed to be dedicated to DBC-related work. Furthermore, they may be widely distributed. For example, the DBC system we will describe later in the this paper utilizes workstation pools located in Washington, Wisconsin, and Maryland. The DBC addresses computational issues that arise in this environment. For example, pools may join or leave the DBC system at any time, the processing power available at the pools may fluctuate with time, job input data may have to be shipped to remote pools, and output data must be collected, and failures of various types must be detected and corrected. In addition, such a system must provide a centralized administrative interface to allow its users to monitor and control distributed batch execution.

A previous paper [2] described our initial prototype of the DBC and reported the results of some preliminary experiments. Since that time, DBC V1, a more stable PVM-based implementation, has been released. The purpose of this paper is to show that wide-area scientific data processing, as supported by the DBC, can be effective and practical. We do so by describing the application of DBC V1 to a large scientific data processing application. Our experience with this application

[1]CS-96-38, Dept. of Computer Science, University of Waterloo

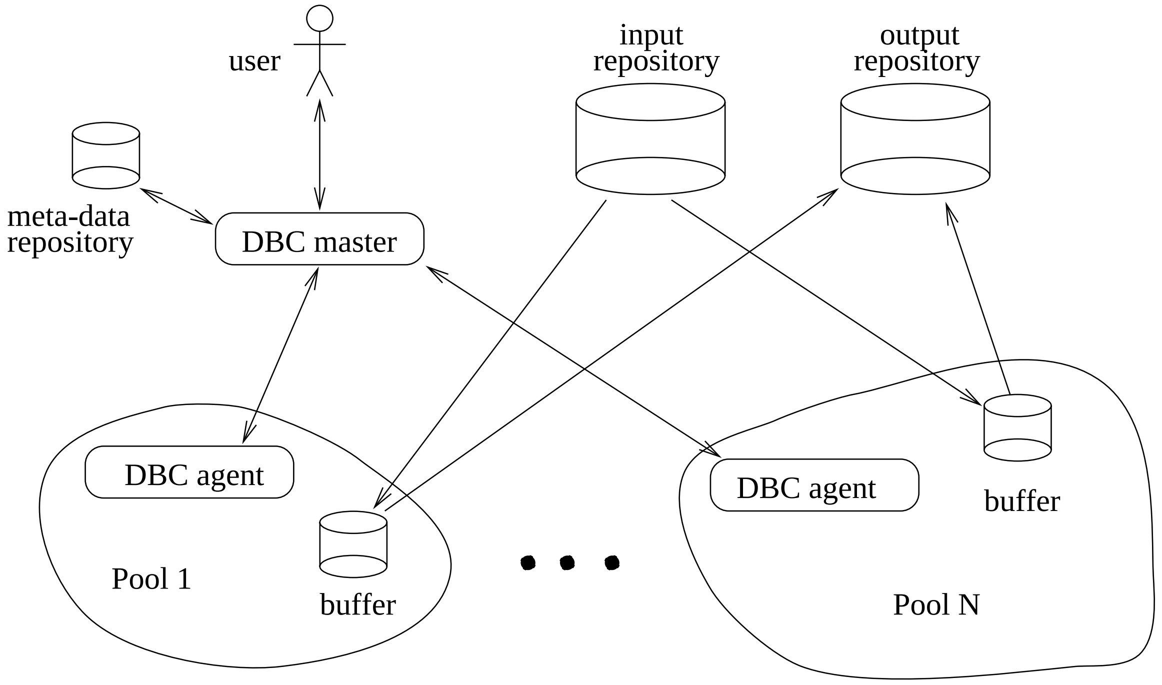

Figure 1: Architecture of the DBC

has also highlighted some of the strengths and weaknesses of the DBC design. We discuss these, and describe how we are addressing some of the weaknesses in DBC V2.

2 DBC Overview

A single DBC job consists of the execution of one or more programs, or *jobsteps*. Jobs are specified using a scripting language defined by the DBC, though the jobstep programs themselves may be written in any language. A job's script may include a set of formal parameters. Typically, these parameters are used to represent input and output file names and run-time parameters for the jobstep programs. Once a script has been prepared, jobs may be created by supplying the necessary actual parameters. A batch is a set of jobs, each an instance of the same script. In the scientific domains to which the DBC is targeted, batches are often large and long-running.

