

# **A Checkpoint and Recovery System for the Pittsburgh Supercomputing Center Terascale Computing System**

Nathan Stone <nstone@psc.edu>, John Kochmar <kochmar@psc.edu>, Raghurama Reddy <rreddy@psc.edu>, J. Ray Scott <scott@psc.edu>, Jason Sommerfield <jasons@psc.edu>, Chad Vizino <vizino@psc.edu>  
Pittsburgh Supercomputing Center, Pittsburgh, PA 15213  
Phone: +1 412 268-4960, Fax: +1 412 268-5832

## **Abstract**

The Pittsburgh Supercomputing Center (PSC) has designed, implemented and deployed a user-level checkpoint and recovery system for the Terascale Computing System (TCS), a Compaq AlphaServer SC platform managed by the open version of the Portable Batch Scheduler (OpenPBS). This checkpoint system allows for the automated recovery of jobs following both machine failures and scheduled maintenance periods. As an added feature, this system allows that any time lost by the user process because of machine failure between the time of the failure and the time of the last checkpoint can be automatically credited back to their allocation. This system can be used to maximize resource utilization in time periods approaching scheduled maintenance. Thus, it improves both the usability and administration of the TCS. It is accessible to C, C++ and Fortran-based jobs and is implemented in a relatively portable manner. This article describes its design goals, logical operations, and implementation details.

## **Overview**

TCS, like many of today's terascale clusters, consists of hundreds of interdependent nodes assembled from commercial, off-the-shelf (and otherwise independent) components. One of the dominant design goals of this installation is to support single parallel jobs that utilize the entire machine. As such, the probability that any node will fail for any reason, abnormally terminating a user's job, is sufficiently high that the mean time to failure of the machine as a whole can be shorter than typical run times for these large jobs. This environment mandates that system architects address automated job recovery. Some platforms (*e.g.* Cray Unicos, SGI IRIX, NEC SX) provide a system-level checkpoint feature principally to facilitate job interruption and migration or to protect against CPU failure. In this article we present original work whose principle goal is to address not merely CPU failure but the potential failure of hundreds of CPUs, file systems, networks, and independent operating systems. PSC has created and deployed a user-level checkpoint infrastructure that addresses these issues and facilitates automated recovery of users' jobs through interactions with the batch scheduling system.

This system was designed with three primary requirements: job recoverability, minimum effect on job running time and scalable, sustainable resource management. However,

given the nature of issues to be addressed by such a system, we would also like to use such a system to maximize the machine utilization, *e.g.*, in the event of so-called “drain” operations where the machine is cleared of running jobs. Minimizing the time required to create checkpoint files is essential to preserve the efficiency of the resource. Our implementation goal is to spend no more than 5% of a job’s wall-clock time within the checkpoint library.

In our early design discussions, we concluded that high-end users desiring to run on thousands of processors were a) typically already using some kind of user-level checkpointing, and b) willing to invest some effort to ensure the recoverability of these unusually large jobs. Therefore we designed an infrastructure that requires the user make slight modifications to both source code and job scripts, as discussed below.

## User library & tools

The Application Program Interface (API) for the PSC’s checkpoint library (see Appendix A) is written in such a way as to be directly applicable to C, C++, and Fortran developers. Users must include these functions in their source code to access the functionality offered by the checkpoint system (see sample code in Appendix B). If the user utilizes this library and invokes the recovery properly in the job script (see sample script in Appendix C), then the integrated job recovery infrastructure will ensure that the job will be automatically recovered and restarted. The user can also utilize a set of shell tools (see list in Appendix D) in the job script or in interactive sessions, which have been provided for certain non-programmatic tasks.

Although users could write their own checkpoint algorithms, such checkpoint files can be lost or unavailable in the event of machine failure. The explicit advantage of this custom checkpoint library is that files written via this system are inherently recoverable and automatically migrated to the proper nodes when job restart is requested.

Checkpoint files written via the checkpoint library have a well-defined and easily understood format, which allows for later direct re-use by applications outside of the checkpoint system. Users can thus leverage the use of checkpoint files for other purposes, *e.g.* visualization files or job status monitoring.

