Distributed Computation System: Distributed Computing in a Heterogeneous Networked Unix Environment

Russell Ruby
Brad Babin
W. G. Rudd
Department of Computer Science
Oregon State University
Corvallis, OR 97331-3902

90-80-1

Distributed Computation System: Distributed Computing in a Heterogeneous Networked Unix Environment

Russell Ruby (russ@cs.orst.edu)
Brad Babin
W. G. Rudd (rudd@cs.orst.edu)
Department of Computer Science
Oregon State University
Corvallis, OR 97331-3902

Abstract

Large integer factorization exemplifies a class of hard computational problems requiring the power of a supercomputer but which have algorithms decomposable into many large independent computations. The availability of internetworking provides the opportunity to solve such problems in distributed fashion on ordinary machines. Such a distributed network might contain a heterogeneous collection of machines running under the administrative authority of different organizations in separate geographic locations. DCS uses standard UNIX[1] BSD 4.2/4.3 features to implement a central scheduler daemon and remote grain server daemons providing support for an extended model of very large grain data flow computation. A system providing such a service needs to address the essential goals of reliability, fault tolerance, courtesy and security in addition to dataflow functionality. In particular, checkpointing becomes essential for reliability and efficiency when grain lifetimes become comparable to system uptimes. A user control program makes grain requests and receives results from a central scheduler which launches and coordinates grain computations on remote grain servers. Grain servers supply checkpoint information at regular intervals to the central scheduler or a passive backup. Grain computations on busy or failed server hosts are automatically moved by the scheduler to quiescent hosts using checkpoint information. Additionally, the system transparently recovers from temporary failure of the central scheduler or the user control program. A simple paradigm for the organization and update protocol of a reliable backing store is introduced. Other design, implementation, and operational issues are discussed. DCS was used to run the computation finding the 34 digit penultimate prime factor of a 93 digit "more wanted" number from the Cunningham Project [BRI88].

1 Introduction

This project is an outgrowth of the DRAFT[2] machine project [RUD84]. Whereas the DRAFT machine is a prototype of an alternative high performance architecture for large integer and parallel computation, the Distributed Computing System uses networking with a variety of ordinary computers and UNIX operating systems to achieve high performance using distributed processing. An important goal of DCS has been to provide a practical distributed engine for doing research

[1]UNIX is a registered trademark of AT&T Corporation.

[2]Dynamically Reconfigurable Architecture for FacToring. See Appendix D for a brief description.

with long running hard computational problems. The mobilization of large numbers of machines available on a non-intrusive low priority basis fuels this capability. DCS uses common features of UNIX systems along with traditional UNIX reliance on simplicity [ALL87] to implement a reliable, efficient, and fault tolerant system to centrally control and coordinate an extended model of very large grain data flow computation.

Other systems have been described in the literature which perform some of the functions of this system. However, none of these systems attempt to handle a fully heterogeneous UNIX environment. An interesting batch system using image based checkpoints for a homogeneous network has been described by Litzkow[LIT87]. The complexity of adding kernel supported transparent checkpointing for Cray's is made evident by a paper of Kinsbury and Kline[KIN89]. A recent paper by Smith and Ioannidis[SMI89] describes a remote fork mechanism with checkpoint/restart on homogeneous machines. Using RPC as an alternative requires the existence on the parent machine of active controlling processes for every remote client job. These processes are not independent. Failure of the parent machine causes failure of all the client jobs. RPC lacks the fault tolerant and checkpointing capabilities of the DCS system. In summary, DCS provides support for very large grain distributed processing which is not available with other networking techniques.

Pardon this introduction which is not yet complete, being one full iteration behind the following sections 2,3, and 4. There is a substantial amount of material still missing from this section, including a discussion of the relationship of DCS with other distributed processing systems such as LINDA and ISIS. The following sections describe the design and enhanced features of the next version of DCS currently being developed. However, from the user's point of view, most of the basic features already exist in the current implementation. The example control programs and example grain programs with checkpointing found in the current revision level of source code distribution closely parallel the description and examples found in this paper. The material found in sections 5,6, and 7 of this paper vary in version level between the existing implementation and part way toward the revised design. Unfortunately, there is a substantial amount of new design discussion which has not yet made its way into the last few sections and some sections have become quite outdated.

2 Model of Computation

Algorithms are not monolithic blobs. Decomposition into computational grains can be made on a scale from the grain size of modules to as small as individual machine instructions. A grain of computation can be viewed simply as a functional unit of computation which transforms an input string representing its data token arguments to an output token string. The data flow model represents the decomposition of an algorithm as a directed graph, with nodes corresponding to grains and the directed arcs representing the movement of data tokens [HWA84]. Unlike the chain of processes in a UNIX pipeline, where a process can continue producing data while it is being consumed by its successor, data flow grains terminate before their successors begin computation. That is, each grain computation fires when all its inputs become completely known.

Strictly speaking, the dataflow graph fragments the view of accumulated grain data and state information to just those grain outputs or tokens corresponding to the input edges to the grain. Although it would be possible with such a model to enhance the grain's view to encompass all the accumulated state information, the increase in structure and complexity of the dataflow graph would be enormous. Rather than produce such an enormous graph corresponding to distributed control at the node level with complete access to accumulated information, it is possible to gain the same benefit without the appearance of overwhelming complexity. The important ingredient from the dataflow paradigm is the notion of decomposition into functional units bare of side effects to other executing grains. Although we desire to exclude side effects between grains, we welcome the seeming use of side effects for control of the overall dataflow computation. Control of these grains may benefit from having an complete view of all completed grain output and messages from partially completed grains. The trace of such a dataflow computation simply corresponds to a dynamically created dataflow graph which effectively has links from all completed grains and partially completed grains to each new grain as it prepares to fire. As the control is central, being done by a program which is either sequential or parallel with shared locking memory, it is possible to keep a sequential log containing the order of finished grains and relevant grain messages. This sequential ordering is sufficient to retrace and recreate the implied dataflow graph up until a point of failure for recovery purposes or else to create the graph for the entire computation. Note that for small grain decompositions this log trace of the computation could become unwieldy, becoming perhaps millions of entries in length. However, for large grain decompositions this extended dataflow model becomes a reasonable and effective tool merging the benefits of functional decomposition and flexible control.

Observing figure 1, all the diagrammed components of DCS communicate through UNIX IPC sockets. The user's control program provides the central control for the computation, making grain requests and receiving information and output from the central scheduler. The central scheduler manages the grain computations using a pool of remote grain servers. The central scheduler maintains a reservoir of accumulated output, grain checkpoint data and feedback messages, logging and other system information.

Let us first consider the effect of grain size using just the simple data flow model with DCS as the computational platform.

A small grain data flow view of an algorithm performing arithmetic computation might consider each addition or multiplication as an independent grain of computation, proceeding when its operands become available. Logically independent computations could occur in parallel, and in turn would feed their respective results as operands to succeeding grain computations. With each small grain computation occurring on a remote host, DCS would perform miserably given the unavoidable high overhead for grain handling relative to the size of the grains.

On a medium scale, the grain size might be the size of short lived procedures, allowing independent procedures to be invoked in parallel whenever their arguments become available. Decomposition of a program into independent procedural sized pieces might occur naturally by partitioning data into sets which can be acted on simultaneously, or by unwinding a recursive divide and conquer formulation of a program. On such a scale, DCS does become an acceptably efficient platform, but perhaps at first glance just another remote procedure call mechanism.

On a larger scale, the grain size could be that of complete long running programs. It now becomes prudent to checkpoint the grain computations and use the checkpoint information to support automatic migration of grains from busy or failed hosts. Additionally, grain computation should not be lost or fail even if the user's or central scheduler's host becomes temporarily disabled. Even if system crashes occur only rarely, scheduled shutdowns for backups and maintenance produce system interruptions. Thus a primary concern of a system supporting long running distributed computations should be to handle host interruptions transparently with minimal loss. In contrast to DCS, typical remote procedure call mechanisms fail completely if either the user's local host or remote call host fails during computation.

Returning to Figure 1, let us continue with an overview of DCS. The user control program is written in a high level language, such as C, and employs a tool set of library calls which individually communicate with the central scheduler. These calls provide the interface to initiate sessions, submit grain requests, gather grain output, messages and other information, and otherwise direct and monitor activity. These calls are described in the next section. Unlike a simple data flow system, which follows the paths of a fixed data flow graph, the DCS model encourages the use of feedback from the total computation to alter the course of the computation or perhaps abandon useless grain computation. Of course such a computational plan could be formulated a priori as an enormously complex dataflow graph, but it is simpler to put the cart behind the horse and consider the dataflow graph as the dynamic trace of centrally controlled grain executions.

The central scheduler acts as shepherd for the grain requests it receives. It maintains ready queues for grains waiting for execution, and launches grains when an appropriate host is available. The remote grain servers call the scheduler when server status changes and at regular intervals to report grain checkpoint or output, and system status information. The scheduler detects busy or failed remote server hosts, moving grains on such hosts to appropriate alternates or to the head of the appropriate ready queue. If the central scheduler's host becomes temporarily disabled, current grain computations continue unaffected with the grain servers calling a passive backup daemon on another host with their checkpoint and status reports. When restarted, the central scheduler rebuilds its internal data structures using both its own state information saved on disk and information from the passive backup.

3 User's view of DCS

The first task for the user is to find a highly parallel decomposition of the problem into many large sized grains. This group of grains is referred to as a *session*. Once the decomposition is determined, each grain is then implemented as an individual executable program with optional support for checkpointing. Input and output for each grain occurs through standard input, standard output and standard error, along with the standard *execve* parameters *argv* and *envp*. Written as ascii text, the standard input file first specifies the initial state of the computation, followed by input data for the computation. If standard input would have a length greater than a rather generous installation defined limit, then arrangement should be made to read excess data read from a separate file specified in argv or the environment. Such secondary files are a convenient way to provide large read only tables of data for repeated instances of a similar grain computation. These conventions ease the development, debugging and testing of individual grains. Further discussion on requirements and technique for providing input and checkpoints for grain programs occurs in section 3.3. First, we discuss the coordination of the distributed computation by the user's control program.