A DBC run-time system is used to process a batch. Each instance of the run-time system processes a single batch. However, nothing prevents multiple DBC systems from running concurrently and sharing available resources. Figure 1 illustrates the architecture of the DBC run-time system.

Computational power is provided to the DBC by one or more resource pools. A DBC agent controls jobs at each pool, and a single DBC master coordinates the agents. The master maintains information about the entire batch, assigns jobs to agents, and is informed when jobs are completed. The master also supports user interfaces that provide a central point of control for a human administrator.

A DBC pool consists of one or more "execution resources", plus some disk buffer space. An execution resource is something, e.g., a workstation, or a cluster of workstations, that is capable of executing the jobstep programs. Jobstep programs running at any of the execution resources in a pool are are assumed to have access to the pool's common disk buffer. The collection of pools that comprise a DBC system is dynamic. Pools are autonomous and may join, leave, and rejoin the system at any time. If a pool leaves the system, the master may reassign its tasks to other pools.

Each pool's agent manages all DBC jobs at that pool. The agent transfers the job's input files from the input repository to its pool's disk buffer. (DBC jobs are not assumed to have direct access to data in the repository.) It then arranges for the jobstep programs to run on the pool's execution resources, and monitors their progress. If the jobstep programs generate output data, the agent moves them from the local disk buffer to the output repository.

The DBC agent has a modular design and can be customized to operate with different types of execution resources. In DBC V1, each execution resource is a workstation cluster managed by the Condor [1, 4] resource management system. This allows the DBC to exploit the power of idle workstations, since Condor assigns jobs to unloaded machines in the cluster. The DBC agent submits jobs execution requests to Condor, which in turn takes care of assignment of jobs to individual workstations, and job migration (within the cluster) when necessary.

3 Implementation

The scientific application discussed in the next section uses Version 1 of the DBC. In Version 1, the DBC master (see Figure 1) is implemented by a master process plus one or more front-end processes, which provide user interfaces. The DBC agents are implemented by an agent process, plus a job-monitor process for each job being managed by that agent. Communication among these processes is implemented using the Parallel Virtual Machine (PVM) [3] system.

All front-end processes communicate with the master process through a standardized API. The API defines methods for starting and stopping pools and jobs, for monitoring job progress, and for setting some system parameters. A number of useful front-end processes have been developed. The *console* front-end is a text command interpreter that generates API calls. The *dbc_submit* front-end parses pool descriptions, job scripts, and job creation commands and submits them to the master process. There is also a simple GUI tool for visual monitoring of batch progress. Since there is a standard API, users can also write custom front-end programs for specific applications.

The master process is responsible for spawning pool agent processes, and for providing jobs to those agents at their request. Agent processes are started by PVM using the Unix remote shell mechanism. Once running, an agent requests jobs from the master and spawns (using PVM) a job-monitor process for each job it is assigned. Each job-monitor arranges for its job's input and output data to be transferred to and from the repositories by spawning calls to the Unix FTP [5] utility. It executes its jobstep programs by submitting execution requests to Condor.

Condor [1, 4] is a resource management system that runs on pools of Unix workstations. Condor harnesses the computational power of unused workstations in the pool. Jobs submitted to Condor, including those submitted by DBC job-monitors, are sent automatically to idle workstations in the pool for processing. Should a workstation become busy, Condor is capable of checkpointing and moving its job to another machine. However, we are currently using Condor in a mode in which such jobs are simply restarted.

Since the DBC uses Condor, it effectively draws its computational power from idle workstations in each pool. Jobs submitted to Condor by a DBC job-monitor must compete with other Condor jobs for the available resources in a pool. Job scheduling policies may vary, as a Condor pool's owner can set the scheduling policies for that pool.