The checkpoint library contains some job monitoring functionality as well as checkpoint and recovery. A user can invoke the `tcs_postmessage_` function (see Appendix A) to track the execution of a running application. Messages can be posted from any combination of processes in an active job. All messages passed to the `tcs_postmessage_` function will be displayed on the web-based monitoring applet showing the TCS machine status (see <http://www.psc.edu/machines/tcs/#monitor>). Furthermore, by formatting the message appropriately, the user can direct that the message is also sent out to an arbitrary email address. This could enable users to track the status of running jobs via a PDA or other wireless devices.

## Checkpoint Plans

PSC's implementation of the checkpoint system allows the user to choose from among several possible checkpoint strategies, or "Plans". Each Plan is identified by an alphanumeric string (e.g. "C1") for convenience. To select a specific Plan, the user can set the TCS\_PLAN environment variable in the job script to one of the alphanumeric values listed below. Some Plans have additional parameters that the user can use to control the precise behavior of the checkpoint. These are listed below in the descriptions of each Plan. If the environment variables specified below are undefined then the checkpoint system will supply default values, also identified below. The default values have been chosen to provide the best all-around performance while also assuring the correct behavior of the checkpoint.

To date, in all Plans the "primary" copies of checkpoint files are written to local disk on the compute nodes. Different Plans achieve robustness by backing up these primary copies in different ways. There may, however, be future implementations in which primary checkpoint files are written directly to remote servers (*i.e.* I/O "redirection" Plans) thus avoiding the potential loss due to compute node failure just after the checkpoint has completed but before the files have been redundantly backed up.

### ***Plan "B2": Duplication to File Server***

Plan B2 is a direct, "brut force" approach. Under Plan B2 every primary checkpoint file is copied to a remote file server immediately after closure. Within the *tcs\_close\_* method implementation, a request is passed to the TCS IO daemon on the (local) compute node requesting that the checkpoint file be replicated to a file server. The user's job then returns to computation while the local daemon performs the third-party file migration on the user's behalf. Primary checkpoint files that are unavailable due to machine failure are thus recovered from their duplicate on the file server.

This Plan achieves the maximum effective bandwidth to disk, utilizing the system's buffered I/O. It also imposes no limits on the size of buffers passed to the *tcs\_write\_* function. Furthermore, since file migration is done by a separate process (the TCS IO daemon), computation is minimally impacted by IO operations. However, this Plan doubles the disk space requirements of the checkpoint system and there is an implicit latency (the file migration time) during which a successfully checkpointed program could lose a node and be forced to recover from a previous checkpoint.

### ***Plan "C1": XOR-based Parity File***

Plan C1 is an implementation making use of unused computing cycles during an I/O operation. Under Plan C1, parity files are calculated via a bit-wise XOR of checkpoint data. On every processor, data are written to the primary checkpoint file via asynchronous I/O while also participating in a checkpoint calculation. The set of processors in the job is divided up into sets spanning N processes, each on different nodes, where N is a configurable parameter. The value of N can be set via the environment variable TCS\_NODES\_XOR; its default value is 8. Thus, a job can be recovered if no more than one out of every N nodes (per XOR set) has failed. During the

write operation, the first process in each XOR set performs an *MPI\_Reduce* operation, calculating the XOR value and writing the resulting data to an XOR file, which is stored on a remote file server. The data from any missing node(s) can be regenerated by the reverse XOR calculation, including the N-1 primary checkpoint files and the XOR file for that set.

By calculating only one XOR file per TCS\_NODES\_XOR files, the disk utilization overhead of this scheme is minimized. However, this Plan requires that there is sufficient free memory to allocate an additional XOR buffer as large as the buffers passed to the *tcs\_write* function. Thus, if a user were utilizing 3GB of memory on each node, a write operation of chunks much larger than 1GB would result in a failure, since our compute nodes are currently configured with only 4GB of physical memory. Furthermore, the asynchronous I/O library (*libaio.so*) cannot make use of the operating system's buffered I/O. Thus, it achieves apparent write bandwidths more characteristic of direct I/O.

## Implementation Details

The design goals of this project are sufficiently broad in scope that an in-depth discussion of implementation details is merited. We address here several facets of the implementation in an effort to provide greater technical clarity on some of the features discussed above.

All components of this system have been written in C, C++ or Perl in such a way as to be highly portable. The presence of a locally-accessible SQL database is required. The high-performance bulk-data transport layer is, however, machine- and interconnect-specific.