3.1 Control program communication and authentication

To manage the distributed computation, the user prepares a control program written in a high level language, such as C. The control program weaves the grains of a session together, using the DCS library calls to initiate, monitor and retrieve grain activity. The control program can be run on either the scheduler's host or on remote hosts permitted by the scheduler. The DCS library routines on the scheduler's host use UNIX domain stream sockets, and on remote hosts utilize Internet domain stream sockets. The overhead of stream sockets has not been a bottleneck, so the development of an ad hoc reliable datagram protocol has been a low priority item. Although some transaction requests could reside within a single datagram, others would require the stream socket benefits of packet sequencing and reconstruction.

For security, it is mandatory that not even a user with knowledge of the DCS source code be able to impersonate a valid user. To accomplish this level of protection, before beginning the user control program the actual user is identified to the scheduler in a secure fashion. Permission is then granted to a valid user by the scheduler transparently providing an authentication token associated with that user. The identification is performed by the `dcs_begin` command which is used to invoke the control program with all its possible arguments and standard input/output redirection.

```
        dcs_begin control_program [args ...] [< file1] [> file2]
```

Execution access to the `dcs_begin` command is restricted to members of a special dcs group. The `dcs_begin` command runs with root privilege and is thus able to use an internet domain privileged port to connect to the scheduler and reliably report the invoking user's name. It leaves open the socket connection to the scheduler and does an `execve` of the user's control program. The control program immediately makes a `d_get_auth(control_ident)` call in order to invisibly receive the authorization token through the inherited socket connection to the scheduler. through the inherited socket connection to the scheduler. The token is held as a static variable within the dcs user library and is not visible in the control program. This token is then implicitly used with all the control program's user library calls to the scheduler. The calls are accepted only if the authorization token is valid and the token's user name bound to the token matches the user name provided with the library call.

The scheduler maintains an authorization token table with each entry containing the following information: the actual token; the user's name; the control program's `control_ident`; the number of active sessions associated with the token; and an integer field for a timestamp. The user name and the `control_ident` are specified by the `dcs_begin` call along with the control program's `get_auth` call. The timestamp is stamped initially and is stamped again whenever the active session count becomes zero. This table is checked periodically to remove authorization entries which simultaneously have an active session count of zero and are older than a installation defined amount. Each active session in the scheduler's database has a pointer to its current authorization token's entry. This has a desired side effect of limiting authority over a session to only one control program

```
d_get_auth(control_ident)         name control program and retrieve authorization token
d_open_ses(session_num)           open a new session
d_resume_ses(session_num,sent_reset)     resume active session and optionally reset
                                         count of grains sent to control program
d_reopen_ses(session_num,sent_reset)     reopen closed session and similarly reset grains sent
d_close_ses(session_num)          close session, remove active grains
d_gexec(session_num,grain_num,gargv,genvv,input_str,host_class,thrash,urgent,notice)
              submit a grain for remote execution on a grain server host
d_gwait(block,session_num,&g_num,&g_completed,&g_sent,&g_termsig,&g_retcode,
        &g_rusage,&g_outlen,&g_output,&g_errlen,&g_error)
              receive results from the next finished grain
d_gkill(session_num,grain_num)           terminate the grain
d_query(query_type,session_num,grain_num,&info_block)
              query the scheduler for system, session or grain information, return in info_block
```

Figure 1: DCS user library calls

at a time. The various library calls for beginning, resuming, or closing a session appropriately set or unset the session's token pointer along with incrementing or decrementing the active session count fields in new or old tokens.

Unintentionally, a user might start a second control program which uses a session number already in use by the same user in another control program. If the existing session is opened by the `d_resume_ses` call which is used to write simple restartable control programs, then the authorization token for the session might be changed and the connection with the original control program broken. However, the control program's `control_ident` helps prevent such accidental disconnections. The `d_resume_ses` call fails if the `control_ident` bound to the token of the second control program differs from the one belonging the token currently associated with the session. It succeeds when the `control_idents` are the same, allowing a restarted control program to reconnect to an orphaned session.

Additionally, there is a set of secure commands which can be invoked by the session's user, scheduler_master or root in order to manipulate a session directly without altering or requiring the use of the authorization token.

3.2 User library interface

Each user session is uniquely identified by the attribute pair user name and session number. The user can have multiple sessions running simultaneously, possibly under the direction of a single control program. Each possible pair of user name and session number corresponds to a session which is non-existent, active, or terminated. The session number is specified as a parameter when making library calls to manipulate sessions and their grains. With all the library calls the user's name is implicitly determined and passed to the scheduler which checks to see that it matches the authorization token's user name. Only grains of active sessions are allowed to execute on a remote server or reside in the scheduler's ready queues. Any grain of a previously initiated session has priority for execution over all grains of later sessions. See figure 3 for a listing of the DCS user library calls.

The `d_open_ses` call initiates a new session by creating the appropriate internal data structures, backing store disk directory and entries. The call fails and returns a corresponding error status if the session is active or terminated.

The d_resume_ses call is used when writing restartable control programs according to techniques described later. The call succeeds and opens a new session if the session designated by the session_num parameter is non-existent. If the session is active and its authorization token's control_ident matches that of the call's token, then the call succeeds and replaces the session's previous authorization token with its own. The second parameter sent_reset can be used to reset the session's counter for grain results returned to the control program. The counter should be reset when using the simple idempotent approach for control program recovery, and not reset if the control program recovers using its own saved state information. If the session is closed or the tokens' control_ident's don't match, then the the call fails and returns an appropriate error status for each failure case.

The d_close_ses call terminates an active session. This action is first asserted by updating as closed the session's status maintained by the backing store. Any of the session's grains executing on remote hosts are removed, and any related data structures on the remote hosts are cleaned up. Additionally, these grains have their status as maintained by the backing store changed to ready. All the scheduler's in memory data structures for the session are purged. However, all the session's backing store data including grain status, checkpoint, output, and other session information is retained. This directory and its contents are owned by the user and are thus available for examination or manual removal. The knowledgeable user can modify the directory contents and reactivate the session by using the d_open_ses call. This call succeeds only with terminated sessions and rebuilds the session's in memory data structures and ready queues using the session information held by the backing store.

The life cycle of a user session begins with non-existence, followed by creation and being active, followed by termination, and finally return to non-existence with the removal of the terminated session directory. The d_clean_ses call succeeds only with a terminated session, removing all the terminated session's backing store data and directory.

Once a session exists and is active, the user's control program can submit grain requests through the d_gexec call. For active sessions, grains can be in one of four states: ready, executing, killed or finished. To prepare for a grain submission, the control program constructs the appropriate argument vector and environment vector for the grain's eventual *execve* on a remote server, along with the string which is used for the grain's standard input. Absolute file path names can be used in gargv, genvv and referenced in grains, but it is more convenient and practical to use path names relative to a DCS designated home directory for the user on remote grain servers. Gargv contains the path name of the grain executable as argument zero, which is used as the name parameter for the grain's eventual *execve* call. The last component of this name is used as *argv[0]* and the remainder of gargv and genvv correspond to the normal *execve* arguments *argv* and *envv*. Besides these parameters specifying the grain program and its input, there are four more parameters pertaining to grain placement, diplomatic and operational considerations.

First consider the host_class parameter. The central scheduler maintains a database of all possible grain server hosts. Each host is designated as belonging to one of thirty two possible host types defined during system installation by representing each host type as a single bit in an unsigned long integer. Typically, considerations such as vendor, architecture, peripheral resources, and operating system characteristics are used to partition the machines into host types. Additionally, administrative, political or geographic considerations can be used. The value of the host_class parameter in the d_gexec call is specified by an OR of the acceptable host types for execution of the grain. When multiple acceptable types are specified for the grain, the types have an implied ordering according to their numerical value as unsigned integers. That is, when the grain is being assigned to a remote host, hosts of the least valued type are considered first, if an acceptable host is not available, then the next higher valued type is tried and so on. Given that the initial input, checkpoint information, and output are written as ascii text, all grain input/output can be written so as to be independent of host architectural differences such as byte ordering. This has proven quite useful for targeting a particular grain computation for a set of host types of similar overall capability for that particular computation, even though their vendors, architectures, or operating system versions may differ. Additionally, the environment variables can be used to help provide information that grains might need to tailor their execution on dissimilar hosts. During its lifetime, the grain is allowed to migrate within the pool of machines designated by its `host_class` parameter. This migration is forced by either a host becoming busy or failing, or by usurpation of the host by a grain belonging to a higher priority session.

Another important concern of DCS is to avoid noticeable interference with the performance of normal users' processes on the remote grain server hosts. Accordingly, when a grain server creates new grains, before *execve*'ing the new grain, a *setpriority* call is done to nice the child process to minimal priority. For compute bound grains with small memory images and little paging this has proven adequate to avoid interference with normal users' processes on the grain's host. However, grains with large images and considerable paging wreak havoc on competing processes no matter how low the grain's priority. An advisory integer parameter, `thrash`, provides a rough measure of the expected memory load of the the grain in half megabyte units. The scheduler compares this value with the host statistics it keeps to avoid sending the grain to a remote host whose current free memory is considered too low. If the value of thrash is zero, then memory load is not considered to be a problem. For all executing grains, the remote server and the grain itself monitor page faults and the grain is withdrawn and returned to the scheduler if thrashing occurs.