4 A DBC Application

DBC V1 is being used as a data processing engine for the TOVS Polar Pathfinder (TOVS Path-P) project at the Polar Science Center Applied Physics Laboratory (APL) at the University of Washington. This project, part of the NOAA/NASA Pathfinder program, is producing daily Arctic

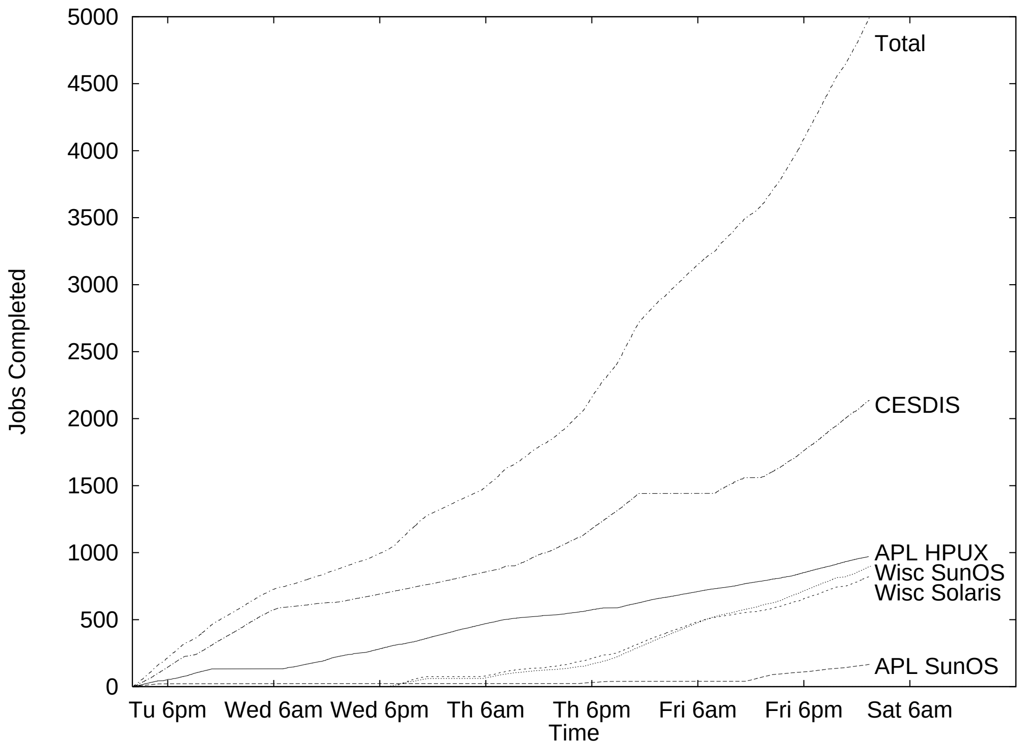

Figure 2: Progress of the a Typical DBC Batch Run

atmospheric temperature and humidity profiles for a 20 year period. Profiles are being generated using data from approximately 100,000 orbital passes. Each execution of the profile retrieval program uses two input files, one containing infra-red data and the other containing microwave data. The total size of these files is about 350 kilobytes compressed. The program generates about 1 megabyte (compressed) of output data.

The time to execute the profile retrieval program ranges from about five minutes of CPU time per orbit on a dedicated Sun Sparc-20/60, to about twelve minutes per orbit on a Sparc-10. Assuming no I/O-related delays, sequential processing of this data set would occupy such a workstation for approximately one to two years. Since the APL team expects to be reprocessing the data two or three times, this application alone may represent three to five workstation-years of computing.

The DBC job script for this application contains a single jobstep program which uncompresses the input files, runs the profile retrieval program, and compresses the output. The DBC run-time system used consists of three widely-distributed workstation pools. The largest pool is at the University of Wisconsin Computer Science Department in Madison, Wisconsin, consisting of about 40 Sun/Solaris workstations, 30 Sun/SunOS4.1.3 workstations, and a 180 megabyte disk buffer. The second pool is at the Center of Excellence in Space Data and Information Systems (CESDIS), located at the NASA Goddard Space Flight Center in Maryland. It consists of 14 Sun/SunOS4.1.3 workstations and a 160 megabyte disk buffer. The third pool consists of the APL's own workstations, including two Sun/Solaris workstations, four HPUX workstations, and a 100MB disk buffer. All three pools are shared by other users, so the number of workstations available for DBC jobs is normally much less than the total.

A RAID storage device at the APL in Seattle is used as both the input repository and the output repository for the system. The DBC master is also located at the APL site. Jobs are processed in batches of 500 to 5000 at a time, and the DBC is typically shut down between batches.