### **Administration**

The installation and version management of this system software is achieved by Depot (see <http://asg.web.cmu.edu/depot/>), a management tool developed at Carnegie Mellon University. It automatically distributes all of the installation files to all of the file systems involved.

State information for the checkpoint system as a whole is maintained in a central Mini SQL (see <http://www.hughes.com.au/library/msql/>) database. This allows read-only access to both users and operators from all nodes within the system and write-access to root processes as well. The checkpoint system stores and manages a significant amount of state information in this database, which is shared by the Resource Management System (RMS). The presence of available checkpoint files, checkpoint activity and authentication information, checkpoint system performance and other job and status-dependent variables are all stored in this central database.

The system requires the presence of daemons (*tcsiod*) running on each compute node and file server. These daemons serve in the authentication and transport operations to provide asynchronous file migration for active jobs, access to the native Quadrics transport layer, access to remote, locally-attached storage, read/write access to the central database, and other implementation-specific functionality. A process control script for this daemon

must reside in /sbin/init.d, with appropriate symbolic links in /sbin/rc#.d, on each compute node and file server as well. There is a command-line tool to verify the presence of responsive daemons throughout the system (see Appendix D).

Job- and daemon- related activity, including error and diagnostic conditions, is logged via the UNIX syslog utility. These log messages are configured to log, not only to each node (localhost), but also to a centralized “management server”. In this way near real-time status messages are collected and filtered to provide an additional diagnostic of the checkpoint/recovery system.

## **Security**

Security in this system is primarily concerned with access permissions for both the database, since state information for each job is stored here, and the checkpoint files. This system uses the standard UNIX file system protection scheme when handling or providing access to any and all checkpoint files. Thus, when reading or writing files via the TCS checkpoint library API (see Appendix A) the protection presented by the file system is sufficient. Regarding created-file permissions, ownership of migrated files can either follow the UID of the requestor, given sufficient permissions to read the file (as with the UNIX “cp” command), or the stats of the original file can be preserved.

Since “third-party” operations are performed by the daemon on behalf of the user, additional care is needed. In such instances the user’s process, utilizing an encapsulated, object-oriented interface to the library, passes its UID/GID to the daemon with any requests. The UID/GID received by the daemon is then used in determining further access permissions.

## **Communication**

The communication layer consists of a single C++ object (“Connection”) that currently encapsulates standard TCP socket communications. It is highly portable and has been used in other projects and on many other platforms. This communication class is used to handle the passing of any and all meta-data and command invocation requests and return parameters. It is an ultra-lightweight schema patterned after the CORBA model.

In the case of bulk data transport, the Connection class is used to handle only the meta-data while the transport itself is handled by a second custom communication library developed at PSC, based on the native Quadrics “Elan3” communication library. This library has been benchmarked at greater than 200 Mbytes/sec transport rates utilizing a single rail of the computational interconnect. Given these two communication vehicles, our infrastructure employs a highly portable control channel and a high-performance data channel.

## **Storage Architecture**

Our system includes 64 file servers that are each equipped with 500 GB of disk storage for user files. This storage is presented via direct-attached SCSI. Each SCSI chain will be attached to two file servers, exploiting a multi-initiator SCSI configuration. In this manner users will be able to retrieve files from this user file system even if one of the file

serving nodes fails. The TCS system was designed to utilize up to 64 file such file servers whose function can be dynamically controlled. Specifically, file servers can be included in the compute server pool and thus used to run user jobs or they can be excluded from this pool of compute nodes, thus serving I/O operations exclusively. If there is a job queued that has high I/O requirements, the system can schedule it for a time when all 64 file servers are held free of running jobs. This is an implementation feature of our customizations to the Portable Batch Scheduler (PBS, see below).