A boolean parameter, `urgent`, indicates that if no suitable grain server host is available, then place the submitted grain at the beginning of the session's ready queue, rather than at the end. Migrating grains are also placed at the beginning of the ready queue if a suitable alternate host is not immediately available. The last parameter, `notice`, is the number of minutes advance notice a grain should be given before checkpointing occurs.

There is an additional important characteristic that the `d_gexec` call possesses that assists with the writing of simple restartable grain programs. If the session has been successfully opened with the `d_resume_ses` call, then the `d_gexec` call exhibits idempotent behaviour. Specifically, this means that if the grain already exists, then regardless of its current state, if the parameters of the `d_gexec` call are identical to those of its original call then the call becomes a null operation and returns successfully. However, if the grain's gargv, envv or input parameters differ, then the call fails. If the call has not occurred before, then the call proceeds as usual.

The control program can abort a particular grain computation specified by its session number and grain number parameters with the `d_gkill` call. All the information related to the grain kept on disk is retained, but the grain's status is marked as killed. If it is executing on a remote grain server, the grain is removed and its related data structures and administrative entries purged on the remote server. Likewise, on the scheduler the grain's entry and data are purged from the session's ready queue or execution list. The grain's status as killed held in the backing store prevents resumption of the grain's computation as a part of the normal DCS system recovery following a scheduler host crash and reboot.

To receive grain output, the control program uses the `d_gwait` call to retrieve the output of the next unreported finished grain computation for a session. The user specifies the session number and several output parameters for receiving grain output and status information. The boolean parameter `block` determines whether the call uses blocking or non-blocking mode. In non-blocking mode, the call returns immediately with an error if every completed grain has already had its output reported by a previous `d_gwait` call. Otherwise the call will return with the status and output corresponding to the next unreported finished grain computation. In blocking mode, if unreported grain output is available, it behaves just as in non-blocking mode. However, if no unreported grain output is available, the `d_gwait` calling process opens a listen socket, informs the scheduler of its address, and waits for a socket connection from the scheduler. Eventually, when one or possibly more of the session's grains finish, the scheduler calls the listen socket and sends the output and status from the first of the unreported finished grain computations. The various output parameters include the identifying number of the grain being reported, the current number of grains completed in the session, the number of grain results already delivered, and the return code and termination signal of the grain. In normal operation the return code and termination signal are always zero, as otherwise the grain is not considered finished. Unfinished grains are not reported unless they are killed by the system following a fixed number of failures while running on different hosts.

Two more output parameters return the lengths of the grain's stdout and stderr files. Different variants of `d_gwait` return the grain's standard output and standard error either as strings or as pointers to streams open for reading. The alternative of strings returns up to the first 20 K characters from each file. As streams, there is no problem with truncated output, but this option is not available from hosts which do not have remote access to the scheduler's backing store data files. The user's control program parses the grain's output, using the information extracted to alter its own global state and to prepare future grain requests. For simple restartable control programs, when the session is opened with the `d_resume_ses` call with the `sent_reset` flag set, the scheduler's counter for the number of grain results returned to this session is reset to zero. Consequently, successive `d_gwait` calls return all the grain results in the same order as they originally occurred.

The `d_query` call is useful both in the user control program and for monitoring the operation of DCS. For monitoring, there is a tailored set of user commands using the `d_query` call providing formatted listings of system status information. For example, the command for hosts displays lists for "active" and "delinquent" hosts, providing host name, configuration and statistics information. Other commands provide listings of various system information and attributes including: user sessions; the executing, ready, and killed grains for a session; all executing grains; all ready grains; all killed grains. Whereas these general status commands only require the restricted DCS group privilege, other commands to retrieve detailed grain information succeed only for the owner of the grain, the scheduler_master, or root. Using the appropriate options, any of the grain information is available such as rusage, number of restarts, current status, initial grain input, most recent message and checkpoint/restart, or the first 20 K of output. Called from a user control program, the `d_query` call provides access to the same data. As with the `d_gwait` call, it is possible to get a open file pointer for the grain output files.

With any of the user library routines, if the attempted connection with the scheduler does not succeed or the socket connection is broken, then the routines return with a corresponding error status.

The DCS library also includes routines supporting administrative tasks. These routines are built into secure commands which can only be invoked by the scheduler_master or root. As indicated previously, there are commands to manipulate sessions and also commands to manipulate individual grains. Some of the grain host configuration parameters can also be changed by command. In particular, changing a grain host's allowable grain limit to zero causes any grains running on the remote server to be removed immediately, and effectively removes that host from service.

3.3 Checkpointing grain programs

A grain program is responsible for providing its checkpoint information. The checkpoint information captures and specifies the complete state of the grain computation so that it can be moved to a new host in case of host failure or forced withdrawal. To utilize checkpointing, the grain program is written to load the starting state of the computation from an initial portion of standard input. Consequently, when a new or restarted grain begins executing, the grain reads standard input to build its state which may either be its true initial state or a checkpointed state. The state information contained in standard input should be written as ascii text in a form understood correctly by all machines in the grain's host class. Additional ascii input data for the grain program may directly follow the initial state information in standard input. When writing the checkpoint state information, the user need not be concerned about the unread portion of standard input. Standard input's file pointer position is returned to the scheduler so that it is able to append precisely the unread input to the checkpointed state information in order to form an appropriate input file for grain restart. It is acceptable for grain programs to open and use other read only files, however the files need to be available on any host in the host classes specified for the grain, and the user's checkpoint state information must specify the appropriate positions for the file offsets.

It is mandatory that checkpoint information is written at a location in the program where it is possible to specify a consistent state for the grain computation and file pointers. For checkpointing to be effective, such a location needs to be encountered at least once during the grain's figurative time slice after which the grain server calls the scheduler to report status and checkpoint information. For some programs a single location in a main loop may be satisfactory. For others, there may need to be checkpoint locations in several major modules or loops. Rather than waste effort writing checkpoint information every time such a location is encountered, the grain server signals the grain process with the grain's `d_gexec` specified `notice` minutes advance warning before a checkpoint is needed. The signal is caught by the grain program and the handler sets an unseen static flag to enable checkpointing. At the checkpoint locations, the boolean macro `d_time_to_ckpt` is called to test and reset the flag. The signal handlers for checkpointing and also for monitoring page fault thrashing are installed by calling the `d_ghandler` routine at the beginning of the grain's program code.

The grain's checkpointing code first prepares a status message string which later becomes accessible to the user control program through a query call. It then calls the DCS library routine `d_opencheck` with the message string as a parameter. First, as checkpoint files are written with the same sequence number as their corresponding incremental output files, it is a simple matter to make sure that checkpoint files older than the one with the previous sequence number have been removed. The immediately preceding checkpoint file needs to be saved as it may be still be in use by the server for a grain activity report to the scheduler. The `d_opencheck` call then opens a new checkpoint file with the current sequence number, where it first writes the message string, followed by the grain's rusage information and finally the file pointer offset for standard input. Its final task is to update the access time of the the grain's standard input file so that system cleanscripts don't remove it from `/tmp`. The return value of `d_opencheck` is the open file pointer to the new checkpoint file. It is now the user's responsibility to write out the current state of the grain computation as ascii text to the file. This state information is precisely what is used by the scheduler to form the initial portion of standard input for a restarted grain. The closing library call for the checkpoint routine d_closecheck takes care of several important details. It flushes the new checkpoint file, and the grain's stdout and stderr files and checks for write errors. If write errors have occurred such as caused by a full partition, the d_closecheck performs an immediate exit for the grain with an error status which indicates the write problems to both the grain server and scheduler. Finally, before returning, it increments the sequence number for the stdout and stderr files, and uses *freopen* to open new incremental output files for use until the next checkpoint.

Currently grains only send passive messages (not guaranteed to be read), and only receive blunt active messages in the form of signals for checkpointing, thrash checking, or termination. The messages sent from the grain to the scheduler have a lifetime until the next reported checkpoint and are meant as nonessential, but possibly useful feedback to the user control program. An example showing the details of grain checkpointing is in Appendix A.

This checkpointing technique has worked well in practice. In addition to being portable, the user created ascii checkpoint file is generally much smaller than a binary image based checkpoint, saving space and communication cost, furthering the goal of minimal impact on system resources. In support of this approach, even in lisp which offers the ability to dump running images, local lisp programmers have often chosen to create their own concise checkpoint files rather than dump a large running image.

If a grain is small enough not to warrant checkpointing, then the entire checkpoint support can simply be omitted. The grain's server and the scheduler recognize this situation. The ouput is merely sent once when the grain finishes, and if restart is necessary the grain restarts from the beginning with its original input file.

3.4 Structure of User Control Program

Disruption of the user's control program has no direct effect on the scheduler's obsession to shepherd executing or ready grains through their course to successful completion. If the central scheduler also fails, then currently executing grains send their checkpoints and output to a passive backup until the scheduler resumes service. However, the passive backup takes no action to launch new grains or to move and restart grains from busy or failed hosts. With this framework of system operation in mind, the user has a choice in the design of the control program in order to handle resumption of the grain program after externally caused failure.

The question is that of how to recover the control program's state prior to failure in order to correctly continue execution. The easiest solution for the programmer would be one which did not require special programming effort and yet provided for automatic recovery and rebuilding of the prior state by simply rerunning the control program. This is possible if just a few simple restrictions are obeyed which are reflected in the requirements that the control program always be restarted with the same initial state, and that only the output from finished grains be used to determine control and grain input. In particular, this implies that query information or status messages from grains not be used for control or to determine input. Additionally, random number generators which might be used in generating input data have to initialized with the same seed at the start of the control program. The order and control of grain submissions does not have to be statically fixed in advance, as it is expected to vary according to control decisions may be made on the basis of returned grain output. Assuming that the session has been opened by a d_resume_ses call with the sent_reset flag set, the previously described idempotent behaviour of the d_gexec call along with the grain results being returned in the same order mimics the original execution.