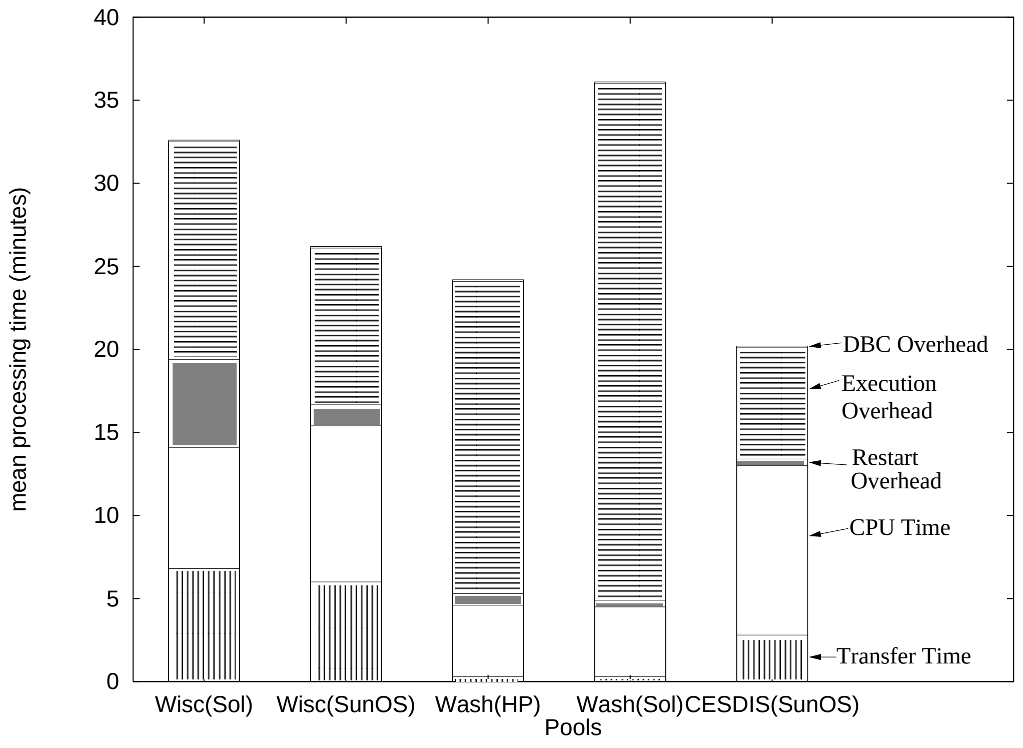

Figure 3: Mean Processing Time at Each Pool

(This mode of operation is preferred by the APL, and is not a requirement of the DBC.) So far the DBC has been used to process over 40,000 jobs.

Figure 2 shows jobs completed as a function of time for one typical batch of 5000 jobs. The figure shows the jobs completed by each pool, as well as the total. Because the pools at Wisconsin and APL each included two different types of machines, we have included two lines for each of those pools in the Figure. Each line represents jobs completed by machines of a particular type in a particular pool. This batch took 86 hours to complete, for an overall throughput of 58 jobs per hour. The remote CESDIS and Wisconsin pools contributed substantially, processing 77% of all the jobs in this batch.

Figure 2 shows fluctuating processing rates for the various pools in the system. Fluctuations occur as workstation availability changes in the pools, or in some cases because a pool temporarily leaves the DBC system. For example, the flat portion of curve for the CESDIS pool on Thursday night was due to a local file server failure during that period. The Wisconsin pool was not activated at all until Wednesday evening, (about hour 30 of the run), as new software was being installed there. Although individual pools were not always operating, the DBC as a whole continued to make progress for the entire duration of the run. This illustrates one of the advantages of distributing batch execution over several pools: problems in one pool need not result in the complete suspension of data processing.

An analysis of DBC job execution times can provide some additional insight into the system's behavior. Figure 3 shows a breakdown of the job response time for each machine type at each pool. A job's response time is measured from the time it is first assigned to an agent by the master to the time the agent reports the job complete. In the figure, the total response time is broken into five parts:

- Transfer Time: the time required to transfer both input and output data to and from the repository.