Key aspects of the storage architecture are configurable via the contents of a configuration file (`/etc/tcsiod.conf`), *e.g.* the host and path specification of primary checkpoint files, parity files, redundant file server fail-over, and the amount of performance logging. Following is a sample segment of that file:

```
# define complex path for local (primary) checkpoint files
TCS_LOCALCHKDIR {local}:/local1/tcschk,/local2/tcschk

# define complex path for central (duplicate) checkpoint files
TCS_CENTRALCHKDIR lemieux[0-32](n+1):/usr/tcschk
```

All “complex” paths specified in this configuration file have the format:  
*hostExp(redundancy):full\_path1,full\_path2*

The host expression (*hostExp*) can either be an RMS-style regular expression, as shown in the value of `TCS_CENTRALCHKDIR`, which can be expanded into a list of node names, or a generic host specification surrounded by braces, as shown in the value of `TCS_LOCALCHKDIR`. We currently support two generic specifications: `{local}`, meaning the actual name of localhost (via `gethostname()`); and `{cluster}`, meaning any node in the TruCluster of which this node is a member. Compaq TruCluster is a software management layer over the Compaq Tru64 operating system that clusters nodes together into groups of up to 32 nodes and provides some failover infrastructure and shared file systems among those nodes.

The complex path can include a “redundancy” specification. The redundancy specifies, for example in the “n+1” case shown, that file servers are paired together as node0:node1, node2:node3, node4:node5, etc. This type of redundancy can be achieved via multi-initiator SCSI between two file-serving nodes. In the event of a file serving node failure, any process attempting to retrieve a checkpoint file has a straightforward path for attempting retrieval from an alternative node--its redundant partner.

In each of the host specifications described above, if the local host is a member of the expanded host list then it is designated as the “target” node, or the node to which checkpoint files will be written. All other nodes in the expanded list will serve as “failover” nodes for reading back those files. If the local host is not a member of the expanded host list then the target node is chosen by a load-balancing scheme employing the requesting agent’s host ID (the numeric part of its host name) modulated by the number of values in the expanded host list.

There is a further load-balancing feature. If the target host is the same as localhost, then the comma-separated list of full paths shown are compared according to available space, whereupon the path with the most free space is chosen. Otherwise, the target path is selected by the same load-balancing scheme described in the previous paragraph.

These load-balancing features allow for an extremely flexible and configurable system at the cost of only a few configuration parsing functions.

## ***Performance Monitoring***

The checkpoint library implementation includes internal diagnostic and performance monitoring. I/O traffic involving the checkpoint library is logged to the central SQL database when checkpoint files are closed. The library is currently tracking time spent in open, read, write and close operations as well as the amount of data read and written and an error mask indicating any errors encountered by the user. Thus administrators can obtain valuable usage and performance information for the library on a per-file basis. Administrators can completely deactivate this feature at any time by modifying the value of a parameter in the TCS I/O configuration file (/etc/tcsiod.conf).

## **Scheduling**

Integration with the scheduling system is a critical aspect of the checkpoint and recovery system. PSC added modifications to our port of the Open version of the Portable Batch Scheduler (OpenPBS) to enable automated job recovery. Significant effort was invested to port OpenPBS to the Compaq AlphaServer SC platform, integrating it into RMS. That effort leverages the presence of the RMS database, as does the checkpoint system. User interaction with the compute nodes is all controlled and monitored by OpenPBS.

OpenPBS provides convenient customization points. Specifically, it allows for the execution of “prologue” and “epilogue” scripts that run immediately before and after each submitted job, respectively. The OpenPBS prologue script assists in the initialization of the checkpoint system by invoking the “tcsinit” shell tool (see Appendix D).

The OpenPBS epilogue script triggers job recovery. This script queries several status values from the central database including the RMS job status (the status resulting from running the “prun” job-execution command) and the checkpoint system status for that job and uses them to look up values in an “action matrix”. Each element in the action matrix determines what actions should appropriately follow a job that terminated with the given status values. For example, if the status values indicate that there was a node failure, then the action(s) triggered are: calculate “lost” time (see “Accounting” below) and rerun the job. When restarting the job, OpenPBS is free to allocate any set of nodes for the recovered job. The recovery system itself provides for recovery and migration of all necessary checkpoint files to whatever host destinations are appropriate to the restarted job. If, however, the status values indicate that the job was aborted by the user, then OpenPBS will invoke the “tcsrjob” cleanup shell tool (see Appendix D) to purge the checkpoint system resources held by that job.

In the case of a restarted job, the recovery of checkpoint files must currently be handled within the user's job script (see Appendix C). The time required to recover can vary widely depending, for example, upon job size and checkpoint file sizes. While we had considered triggering job recovery via the OpenPBS prologue script, this wide variability in recovery time made it more reliable to trigger recovery within the user's job script. The recovery time, is also recorded and credited back to the user's grant allocation (see "Accounting" below).