Consequently, a restarted control program effectively retraces its previous execution and continues from the point of interruption.

The control library calls are atomic in the sense that if the control program is interrupted before a call successfully returns, either the scheduler's state will be unchanged or it will reflect the changes dictated by a successfully completed call. In particular, a partially completed `d_gexec` call would be purged by the scheduler if all the parameters and input had not yet been committed to the backing store. Similarly, once the parameters for a `d_gkill` call have been received, it is inevitably carried through to completion even if the scheduler is interrupted.

Consider the following scenario regarding the `d_gkill` call. If the control program is interrupted just before a grain is to be killed with a `d_gkill` call, the reprieved grain might finish and place its output and status in the queue for future `d_gwait` calls before the control program is restarted. In this case the restarted control program retraces its prior execution, finally making the `d_gkill` call, which then would either kill the grain if still running or remove it from the output queue if it has already finished.

A more general approach for successfully restarting the control program requires saving state information at each transaction with the scheduler. On restart, this saved state would reconstruct the computation up to the next scheduled library call. As the next library call might have been done but not recorded or vice versa, it is possible take advantage of the idempotent behaviour of the `d_gexec` call if the parameters are known to be the same. Otherwise, more generally a query call can be made to determine whether the next call had been made and act accordingly. At this point, the control program has recovered fully and can continue. This program approach allows more flexibility in using query information and grain messages to influence grain control and input.

An example of a restartable control program using the simple approach of not using explicitly saved state can be found in appendix B.

3.5 Grain binary placement

Users of DCS are not assumed to have accounts on grain hosts. However, the set of users privileged to use DCS must have have unique login names on the machines permitted to issue user library calls to the scheduler. These login names are used by the scheduler to identify the user's various data structure entries on the scheduler, and on the remote grain servers to determine the user's DCS home directory. When a grain is submitted to the scheduler, its program binary can be specified either by an absolute path, or by a path relative to the user's DCS home directory on the remote server. For grain servers where the user has an account, the special directory might be a symbolic link into a subdirectory of the user's real home directory. If a binary is not found or the special subdirectory is missing, the scheduler sets the host's bit in grain's bad host bit vector kept in the grain's data structure on the scheduler. This prevents the host from being tried again unless the bit is cleared.

Currently there is no support for automatic distribution of grain binaries. It is planned to extend the scheduler's grain host data base to include binary compatibility classes, so that the scheduler could manage a central library of grain binaries. A missing or out of date binary could be replaced at the time of actual grain startup. The grain servers could periodically remove stale binaries, thus reducing the disk storage load on grain servers.

4 The backing store as the basis for reliable recovery

An essential property of DCS is its ability to recover from unexpected host crashes affecting the control program, central scheduler, or grain servers. The central scheduler maintains a backing store which at any time specifies a complete consistent state for DCS operation. A simple but effective protocol for committing data and related state changes to the backing store is used to insure its integrity and consistency. Transactions with control programs and remote grain servers are patterned to promote the central role of the backing store for asserting a consistent state of system operation. The scheduler's backing store thus provides a foundation for recovery after transient failure of the scheduler and any or all of the remote grain servers.

4.1 DCS interaction with the backing store

The total state of DCS can be reconstructed from the backing store's image of asserted changes in system state from control program transactions and the threads of incremental state changes asserted by the activity spawned by these transactions. The general pattern of active control program transactions is as follows: 1) the request with its parameters is passed to the scheduler, 2) this information is committed to the backing store implying an assertion that it will be carried out, and 3) an acknowledgement is made to the control program. As mentioned previously, a control program recovering using its own stored state information may not know whether the last pending transaction was accepted, but it can use a query to check whether it needs to repeat the request before continuing. A control program's query for information is a passive transaction, not requiring a change in state for the scheduler. On the other hand, a successful non-blocking call for grain results does assert an increment in the count of grain results sent.

Transactions between the scheduler and servers produce changes of state in individual grains as their activity reports assert changes in their checkpointed state and and output. Additionally, the grain's running or ready state fluctuates as it follows its possible path of migration. All these asserted changes are comitted to the backing store in a manner which guarantees a correct, consistent state of DCS in the event of random failure.

4.2 Backing store update protocol model

Let us examine the protocol model used to commit the various system activity data and state change assertions to the backing store in a reliable, consistent fashion. The logical relationships between backing store data files and their accompanying update protocol can be modeled simply as a forest of rooted update tree templates with linear branches. Each node of an update tree template represents either a singleton or ordered pair of backing store files and their function at update. The restriction of linear branches means simply that only the root nodes may have more than one child. As described later, files which occur in more than one update tree must occur as a root node file in exactly one update tree.

A single update tree template coordinates a group of closely related files representing an activity or perhaps a class of activity. At any time the current backing store contents of an update tree template correspond to an instance of that template. This instance may be that of a tree which has either been partially prepared for update commitment, committed, undergoing update transformation, or completed update.

Here are the templates for the example DCS update trees:

Figure 2: Update tree templates for DCS

An ordered pair node implies an action to be taken during update transformation between an existing file and an optional temporary file. If the existing file is absent, then the temporary file is renamed as the node's existing file. If the existing file is present, then the template node specifies whether the action for the temporary file is either to replace or append to the existing file. If the temporary file is absent, then no action occurs.

A singleton node may contain either an existing file or a temporary file. In any case, a singleton file provides information relevant in the construction of the temporary root node file. In particular, a singleton existing file represents information which is a fixed part of the entity represented by the nodes of the branch. For a temporary file, the template node specifies whether its update action is to become renamed as the node's existing file, or simply to be discarded. Additionally, either the root node file or its committed temporary can occur in an information bearing singleton node in a different update tree.

The root node always consists of an ordered pair of files with the temporary replacing or creating the existing root file. The update tree is committed for incorporation into the backing store after first each of the update tree's included temporary files has been committed, and then precisely when the temporary root file is itself committed. If the update tree has but a single branch, then commitment of the temporary files on the branch is implied by commitment of the temporary root file.

Just as a single branch may represent the constituent files of a single entity, multiple branches may represent the complete or restricted membership of a class. Figure 2 displays such a hierarchical organization with the update templates of DCS. Further details of the DCS backing store update process beyond the scope of the following discussion are found in later sections.

The complete state of a single DCS grain's computation is represented by an instance of an update tree template containing just a single branch with five nodes plus the root node. The root node for this template is the status file for the grain. The grain's status file holds a time stamp which is changed in the committed temporary status file only if the grain is changing its status between running, ready, finished or killed. The time stamp is unique for all the session's grains as it is derived from a shared counter.

The time stamp for the grain's status file provides a simple mechanism for subsequently defining the session update tree template which has a variable number of branches and the session's status file as the root node. Each branch consists of a single leaf node containing the existing or committed status file of one of the session's grains. These branches correspond to the subclass of the session's grains which have a newer grain time stamp than the grain time stamp kept in the root node's existing session status file. When the root node's temporary file is committed, its grain time stamp value becomes that of the newest grain included in the update. An invariant property of this update tree is that at any time a new session status file can be constructed and committed from the current existing session status file and the implied accumulation of newer branches. This example exposes and encourages the notion of *lazy* commitment of update trees. This helps decrease the amount of disk traffic and allows scheduling of the session status file commitment between other activity. In normal operation, the new committed session status file is trivially constructable from a memory data structure, but the update tree protocol correctly constructs one during recovery when only backing store information is available.

Some user calls or system state change assertions require undelayed commitment of an update tree. With the session status update tree, this situation occurs with a user call to close the session which requires commitment of the altered session status file in order to assert the cleanup of the session's activity from the system.

Another DCS update tree concerns dynamic host statistics and configuration and has for its root node a file containing the dynamic host configuration table. The branches each have a single leaf node with a committed temporary file containing statistics and status information for a single host. Following a similar lazy update commitment of the root node, the leaf node files are discarded.

Finally, the authorization token table update tree is similar in style to the session state tree. The leaf nodes contain existing or committed session status files which have changed status between open and closed after the the authorization tree's root node was last committed. Again, a shared counter is used as a time stamp. Commitment of the root node is forced if a new control program is authorized.

Now let us return to a discussion of the correctness of the update tree protocol. Every file in the backing store is a member of at least one update tree. Each update tree is an instance of some update tree template. Each node in an update tree template is specified as having either a singleton or ordered pair of files along with a designated action as described previously. The only existing or temporary files which the protocol requires for a valid committed update tree to contain is the committed temporary root file. Consequently, a valid instance of an update tree may contain only a root node file, all the other files specified in the template's nodes are optional. Thus update preparation for a tree may involve all the template's nodes, or as few as just the root node alone. Multiple update preparation procedures involving different subsets of an update tree's template's nodes are allowed and typically correspond to distinct types of transactions involving the update tree entity. Only root node files or comitted root node files can occur in more than one update tree, and in secondary occurrences only as an information bearing branch node. Once the update tree's temporary root node file is committed, the update process commences with each branch of the tree being traversed from the leaf to the root with the following rules being applied to each node of the tree's template.

i) A singleton node which is empty or contains either an existing file or the root node file of another update tree is skipped.

ii) If the node contains a singleton temporary file whose action is rename, then it is renamed as the node's existing file.

iii) If the node contains a singleton temporary file used solely as information for the committed root node, then the temporary file is discarded.

iv) An order pair node without its temporary file member is skipped.

v) If an ordered pair node contains a temporary file member whose action is rename, then the corresponding existing file is either created or replaced by renaming the temporary file.

vi) If an ordered pair node contains a temporary file member whose action is append, then the temporary file is appended to the existing file or renamed as the existing file if doesn't exist.

vii) The root node is processed last, and the committed temporary root file is renamed to replace or create the corresponding existing file. After this step, the update process is finished.