- CPU Time: processor time spent executing the profile retrieval jobstep program.
- Execution Overhead: this has several sub-components, including Condor scheduling overhead and resource queueing time, and any delays due to file system operations once the jobstep program has started executing at a workstation. Since the profile retrieval program is CPU-limited, we expect that delays for file operations are minimal for this application. The Condor scheduling overhead is due to polling performed by Condor to detect the arrival of newly-submitted jobs. Once a submission has been detected, it must then wait until Condor assigns it to an idle workstation for processing. This is the resource queueing time. It depends on how heavily utilized the pool is, and on Condor's scheduling policies, which attempt to allocate pool resources equitably among many users.
- Restart Overhead: Since Condor assigns work only to otherwise idle workstations, a job may be preempted if its assigned workstation becomes busy. In this case, Condor will automatically restart the job on another workstation. Time spend executing before the preemption is wasted, and is called the restart overhead.
- DBC Overhead: time spent in the DBC itself. This is calculated as the amount of time remaining after the other times are subtracted from the total job response time.

The DBC system directly measures total response time, transfer time, CPU time, and the total overhead (the sum of execution overhead and restart overhead) for each job. Most jobs (94% over all pools) do not restart. Such jobs have no restart overhead, so our total overhead measurement for these jobs is really a direct measure of execution overhead. For jobs that do restart, we estimate their execution overhead to be the same as that of jobs that do not restart, and take the restart overhead as the balance. The times shown in Figure 3 are weighted average times over both types of jobs.

The data in Figure 3 illustrate several characteristics of this particular DBC system. First, the data repository itself was not a performance bottleneck. If it were, we would expect to see much larger transfer times at the Washington pools. Second, the remote pools do experience substantial data transfer delays due to the limited network bandwidth available. However, as long as sufficient buffer space is available at each pool, the DBC agent can hide this latency by transferring data for some jobs while it is processing others. Thus, we do not believe that data transfer was a performance bottleneck for this system, even at the remote pools. Third, the execution overhead time was substantial at all of the pools. We know that only a few minutes of this time is due to scheduling overhead (Condor polling). The remaining time is spent waiting for a workstation to become available. These data show that the system's performance at all of the pools was limited by workstation availability. Finally, the data show that the DBC itself does did add any significant overhead cost.

5 Design and Implementation Issues

Our experience with the DBC has brought a number of design and implementation issues into focus. Three of these, namely failures and recovery, security, and meta-data management, are discussed here.

Pool	Throughput (jobs/hr)	%Downtime	Ideal Throughput
Wisconsin/Solaris	6.4	58.3	15.3
Wisconsin/SunOS	6.8	50.8	13.9
CESDIS/SunOS	7.0	72.5	25.5
APL/Solaris	4.5	48.3	8.6
APL/HPUX	12.6	23.7	16.5
APL/SunOS	1.3	88.5	6.1
Total All Pools	38.6		85.9

Figure 4: Throughput at Each Pool

5.1 Fault Recovery

Failures are inevitable in a long-running distributed system like the DBC. The DBC has been designed to anticipate many types of failure, and much of the implementation effort for DBC V1 has gone into failure detection and recovery. Failures can occur at the global, pool, and job levels. We have found that while failures at the global and individual job levels are rare, and easily handled, pool failures can have a major impact on DBC performance.

To examine the impact of pool failures, we analyzed data from a set of seven batches of profile retrieval jobs (about 15,000 jobs total). The first column of Figure 4 shows the average throughput for each pool, measured over all of the batches, and the second column shows the percentage of time that each pool was down while the DBC was running. The third column shows the throughput we would expect if pools never went down, assuming their average processing rate remained unchanged. If all pools had run without failure, the overall power of the DBC would have been 85.9 jobs per hour, more than twice the measured value of 38.6 jobs per hour.

Not all pool downtime is caused by failures. For example, during the batch execution shown in Figure 2, the Wisconsin pool was down during the first thirty hours of the batch because new software was being installed and tested. However, we know that failures are common. They typically occurred about about once a day at the APL and CESDIS pools, and about 3 times a day at the Wisconsin pools. We have observed failures caused by many things, including broken master/agent TCP/IP connections, local network problems at a pool, Condor software failures, and process overload. The latter problem is exacerbated by our DBC V1 implementation, which uses one job-monitor process per job assigned to a pool.