Determining which jobs should be restarted is a critical element of the automated recovery system. Discriminating between job termination caused by machine failure and job termination for other reasons, *e.g.* successful completion, user abort, or other software abort conditions, required a small amount of additional customization of the system management software. RMS provides the capability to recognize when nodes have failed. Node failure is one of RMS's well-defined set of "event classes". The administrator has the ability to direct the RMS event handler to run specific scripts according to the event type. We created a "node event" handler script that notifies the checkpoint system if any node running an active job fails. In such a case, the checkpoint system sets a flag in the central SQL database requesting that the job be recovered. This flag is subsequently queried by the OpenPBS epilogue script, as described above.

## Accounting

Users of the TCS machine are permitted access by means of a grant allocation, which has an explicit maximum allocation time. All machine utilization, excluding that on interactive nodes, is timed and this time is subtracted from the grant allocation until it is expired. This is the standard operating mode of virtually all supercomputing centers.

Users submit queue their jobs for running on the TCS via OpenPBS. Each request is, by definition, an OpenPBS "job". When the scheduler chooses a job for running, it creates an RMS "resource" which is effectively a reservation of a set of TCS nodes. A user's job script may include more than one command and application execution, provided that they only utilize the nodes included in the RMS resource. Our accounting system bases its definition of "service units" on the wall clock time and node count used by a user's RMS resources.

If a user's job is running on a node which fails, then the calculations performed since the last complete checkpoint will be effectively lost. The checkpoint system automatically calculates the amount of wall-clock time which was lost, based on the time stamp on the checkpoint file entries in the central database. If that time does not exceed a maximum refund amount, a value set by policy, then the time will be refunded to the user's grant allocation. Once the "lost time" has been calculated for a failed RMS resource, the job is requeued and becomes eligible for execution again, thus acquiring an additional RMS resource.

The actual charge to a user's grant allocation is then a simple sum:

$$C = \sum_{i=1}^{N_r} UT_i - RT_i - \min(LT_i, MaxLT)$$

where:

$C$  = total charge for the user's job

$N_r$  = total number of RMS resources created by an OpenPBS job

$UT_i$  = wall-clock time for each RMS resource

$RT_i$  = time spent in checkpoint file recovery for each RMS resource

$LT_i$  = "lost time" calculated for each RMS resource

MaxLT = maximum refundable "lost time", set by policy

## Appendix A: TCS Checkpoint Library API

The Application Programming Interface (API) for the PSC checkpoint library is written in such a way as to be directly accessible to C, C++, and Fortran developers (Fortran users invoke these functions without the trailing underscore). Following is a discussion of each of the functions in the current library.

### **int tcs\_init\_(void \*exp)**

This function initializes the TCS IO library. It must be called before any other functions in the library. If this function is not called first, all other TCS IO library functions will fail and print an appropriate error message to STDERR. The purpose of the *tcs\_init\_* function is to allocate the necessary resources (e.g. in the RMS DB, the file systems, etc.) to track the state of the running job. The *exp* ("expansion") parameter is reserved for future use, so users must specify it as 0. The function returns a non-zero value if unsuccessful.

### **int tcs\_jobrestarted\_(void)**

This function indicates whether a running job is a restarted job or not. It returns zero if it is not a restarted job. It returns a positive integer if the job has been automatically restarted. It returns a negative integer if it failed to determine the restart status of the job. A job should call this function to determine whether it should read data from checkpoint files before resuming normal processing or just begin normal processing.

### **int tcs\_open\_write\_(char \*prefix)**

This function opens a checkpoint file for writing. The value of the *prefix* variable is an alphanumeric string of no more than 20 characters. Each *prefix* corresponds to a *prefix set* of checkpoint files. No two open checkpoint files can have the same *prefix*, but if a user closes a checkpoint file and then opens another with the same *prefix*, a new checkpoint file is created in the set of files for that *prefix* and the "series" number is incremented by one. A program can have multiple *prefix sets*, with different *prefixes*, open at the same time. Multiple *prefix sets* would be used to store different types of data in separate checkpoint files. A complete checkpoint set must have exactly one file for each *prefix set*. Such a set is uniquely distinguished by its "series" number. Job recovery is thus performed by gathering all files for a given job ID having the latest series number.