The individual operations of the protocol can be reliably accomplished using normal UNIX system calls. The replacement operation is done with the **rename** system call. For the append operation, first the existing target file is truncated to its old length which should be specified in the existing root file. Next, the temporary file is appended to it, and the temporary file is discarded. This approach prevents repeated appends in case of failure during the operation. When the temporary root file for a tree is written to commit the update, it specifies the length files subject to appends will have after the update.

We can track the behaviour of the protocol to provide an informal proof of its success in incorporating a committed update tree to the backing store even if unpredictably interrupted by host failure. If failure occurs during the update process for a committed tree, as a part of system recovery the uncompleted update process is discovered by the existence of a committed root file and is restarted. As the nodes of the template are processed from the leaf toward the root for each branch, the protocol rules imply that nodes which were previously processed are effectively skipped over as null operations. If failure occurs before completion of either a replacement or append operation for a node, then as described above both operations successfully complete when rerun. The remaider of the tree is processed normally and the committed tree is successfully incorporated into the backing store.

Additionally, during the recovery phase after host failure, all uncommitted update trees are checked for broken branches. A broken branch is one which contains an uncommitted temporary file. Such a broken branch should have all its temporary files discarded.

Recall that the protocol rules allow a single template to represent several kinds of updates for a single update tree entity. In the case of the DCS grain update tree, it's beginning update uses only the template's initial input and root nodes. While the grain is running, activity updates involve all the template nodes. Simple status change updates use only the root node. Finally, when the grain finishes, only the output nodes and root node are used. With the possibility of different update preparation patterns for a branch depending on the kind of update, the question arises of how broken branches might be detected following transaction failure or other system failure. For the case of a single branch tree, if failure occurs before the temporary root node file is committed, then the branch is considered broken and the temporary files are removed. Similarly, in an update tree with multiple branches, broken branches can trivially be detected during subsequent update preparation if the following convention is used for preparing branches. The first branch's first temporary file written is not committed until the last temporary file on the branch is committed.

A very important issue is the behaviour of the update tree model when all activity can occur in *parallel*. That is, what happens when preparation for commitment of update trees and additionally the update protocol process for committed trees can all occur in parallel.

First, update trees which represent a single entity, such as the DCS grain update template, are prepared and committed by a single transaction. Simple locking mechanisms can arrange for such transactions to occur in sequential fashion. If such a transaction should fail before completion, perhaps by hanging and timing out, then any broken branches are cleaned up and the lock is released, allowing another pending transaction for the same update tree to proceed.

For update tree templates representing a subclass or class of entities, the branches are independent and branch preparation can occur in parallel. The update commit process is delayed in lazy fashion, and only needs to lock the preparation and commitment of a new root node.

Finally, the question of possible *deadlock* needs to be resolved. As all locking occurs in coincidence with incorporation of a committed update tree into the backing store, the answer depends on a simple analysis of the graph consisting of the forest of update tree templates. First, supplement the forest of tree templates with edges between any template root node and its possible occurrences as a singleton file node in another tree template. Now, topologically sort the supplemented forest of templates and if it is cycle free then deadlock avoidance during the update process is guaranteed. An examination of the example templates for DCS finds that secondary references of root nodes occurs in a hierarchical fashion and that the supplemented forest is cycle free.

In summary, the update tree paradigm provides a simple way to model the backing store data for DCS in a fashion which effectively organizes and binds the logging information directly to the entities involved. Unlike traditional logging methods which can accumulate data associated with multiple transactions involving the same entity, the update tree model bounds the total data stored for each entity to be less than twice its maximal updated size.

5 Scheduler and server daemons

The central scheduler coordinates all system activity, and provides fault tolerant management of grain activity information. It provides and checks for authentication of user directives and acts automatically as a shepherd for grain computations, using checkpoint information to move grain computations from failed or busy hosts.

The scheduler and the servers can write a logfile of system activity. For both the scheduler and servers, there are several levels of logging ranging from a succinct listing of major events, to details of all transactions which may occur. The default logging level for the scheduler and each server can be reset while the system is running by the scheduler administrator or the scheduler host's root while the system is running. The default level for servers is for logging to be turned off.

5.1 Scheduler communication with servers and users

The scheduler spends most of its time listening for the arrival of user directives and remote server activity reports. The communication between scheduler and servers is done with internet domain stream sockets. User calls from designated remote hosts use internet domain stream sockets and calls from the scheduler's host use Unix domain stream sockets. The scheduler uses distinct internet domain ports to listen for server calls and user calls. The scheduler is thus able to conveniently alternate between server and user calls in a fashion weighted toward server calls.

In any case, if communication with a server or user control program becomes hung or a transaction is taking too long, the connection is broken and after any required cleanup occurs the scheduler proceeds to its next activity. Broken connections have occurred occasionally, usually with a remote server many hops and thousands of miles away. In this situation the server would be marked by the scheduler as delinquent, but could restore itself to good standing with a successful conversation on its next normally occurring status report. However, depending on the scheduler's configuration values for the remote server host, the server's may be considered failed before a later call and its grains moved or put on the ready queue. This is explained further in the sections on host configuration and servers.

5.2 Session data management

When a new session is opened, the scheduler creates a subdirectory specifically to hold data for that session. Some of the scheduler's transient information is kept only in memory, some state information is stored in both memory and disk, and some bulky items like grain input, output and checkpointed state information are maintained only on disk. The disk information not only provides data support for grain migration and user queries, but also provides a consistent data base for scheduler and system recovery following unexpected failure and reboot of the scheduler's host.

5.2.1 Global session data

Much of the global session data held in memory data structures is implicit in the accumulated grain data kept on disk. Some of the following structure data is redundant, but for convenience the following in memory data structure is written to disk in steps intertwined with the soon to be described protocol for reliable update of disk grain data.

```
struct session_info {
    char username[10];  /* the user name and session number pair */
    int session_num;    /*            identify this session        */
    int auth_index;     /* index to session's authentication table entry */
    int block_flag;     /* control program blocked waiting for grain results ? */
    int control_addr;   /* 0 if unix domain, otherwise hex internet addr */
    short control_port; /* if internet, then port number of listen socket */
    int active_grain_count;  /* currently executing grains */
    int ready_grain_count;   /* on ready queue */
    int completed_count;     /* finished grains */
    int grains_sent;         /* how many finished grain results sent */
    int fin_vector[8];       /* bit vector showing finished grains */
    char reslist[GRAINS_MAX];  /* list of grains in order finished */
};
```

The only new essential data found here which is needed when the scheduler restarts is the ordered history list of grain finishes and the blocking information. The first **completed_count** entries of the **reslist** array contain the history list. If the control program is blocked waiting for a grain result from this session, then if the **control_addr** field is zero then connection is made through a known unix domain socket, otherwise the connection is made to the internet address and port specified in the structure. As implied in the structure, grain numbers are restricted to the range 0...255.

If more grains are required, it is recommended to decompose the problem into multiple sessions, possibly run by the same control program.

Just as the scheduler spends most of its time listening, control programs spend most of their time blocked listening for grain results. If the control program is running on a different host from the scheduler and is blocked listening, then there is a good chance that the restarted scheduler can continue its interaction with the control program without error. The session_info structure is written to disk as a part of the update protocol for a grain activity report involving a finished grain. Similarly, it is also written to disk after a control program makes a d_gwait call indicating that it is blocking to wait for grain results.

5.2.2 Individual grain data

Each grain has separate files for holding the following activity information items: the original submission input preceded by a header and the exec parameters, status information, most recent message, most recent checkpoint, standard output and standard error data. The input file consists of the initial state specification followed by any additional input. The message and checkpoint files are replaced each time the remote host calls with an update of the grain's activity. The grain's standard output and error files additionally are updated incrementally in synchronization with the grain's checkpointed state. The content of the grain's status file is described by the following structure.

```
struct grain_info {
    int grain_number;  /* the assigned number of this grain */
    int host_class;    /* the logical or of host types this grain can run on */
    int nogo_vector[8];  /* hosts tried, but can't run on, missing binary etc */
    int fail_kntr;     /* more than two non-innocent failures, grain is killed */
    int restart_num;   /* 0 for initial startup, incremented at grain restart */
    int param_length;  /* length of grain's identifying triple, argv, and env */
    int input_ptr_pos; /* ptr to remaining input, computed at checkpoint update */
    int restart_ckpt_len;  /* length of the checkpoint used for current restart */
    int sequence_num;  /* current sequence number of received stdout and stderr */
    int output_len;    /* current length of grain's output */
    int error_len;     /* current length of grain's standard error */
    int grain_status;  /* running, ready, finished, or killed */
    int current_host;  /* if running, index in host list, else -1 */
    int last_contact;  /* time of last report to scheduler */
    int last_ckpt_time;  /* time on server checkpoint was created */
    struct rusage   current_restart_rusage, total_rusage;
};
```

At the time a grain is submitted, the status information file is created with a grain_status field value of ready, and zero values for all the other fields. The message, checkpoint, standard output and error files are created, each with zero length. When the grain is first started on a remote host, the status file is rewritten with a running grain_status and an appropriate current_host field, but the other fields are not changed. Later, the status file is rewritten at the time of grain activity updates, or when the grain's operating status changes between running, ready, finished or killed. The other grain files are changed only by grain activity updates made when the grain's server reports to the scheduler.

Care is taken to maintain the consistency and integrity of the scheduler's disk data. A simple but effective protocol is used to avoid loss or corruption of session grain data resulting from scheduler or server host failure during a grain activity update or when the operating status of the grain changes.

The update process consists of creating the appropriate temporary files containing new information, committing the update, and then altering the actual grain files. Once the update has been committed, the update process proceeds to completion, even if rudely interrupted by scheduler host failure. In case of host failure, the normal scheduler recovery process upon reboot finishes the committed update. If failure occurs before the update has been committed, then the partial update information is discarded during recovery. The actual changes which occur from an update involve changing fields in the grain status information file, replacing the message and checkpoint files, and appending incremental output to the grain's standard out and error files.