Another cause of pool failures is a mechanism built into DBC V1 that will automatically shut a pool down if the job failure rate at that pool exceeds a threshold. A small fraction of jobs fail the first time they are executed. Some of these failures are due to application problems such as bad input data or bugs in the jobstep programs. However, others are due to system-related causes such as heavy Condor usage (causing a submitted job to eventually time out) or Internet failures that block repeated attempts to transfer data from the repositories. These types of failures tend to be bursty, and will sometimes manifest themselves as a pool shutdown by causing the threshold to be exceeded.

When a pool fails for any reason other than the automatic shutdown mechanism, the master attempts several times to restart it before eventually giving up. Pools that succumb to the automatic shutdown mechanism are not restarted automatically, since jobs are likely to fail there again if the underlying problem is not corrected. When automatic restart fails or is not employed, it is up to the DBC user to restart the pool manually by issuing the appropriate command from the DBC console. It is our experience that the DBC must still be monitored several times a day to determine if manual intervention is required.

Version 2 of the DBC will incorporate several design changes that we hope will reduce the cost or the likelihood of failures. In particular, DBC V2 will combine the job-monitors into a single multi-threaded process, and will permit disconnected pool operation in the event of a broken agent/master connection. We will return to the discussion of DBC V2 in Section 6.

5.2 Security

Our current design requires that the DBC user have 'login' access to a workstation at a pool before that pool can be included in a DBC run-time system. Our design is not intended as a mechanism for allowing strangers to use a pool's idle resources. Rather, the owners of the pool must be willing to share their resources with the DBC users.

We assume that a pool's owners trust a DBC user as they would any other user with login privileges on pool machines. The DBC does not allow an unscrupulous user to exploit the pool in any way they could not already do with login access. However, even if we assume that DBC users are trustworthy, there are still a number of security concerns that need to be addressed in a system such as the DBC.

First, a mechanism is needed for authenticating remote users when they log in to a pool machine. This problem is common to many Internet applications. The DBC could exploit a general purpose, secure facility for remote process creation, if such existed. Second, once running, a DBC agent should authenticate job requests: "Did this job really come from the Master?". This is to prevent unauthorized users from trying to exploit a running pool. Other than the mechanisms built into PVM, we do not address this problem in the current design. The DBC V2 is being built in Java, which does provide some control over the creation of socket connections with outside processes.

Each of the mechanisms described above would protect the owners of pool resources. It may also be desirable to provide protection for the DBC user. In particular, the master should authenticate messages from agents. This would prevent 'rogue' pools from joining a running system. Rogue pools could sabotage a batch by processing jobs incorrectly, or they could steal input/output data. There is no such authentication mechanism built into DBC V1. However, all agents are initiated directly by the master, which is under the control of the DBC user.

5.3 Meta-Data Management

Management of information about batch jobs becomes a major concern when batches are large. Meta-data are important for several reasons:

- Performance Analysis: The DBC user should be able to track how many jobs were processed by each pool and how long they took. This is particularly important for identifying performance problems or failed pools in a running DBC system. It is also useful for configuring the run-time system for future batches.

- Interpretation of Results: Different jobs in a DBC batch may run on different types of machines, depending on the makeup of the pools in the run-time system. The type of machine on which a job runs may have an impact on the output data it generates. For example, floating point numeric calculations may differ slightly on different machines. To properly interpret the output of a batch job, a record of the environment in which each job was processed is essential.

- Debugging: A DBC user's jobstep programs will not be free of errors. When a job fails, some record of its state, e.g., a "core dump", should be available for subsequent post-mortem analysis. This is complicated by the fact that job failures may be scattered widely across the pools.

Currently, the DBC makes several types of meta-data available to its users. Performance statistics are available in real time through the DBC master's console front end, or the GUI front-end. This allows the user to determine the status of any job or pool in the system, and the rate of job completion at the various pools. In addition, a variety of information about job execution environments and execution times is shipped to the master and logged. Tools are provided for subsequent analysis of the logs. All of the measurements presented in this paper were produced from analyses of DBC-generated logs. We are planning a more comprehensive treatment of these data in DBC V2, as described in the next section.