In detail, the grain activity update events occur in the following order. The message, checkpoint, and incremental output and error are each written to temporary files. Values for a new status information file are computed. The fields restart_num, restart_ckpt_len, grain_status, and current_host remain the same. The input_ptr_pos field is a byte offset value in the original submission input and is used as a pointer to unread input. The grain server returns the current byte offset in the grain's input file. Thus, if no restart has occurred, i.e. restart_num equals zero, then the new input_ptr_pos is simply the current byte offset. Otherwise, the input_ptr_pos is the sum of the old input_ptr_pos plus the current byte offset minus the restart_ckpt_len. When the server sends the incremental output and error, it also sends the contiguous range of sequence numbers involved by sending the first and last numbers. These will usually be the same number unless the server has been unable to call the scheduler or the passive backup for more than one update cycle, causing incremental output files to stack up. The sequence numbers are used to insure no gaps occur in output. Sequence number gaps are considered an error causing restart using the current consistent checkpointed state, unless the gaps are determined to have had zero length by comparing the total length values maintained on both the server and scheduler. Assuming no error, the the sequence_num field is given the new high value. The fields output_len and error_len are given values corresponding to their new total lengths. Similarly the current_restart_rusage and total_rusage fields have the update's incremental rusage amounts added to their component fields. Thus the current_restart_rusage and total_rusage reflect the total rusage by the current grain process and by the grain through its complete migration path respectively.

After the temporary grain status file is written, the update is committed by using the system call **rename** to add the extension ".commit" to its name. If the scheduler host crashes, on recovery the scheduler recognizes the committed status file and finishes what might be left of the following steps. The message and checkpoint files are replaced by renaming their temporary files. The appends to the standard output and then to the standard error files are each done with the following three steps: 1) if the incremental file still exists, truncate the main file to the old length specified in the old status file, 2) append the incremental file to the end of the main file, and 3) delete the incremental file. Finally, the status information file is replaced by renaming its committed temporary file.

During scheduler recovery after host failure, the same steps are followed when a committed status information file is encountered. However, if temporary files are missing, the related operation is not attempted. Thus, in the alternate case of a grain operating status change, where only the status information file is written, the same recovery procedure still works correctly. As temporary files without a committed status file are removed on recovery, there are no stale temporary files left around to corrupt a later simple status change update where only a committed status file had been created.

If the grain is late making a checkpoint, or is not doing them at all, the server simply informs the scheduler that no activity update information is available. In this case, the scheduler does no rewriting of the grain's activity files.

When the grain finishes, the final update of its activity information is done slightly differently. The same commit protocol is followed, however no temporary files are written for the message or checkpoint files. The temporary files for incremental output and error are written. The temporary status information file is written and committed, but its contents are determined by slightly different rules. The grain_status field becomes finished and the rusage fields are updated. However, the other fields remain unchanged, keeping the values consistent with the last checkpoint update. Thus the length fields in the status information file may not agree with the actual length of the finished output and error files.

When the scheduler changes a grain's state from ready to running, only the status information file is rewritten with the following changes. The restart_num field is incremented. The restart_ckpt_len field is assigned the length of the current checkpoint file. The grain_status is changed to running. The current_host is assigned the internet address of the grain's host.

The scheduler also maintains relevant parts of the grain's activity information in data structures in memory.

5.3 Scheduler passive backup and restart

When making their grain activity update calls to the scheduler, the grain servers use an exponential backoff approach in their attempts to connect with the central scheduler. It has been observed that the central scheduler has never been so busy as to force a remote server to make even a second call attempt. If a fixed small number of attempts are unsuccessful, then the scheduler is assumed down and a passive backup is called in its place. The passive backup collects checkpoint and status information until the central scheduler resumes operation. If the remote server is unable to reach either the central scheduler or the passive backup, it continues grain processing, appropriately updating checkpoint information and accumulating the sequenced incremental output files until the server can make a successful call to deliver its activity update. When the central scheduler restarts, it first reads from disk its own checkpoint and status information for each session, rebuilding the grain ready queues, grain running lists, and other data structures. It then calls the passive backup to retrieve the information gathered by the passive backup, updating its memory data structures and data stored on disk.

The local system has run for months, recovering successfully from many scheduled downtimes and several unexpected host crashes. The scheduler should reside on a very reliable host, both in terms of hardware and software. The local host for the scheduler has been an HP 9000-320 running the Utah port of 4.3 BSD. This choice has provided extremely reliable service.

5.4 Scheduler queue management and grain selection

As mentioned previously, DCS allows multiple users and sessions. Each session has its own queue for ready grains and list for executing grains. All grains of a particular session have priority over grains of sessions created later.

The remote grain servers make periodic calls to the central scheduler according to their designated time slice to report status and activity. If a server has an available slot for grain execution, the session grain ready queues are checked according to session seniority to find a ready grain compatible with the grain server's host class. Session ready queues belonging to users not permitted to use the server are skipped. This is determined by configuration information kept on the scheduler. If an acceptable grain is found, the scheduler attempts to start the grain on the remote server. If the attempt fails perhaps because of a missing grain binary, the scheduler continues its search until it has checked all the grains in the ready queues. If the server has more than one slot available, the process is continued. In a startup attempt failed because of a missing binary, the server's bit position in the grain's **nogo_bitvector** is set. This prevents the grain from trying this host again until its bit is cleared.

The scheduler maintains individual lists of active hosts for each host type. When a user submits a new grain request, the list for the smallest numerical valued host type in the host_class is checked first for a host with an available grain slot. If a grain slot is not found, then the list for next higher valued host type in the grain's host_class is checked until all the acceptable host type lists are checked. If a matching host with an available grain slot is found, then first the scheduler's host configuration information is checked to make sure the user is permitted to use the server. If the user is allowed, then remote startup is attempted, otherwise the host search continues. If the startup fails, then the grain's nogo_vector bit for that host is set and the host search is continued. Finally, if no empty slot on an acceptable host is available, the grain is added to the end of its session's ready queue. If the grain request had been designated urgent, instead it is pushed onto the front of the queue. Note that lower priority grains which might be running on hosts matching the new submission's host class are not affected at this time.

Lower priority grains running on a server are usurped only at a time slice boundary when the server calls in to report status and activity. When a server calls in, first any free slots the server might have are filled according to the process previously described. After the free slots are filled, then the the search is continued through the remaining ready grains which are on ready queues of higher seniority sessions than the session of one or more executing grains. If an acceptable higher priority grain is found, a tentative startup is attempted which checks to make sure that the new grain can run on the server host before the lower priority grain is suspended and removed.

Grains which are suspended from busy hosts or whose hosts have failed are placed on the front of their session's ready queue if an acceptable alternate host is not immediately available. This would be the case for a grain removed from a server because of excessive page faults. On the other hand, if a grain is suffers failure during execution three times, then it is killed. The failure count is kept by the **fail_kntr** field in the grain's status structure.

5.5 Valid users information

As briefly mentioned previously, user calls to the DCS scheduler can only be made from source machines designated by the scheduler. Likewise, on such a machine the actual users allowed to make calls are limited to those listed as members in a special DCS group in the /etc/group file. Users in DCS are identified by login name, not by uid. DCS users need not have accounts on all the designated machines, and are not allowed to use accounts on machines which conflict with another users identifying login name. This way, the login name can be used as a primary key for session and grain activity on both the scheduler and remote grain servers. Possible bogus users are avoided by including only login names corresponding to the valid user associated with that login name in each source machine's DCS group list.

For each grain server host the scheduler has a list of users whose grains are permitted to run on that server.

For grain servers which do not run as root, the grain processes run under the server owner's uid. If the grain server is run by root, then an additional list kept by the scheduler for each grain server tells whether the user's grain process can and should be run setuid to the account with the user's login name. If the account with user's identifying key login name belongs to someone else or the user has no account, then the grain should be run under an anonymous uid. Currently there is no provision for mapping the session user name to different user names on grain servers.

The described unique login names of valid DCS users are kept in a DCS configuration file. This file is read on scheduler startup and reread when the scheduler is directed to update the user information. The user names are put into a hash table and each assigned a unique integer id which is used as a bit vector index. Using this hash function, two files, containing lists of permitted users for each host and users permitted to run as themselves for each host, are read or reread to build bit vectors for each host containing this user information. These bit vectors are in fact fields in each host's configuration structure.

/subsectionAuthentication for user calls

All normal user directives received by the scheduler explicitly or implicitly specify a user name, session number, and authentication token. The corresponding session entry on the scheduler has an index into the authentication table. If the authentication token passed by a call matches the one found in this table entry, then the directive is accepted.

```
struct authentication_entry {
    long auth_token;    /* the authorization token */
    char username[9];  /* the user's login name as a string */
    char cp_ident[10]; /* control program identifier string */
    int session_knt;    /* count of active sessions using this token */
    long time_stamp;    /* zero, except when session_knt zero, the stamped */
};
```

When a user sends a directive to open a brand new session, the authentication table is searched for an authentication token matching the token and user name sent. If the entry is found then its position in the table is assigned to the **auth_index** field in the new session's data structure. Otherwise, the request is rejected. If the user sends a directive to reopen an existing session, and if the existing session's token entry cp_ident field matches the new token's cp_ident field, then the old entry's session_knt is decremented and the session's auth_index field is changed to point to the new token entry and it's active session count is incremented. When a session is closed, the corresponding tokens session's count is decremented.

5.6 Server configuration information

All potential grain hosts have an entry in the grain host configuration table on the scheduler. The table is built from a grain host configuration file which is read when the scheduler starts up, or when directed to rebuild the configuration table. The identifying key for the host table entries is the 32 bit internet address associated with each host. This is the structure representing a grain server configuration entry.

```
struct server_config {
        unsigned long inet_addr;  /* server internet address in hex */
        char name[NAMELEN];       /* grain server's official host name */
        unsigned hostclass;       /* bit mask value representing the hostclass */
        int time_zone_offset;     /* time_zone offset in minutes from scheduler */
        int grain_limit;          /* number of grains host can run simultaneously */
        int call_in_interval;     /* call in period or time slice for server grains */
        int call_in_alert;        /* server delinquent if not heard from for this long */
        int call_in_limit;        /* server considered dead, grains moved */
        unsigned long mach; /* internet address of machine to check, else zero */
        unsigned long valid_users, setuid_users; /* bit vectors for access etc */
};
```

When hosts are either added or deleted from a running.

Grain server host information can be set or reset using the **d_sethost** library call. This call can be used to reset the job limit count of a reformed grain server which misbehaved and had its job limit count set to zero. Additionally, this call can even reset the bit entry in the server host avoidance bit vector contained in a grain's data structure in memory on the scheduler. This is useful after a problem with a remote grain server has been repaired, but a grain waiting on the ready queue still thinks it should avoid the previously delinquent host. This typically occurs when the installation of a new grain server was botched.

6 Grain Servers

The central scheduler maintains a database of candidate grain server configuration information which includes for each host: its name, internet address, host class, number of simultaneous grains allowed, call in interval, time out limit, extended time out limit, file server host (if any), and gateway (if any). When a grain server host calls in for the first time, it is rejected if it is not in the database, otherwise its call in interval and grain count limit are sent to it, and its host class and other scheduler grain host list attributes are set from the database. Typically, only multiproccessor hosts are allowed to run more than one grain simultaneously. The grain servers call the central scheduler each "call in interval" to report possible grain status and checkpoint information and to maintain their host active status. If a grain server host which becomes overdue is running grains, then a child of the scheduler will assess whether the host and grain server are still up. The grain server host, file server and gateway will be checked to see if they are active. If it is determined possible for the host to be active, then it can be delinquent "extended time out period" time before its grain(s) are placed on the ready queue for possible reassignment to other grain servers. If the missing host shows up with the grain still running, and the grain request is still on the ready queue, then all is forgiven and it is reinstated on the exec list. If the grain has been reassigned to a new host, and later the missing host shows up, then the copy of the grain which has less total running time is killed. This feature helps prevent needless shifting of grains and lost effort because of gateway maintenance or temporary problems.

Grain servers use a child process to call the central scheduler in order to remain ready to receive calls from the scheduler. The grain servers call in with a period which effectively becomes the time slice quantum of grain computations. Premature calls to the scheduler are made if a grain changes state by terminating, or a "thrash" job being suspended by the server. The return code and termination signal are reported to the scheduler host so it can determine whether the grain terminated normally. As programs are assumed to have been thoroughly debugged, errors have usually indicated system errors. A failed grain is first retried on the same machine. If it continues to fail, the host is effectively marked as down and the grain is attempted on another host. If the grain fails again, it is terminated and reported as failing. Interestingly, some obscure bad disk block errors have been discovered by a failed grain which was successfully restarted and completed on the same machine. Unobtrusive, courteous use of grain server hosts is a very important goal of this system. For grains running in the small image, little paging, compute bound category, the rusage information returned to the scheduler is used to determine whether the grain should be moved because it has been shut out by heavy usage. For "thrash" category grains, immediate suspension and migration occurs if grain server system load is perceived to be too high. Idle grain servers also report system load status information with their periodic calls to the scheduler.

The grain server can run as root or under a user uid. When running as root, grains are run under the user's uid if the user has an account, or a nobody style uid otherwise. When running as a nobody, the binaries need to have been placed in the users subdirectory of the grain servers directory for user binaries. When the server is running under a user's uid, all grains run under that user's uid. This feature is convenient for grain servers residing on remote machines where it is not possible or practical to run the server as root.

Temporary files for collecting output reside in a subdirectory in the grain server's home directory. The file names contain the host name, session number, and grain number in order to avoid any problems with grain server hosts which share file systems.

7 Security

The scheduler runs as root using privileged sockets. This gives assurance to grain servers that they are receiving requests from a root process on the scheduler host which should be a secure machine. Requests for grain activity come only either from user control programs, or on restart from secure checkpoint files. The user control programs use Unix domain sockets and therefore run on the same host as the central scheduler. A list of allowable users is maintained used by the scheduler. Calls from grain servers on machines not listed in the scheduler's host configuration information are not accepted. Grain server's cannot initiate grain requests, only accept grain execution requests from the scheduler. The grain servers can run as root and use privileged, but as the server sometimes has a rather tenuous guest status on machines, it is useful to allow it to run under an ordinary user uid. The scheduler handles servers of either type.

References

ALL87 Allman, E., "UNIX: The Data Forms," Proceedings of the Winter 1987 Usenix Conference, Usenix Association, January 1987.

ALM86 Almes, G. T. "The Impact of Language and System on Remote Procedure Call Design" Proceedings of the 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, 1986, 414-421.

ALO88 Alonso, R., and Cova, L. L., "Sharing Jobs Among Independently Owned Processors" Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, California, 1988, 282-287.

BAB84 Babb. R. G., II, "Parallel Processing with Large-Grain Data Flow Techniques" Computer, July 1984, 55-61.

BIR85 Birman, K. "Replication and Fault-Tolerance in the ISIS System" Proceedings of the 10th ACM Symposium on Operating Systems Principles, Orcas Island, Washington, 1985, 87-96

BIR87 Birman, K. and Joseph, T. "Exploiting Virtual Synchrony in Distributed Systems" Proceedings of the 10th ACM Symposium on Operating Systems Principles, Austin, Texas, 1987, 123-138.

BRI88 Brillhart, J., Lehmer, D. H., Selfridge, J. L., Tuckerman, B. and Wagstaff, Jr., S. S., *Factorizations of* $b^n \pm 1, b = 2, 3, 5, 6, 7, 10, 11, 12,$ *up to high powers, Second Edition*, American Mathematical Society, R.I., 1988.

CAB86 Cabrera, L., Sechrest, S., and Caceres, R., "The Administration of Distributed Computations in a Networked Environment, An Interim Report" Proceedings of the 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, 1986, 389-397.

CAR86 Carriero, N. and Gelernter, D. "The S/Net's Linda Kernel" ACM Transactions on Computing, Vol 4, No 2, 1986, 110-129.

CHE86 Cheriton, D. "Problem-oriented Shared Memory: A Decentralized Approach to Distributed System Design" Proceedings of the 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, 1986, 190-197.

DOU87 Douglis, F. and Ousterhout, J. "Process Migration in the Sprite Operating System" Proceedings of the 7th International Conference on Distributed Computing Systems, Berlin, West Germany, 1987, 18-25.

FIT86 Fitzgerald, R. and Rashid, R., "The Integration of Virtual Memory Management and Interprocess Communication in Accent". ACM Transactions on Computing, Vol 4, No.2, 1986, 147-177.

HAG86 Hagmann, R. "Process Server: Sharing Processing Power in a Workstation Environment" Proceedings of the 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, 1986, 260-267.

HWA84 Hwang, K. and Briggs, F., *Computer Architecture and Parallel Processing,* McGraw-Hill, New York, 1984.

JOS86 Joseph, T. and Birman K., "Low Cost Management of Replicatied Data in Fault-Tolerant Distributed Systems" ACM Transactions on Computer Systems, Vol. 4, No. 1, 1986, 54-70.

KIN89 Kingsbury, B.A. and Kline, J.T., "Job and Process Recovery in a Unix-based Operating System," Proceedings of the Winter 1989 Usenix Conference, Usenix Association, January 1989.

KRU88 Krueger, P., and Livny, M., "A Comparison of Preemptive and Non-Preemptive Load Distributing" Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, California, 1988, 123-130.

LIT87 Litzkow, M. J., "Remote Unix - Turning Idle Workstations into Cycle Servers," Proceedings of the Summer 1987 Usenix Conference, Usenix Association, June 1987.

LIT88 Litzkow, M., Livny, M., and Mutka, M., "Condor - A Hunter of Idle Workstations" Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, California, 1988, 104-111.

LOH88 Lohr, K., Muller, J., and Nentwig, L., "DAPHNE: Support for Distributed Applications Programming in Heterogeneous Computer Networks" Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, California, 1988, 63-71.

MUT87 Mutka, M. W. and Livny M., "Scheduling Remote Processing Capacity In A Workstation-Processor Bank Network" Proceedings of the 7th International Conference on Distributed Computing Systems, Berlin, West Germany, 1987, 2-9.

NIC87 Nichols, D. "Using Idle Workstations in a Shared Computing Environment" Proceedings of the 10th ACM Symposium on Operating Systems Principles, Austin, Texas, 1987, 5-12.

REI85 Reinhardt, S. "A Data-Flow Approach to Multitasking on Cray X-MP Computers" Proceedings of the 10th ACM Symposium on Operating Systems Principles, Orcas Island, Washington, 1985, 107-114.

RUD84 Rudd, W. G., Buell, D. A., and Chiarulli, D. M., "A High Performance Factoring Machine" Proceedings of the 11th International Symposium on Computer Architecture, Ann Arbor, Michagan, 1984, 297-300.

SCH86 Schantz, R., Thomas, R., and Girome, B., "The Architecture of the Cronus Distributed Operation System" Proceedings of the 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, 1986, 250-259.

SMI89 Smith, J. M., and Ioannidis, J., "Implementing remote fork() with checkpoint/restart" *Operating Systems Technical Committee Newsletter*, **3**,1 (1989), 15-19.

SOU86 Souza, R. J. and Miller S. P. "Unix and Remoter Procedure Calls: A Peaceful Coexistence?" Proceedings of the 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, 1986, 268-277.