Debugging data, such as core files, are not shipped to the master because they are bulky and because they are not always wanted. Debugging data is saved in each pool's local disk buffer. The amount of data to save can be controlled by run-time parameters. Options include saving everything (including bulky core files), saving just small files (typically stdout and stderr output), or saving nothing at all other than the output data (normal operation). In the current design, saving lots of debugging data can cause a graduate decline in the performance of a pool, since the debugging data occupies space in the pool's disk buffer. A reduction in disk buffer space reduces the amount of data prefetching that can be performed, and may limit the maximum number of batch jobs that can execute concurrently in a pool.

6 Conclusions and Future Work

We have described the DBC system and its application to a large scientific data processing problem. Our work with this system demonstrates that a widely-distributed collection of workstation pools can be utilized as a data processing engine. Such a system may be particularly valuable to collaborative research teams, which may control a variety of widely-distributed computing resources. By combining the power of several pools, the time required to complete large batch processing tasks can be substantially reduced. Such a system is flexible, and can provide computing cycles even when some pools are unavailable. We have also identified some of the issues that arise in the design and implementation of such a system.

DBC V1 has been stable and in use since early 1996. We are currently developing DBC Version 2, which contains a number of changes inspired by our experience with V1. Whereas V1 is implemented in C, DBC V2 is written in Java. We are exploiting several features of the new language. In DBC V2, each agent is now a multi-threaded Java *virtual machine* that incorporates the functionality that was spread over the agent process and the job-monitor processes in V1. This moves us from a "process-per-job" design to a "thread-per-job" design, which should reduce the load each DBC agent puts on its host pool. We are using Java's remote method invocation (RMI) facility to implement communication between the master, agents, and front-ends. This replaces (and simplifies) services formerly provided by PVM. Finally, Java's security features permit explicit manipulation of DBC access restrictions.

The DBC V2 agent is designed to interoperate with various kinds of local execution resources, in addition to Condor. In particular, it will be possible to treat a single Unix workstation as a DBC pool. We also hope to interoperate with other job queueing systems, such as LSF (formerly Utopia [7, 8]) or DQS [6]. In recognition of the fact the batch meta-data is valuable and should be properly managed, the DBC V2 master will support a direct interface to a relational database system. The database systems will permit more thorough and flexible analysis of the meta-data than is possible with the simple log and query utilities available in V1.

7 Acknowledgments

We are grateful to CESDIS for providing access to its workstations in support of the DBC system described here. Axel Schweiger and Mark Ortmeyer at the APL have been patient with the system, and have provided us with access to DBC logs and other data. Victor Wiewiorowski, at the University of Waterloo, implemented the graphical front-end for DBC V1. Work on the DBC is supported by NASA through its Applied Information Systems Research Program.

References

[1] A. Bricker, M. Litzkow, and M. Livny. Condor technical summary. Technical Report TR 1069, Department of Computer Science, University of Wisconsin, Oct. 1991.

[2] C. Chen, K. Salem, and M. Livny. The DBC: Processing scientific data over the internet. 16th International Conference on Distributed Computing Systems, May 1996.

[3] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: parallel virtual machine - a user's guide and tutorial for networked parallel computing.* The MIT Press, Cambridge, MA, 1994.

[4] M. Litzkow and M. Livny. Experience with the Condor distributed batch system. In *Proc. of the IEEE Workshop on Experimental Distributed Systems*, pages 97–101, Oct. 1990.

[5] J. Postel and J. Reynolds. File transfer protocol (FTP). Technical Report RFC-959, USC Information Sciences Institute, 1985.

[6] Supercomputer Computations Research Institute, Florida State University, Talahassee, Florida. *DQS User Manual*, DQS version 3.1.2.3 edition, June 1995.

[7] J. Wang, S. Zhou, K. Ahmed, and W. Long. LSBATCH: A distributed load sharing batch system. Technical Report CSRI-286, Computer Systems Research Institute, University of Toronto, Apr. 1993.

[8] S. Zhou, J. Wang, X. Zheng, and P. Delisle. UTOPIA: A load sharing facility for large, heterogeneous distributed computer systems. Technical Report CSRI-257, Computer Systems Research Institute, University of Toronto, Apr. 1992.