THE85 Theimer, M., Lantz, K., and Cheriton, D. "Preemptable Remote Execution Facilities for the V-System" Proceedings of the 10th ACM Symposium on Operating Systems Principles, Orcas Island, Washington, 1985, 2-12.

THE88 Theimer M., and Lantz, K., "Finding Idle Machines in a Workstation-based Distributed System" Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, California, 1988, 112-122.

YAP88 Yap, K. S., Jalote P., and Tripathi, S., "Fault Tolerant Remote Procedure Call" Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, California, 1988, 48-54.

YOU87 Young, M. et al "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System" Proceedings of the 10th ACM Symposium on Operating Systems Principles, Austin, Texas, 1987, 63-76.

ZAY87 Zayas, E. "Attacking the Process Migration Bottleneck" Proceedings of the 10th ACM Symposium on Operating Systems Principles, Austin, Texas, 1987, 13-24.

APPENDIX A: Grain checkpointing example

from DCS library source files

```
static int  ckpt_flag = 0;

SIG_TYPE d_catch_sig()     /* checkpoint signal handler, sets flag */
{  ckpt_flag = 1;}

void d_ghandler()       /* install signal handlers, initialize static vars */
{ signal(CKPT_SIG, d_catch_sig()); ... }

int d_time_to_ckpt();  /* query function to test and reset checkpoint flag */
{  if (ckpt_flag){
     ckpt_flag = 0;  return(1);}
   return(0);
}
FILE *d_opencheck(message)  /* returns file pointer to new checkpoint file */
char *message;               /* handles message, status, and stdin position */
{ ... };

void d_closecheck();        /* checks for file errors, such as partition full */
{ ... };
```

Grain main program file

```
#include "grainlib.h" /* declarations of grain library functions */
/* includes stdio.h */
main(argc,argv)
{
  FILE *ckfpb, *fp;  /* fp, for example of checkpointing read only file */
  char *message, o1[256], o2[256], o3[256], *outhex();
  int powersize, pkntr, pkntlim2, numcurves, i;
  MULTIPRECISION n,x[],y[],z[];  /* large integer type, outhex converts to hex string */
  ...
  /* set up signal handler for checkpointing */
  d_ghandler();
  ...
  /* read and create initial program state from standard input */
  ...
  /* checkpoint at location where consistent program state can be specified */
  if ( d_time_to_ckpt() ){
    message = "message available to the user's control program";
    ckfpb = d_opencheck(message);  /* takes care of message, stdin, status */
    /* write the grain's current state information to ckfpb */
    fprintf(ckfpb, "%s\n%x\n%ld\n", outhex(n,o1), powersize, ftell(fp));
    fprintf(ckfpb, "%d\n%d\n%d\n", pkntr, pkntlim2, numcurves);
    for(i=0;  i < numcurves;  i++){
      fprintf(ckfpb, "%s\n%s\n%s\n", outhex(x[i],o1),outhex(y[i],o2),outhex(a[i],o3));}
    d_closecheck(ckfpb); /* checks for file errors and renames checkpoint */
  }
  ...
}
```

APPENDIX B: Simple control program example

This sample program uses the simple "don't care" approach for restarting control programs which takes advantage of the optional idempotent style behaviour of the d_resume_ses, d_close_ses, and d_gexec calls. Additionally the d_gwait call returns grain output in the same order as prior disrupted control program executions. The d_resume_ses call returns without error for either terminated or active sessions if the control program name strings match. If the session doesn't exist, the session is created and the call returns without error. Additionally, for active sessions, with sent_reset parameter set to TRUE, all grain results are resent, allowing a simple tracing of activity to regain the state of the session computation. The d_gexec call becomes a successful null operation if the sesssion has been "opened" with d_resume_ses, and is identical in all parameter values to a previous call. These properties are sufficient to allow successful restart of a program which doesn't use grain messages and has fixed initial state.

This example control program is invoked by the following command line.

```
        dcs_exec find_factor < composite_big_integer_list
```

Source listing of example control program find_factor.c

```
#include "dcs.h"
/*****  here are the grain host class definitions from dcs.h
    #define UNKNOWN 0X0
    #define VAX     0X00000001  /* campus 750's and micro vax II's */
    #define SEQ     0X00000002  /* department Sequent Balance */
    #define RTPC    0X00000004  /* campus RT's */
    #define HP320   0X00000008  /* department HP 9000-320, also the scheduler */
    #define UTEKA   0X00000040  /* department Tektronix utek apple servers */
    #define UTEKC   0X00000080  /* department Tektronix utek lab clients */
    #define UTEKS   0X00000100  /* department Tektronix utek lab file server */
    #define SUN35   0X00000200  /* campus Sun 3/50's */
    #define SUN36   0X00000400  /* campus Sun 3/60's */
    #define SUN37   0X00000800  /* campus Sun 3/260's, 3/280's */
    #define SUN46   0X00001000  /* campus Sun 4/280's */
    #define UTEKE   0X00002000  /* Tektronix uteks in ECE */
    #define ZAPPY   0X00004000  /* Sun 3/50 off campus */
    #define NEXTE   0X00008000  /* NEXT machines in ECE */
    #define NEXTC   0X00010000  /* department NEXT machines */
*****/
#include "sdefines.h"
#include <signal.h>
#include <stdio.h>
#include <strings.h>

#define LENPROG  "ellipsc"
#define POWERSIZE 0x1000
#define PKNTLIM2 50000
#define SEEDVALUE 596561004
#define SIZE 256
static char state[SIZE];
extern char  *outhex();
```

```
int  session;
long time(), tloc;
char *ctime();
int onintr();

makecurves(num, powersize, pkntlim, numcurves, buffer)
    int  powersize, pkntlim, numcurves;
    char *num, *buffer;
{
  NUMSTR  o1, o2, o3, o4, o5, o6;
  MULTIPRECISION  x, y, n, a, b, d;
  int  i, string_index;

  inhex(n,num);
  sprintf(buffer, "%s\n%x\n%d\n%d\n%d\n",
          num, powersize, 0, pkntlim, numcurves);
  string_index = strlen(buffer);

  for(i=0;  i < numcurves;  i++){
    selectcurve(x,y,a,n,b,d);
    printf("curve %d\nx=%s\ny=%s\na=%s\nn=%s\nb=%s\nd=%s\n",
           i+1,outhex(x,o1),outhex(y,o2),
           outhex(a,o3),outhex(n,o4),outhex(b,o5),outhex(d,o6));
    sprintf(&buffer[string_index], "%s\n%s\n%s\n",
           outhex(x,o1), outhex(y,o2), outhex(a,o3));
    string_index += strlen(&buffer[string_index]);}
}


main(argc,argv)
     int  argc;
     char  **argv;

{
  char *arg_vector[5], *env_vector[5], *output, *error;
  int  count, grain_number, grain_termsig, grain_retcode, status, trys;
  int  badknt, grains_completed, grains_sent;
  char filename[40];
  int  smallcurves, mediumcurves, largecurves;
  int  lim_utek, lim_seq, lim_sun35, lim_sun36, lim_sun37, grain_num;
  char num[200], bigbuffer[MAXSTRING];

  /* get authorization token and pass this program's name "ellips1" to scheduler */
  d_get_auth("ellips1");

  smallcurves = 3;    mediumcurves = 4;    largecurves = 5;
  lim_utek = 15;    lim_sun35 = 10;    lim_sun36 = 4;
  lim_sun37 = 3;    lim_seq = 10;
  initstate(seedvalue,state,SIZE);
  session=0;

  while ( scanf("%s",num) != EOF ){
    session++;
    if ((status = d_resume_ses(session, TRUE)) != NO_ERROR ){
        fprintf(stderr, "Error initializing session %d - status = %d,  %s",
                session, errno, ctime(&tloc));
        exit(1);}

      arg_vector[0] = LENPROG;
      arg_vector[1] = NULL;
      env_vector[0] = NULL;
      for (grain_num=0;  grain_num<lim_seq;  grain_num++){
        makecurves(num, POWERSIZE, PKNTLIM2, smallcurves, bigbuffer);
        status=d_gexec(session,grain_num,arg_vector,env_vector,bigbuffer,
                SEQ,0,0,5);}
      for (grain_num=0;  grain_num<lim_utek+lim_sun35;  grain_num++){
        makecurves(num, POWERSIZE, PKNTLIM2, smallcurves, bigbuffer);
        status=d_gexec(session,grain_num,arg_vector,env_vector,bigbuffer,
        UTEKE|UTEKC|SUN35, 0,0,5);}
      for (grain_num=0;  grain_num<lim_sun36;  grain_num++){
        makecurves(num, POWERSIZE, PKNTLIM2, mediumcurves, bigbuffer);
        status=d_gexec(session,grain_num,arg_vector,env_vector,bigbuffer,
        SUN36|NEXTE|NEXTC, 0,0,5);}
      for (grain_num=0;  grain_num<lim_sun37;  grain_num++){
        makecurves(num, POWERSIZE, PKNTLIM2, largecurves, bigbuffer);
        status=d_gexec(session,grain_num,arg_vector,env_vector,bigbuffer,
        SUN37, 0,0,5);}

      if ( (status = d_gwait(BLOCK, session, &grain_number, &grains_completed,
      &grains_sent, &grain_termsig, &grain_retcode,
      &g_rusage, &grain_outlen, &grain_errlen,
                              &output, &error)) == NOERROR ){
        printf("Grain %d, session %d, sig %d, retcode %d, comp %d, sent %d\n",
              grain_number, session, grain_termsig, grain_retcode,
       grains_completed, grains_sent);
        printf("output -\n%s\nstandard error -\n%s\n", output, error);}
      else
      {
        /*  Otherwise, if the system is down, terminate the program.  */
        if (errno == ECONNREFUSED)
          exit(ERROR);
      }

      d_close_ses(session);
    }
  }
```