When using a *prefix* to open a checkpoint file for reading, a job will be given the latest complete file in the set of files for that *prefix*. Thus, a common checkpoint strategy will be to open, for each group of data to be saved, a checkpoint file with an appropriate

*prefix* at the bottom of the program's "outer loop", write the checkpoint data, and then to close each file. Each pass through the loop will create one new series of a checkpoint file for each of the checkpoint *prefix* sets. A barrier may be required to prevent the *prefix* sets from becoming out of sync by more than one series across the processors. If there is a system failure the latest checkpoint file in each *prefix set* will be accessible. This function returns -1 if unsuccessful. Otherwise it returns a logical unit number, which must be passed to the write and close functions.

### **int tcs\_open\_read\_(char \*prefix)**

This function opens a checkpoint file for reading. This is the function to use if the function *tcs\_jobrestarted\_* indicated that the current job is a restarted job. The value of the *prefix* variable is an alphanumeric string of not more than 20 characters that is used to determine which set of checkpoint files will be considered for opening; it corresponds to the values passed to the *tcs\_open\_write\_* function. The job will be given the latest complete checkpoint file from the set of files with the *prefix* specified. This function returns -1 if unsuccessful. Otherwise it returns a logical unit number, which must be passed to the read and close functions.

### **int tcs\_write\_(int \*lun, void \*A, const long \*len, int \*datasize)**

This function writes to a checkpoint file. The value of *len* is a logical unit number from a previous invocation of *tcs\_open\_write\_*. Variable *A* is a pointer to the buffer where the data to be written out to the checkpoint file resides. Variable *len* specifies the number of data elements to write, while *datasize* indicates the size of a data element in bytes. This function returns a non-zero value if unsuccessful.

### **int tcs\_write8\_(int \*lun, void \*A, const long \*len)**

This function writes 8-byte data elements to a checkpoint file. The value of *lun* is a logical unit number from a previous invocation of *tcs\_open\_write\_*. Variable *A* is a pointer to the buffer where the data to be written out resides. Variable *len* indicates the number of 8-byte data elements to write. This function returns a non-zero value if unsuccessful.

### **int tcs\_write4\_(int \*lun, void \*A, const long \*len)**

This function writes 4-byte data elements to a checkpoint file. The value of *lun* is a logical unit number from a previous invocation of *tcs\_open\_write\_*. Variable *A* is a pointer to the buffer where the data to be written out resides. Variable *len* indicates the number of 4-byte data elements to write. This function returns a non-zero value if unsuccessful.

### **int tcs\_read\_(int \*lun, void \*A, int \*len, int \*datasize)**

This function reads from a checkpoint file. The value of *lun* is a logical unit number from a previous invocation of *tcs\_open\_read\_*. Variable *A* is a pointer to the data buffer where the data will be placed. Variable *len* is the number of data elements to be read from the checkpoint file, while *datasize* indicates the size of a data element in bytes. This function returns a non-zero value if unsuccessful.

### **tcs\_read8\_(int \*lun, void \*A, long \*len)**

This function reads 8-byte data elements from a checkpoint file. The value of *lun* is a logical unit number from a previous invocation of *tcs\_open\_read\_*. Variable *A* is a pointer to the data buffer where the data will be placed. Variable *len* indicates the number of 8-byte elements to read. This function returns a non-zero value if unsuccessful.

### **tcs\_read4\_(int \*lun, void \*A, long \*len)**

This function reads 4-byte data elements from a checkpoint file. The value of *lun* is a logical unit number from a previous invocation of *tcs\_open\_read\_*. Variable *A* is a pointer to the data buffer where the data will be placed. Variable *len* indicates the number of 4-byte elements to read. This function returns a non-zero value if unsuccessful.

### **int tcs\_close\_(int \*lun, int \*keep)**

This function closes a checkpoint file. The value for *lun* is a logical unit number from a previous invocation of *tcs\_open\_read\_* or *tcs\_open\_write\_*. The value for *keep* can be 0 if a user just wants checkpoint files to be used for recovery purposes, in which case the checkpoint file is saved in a system file area. However, a user can specify a value of 1 for *keep* to store a copy of the file in the job's \$TMPDIR directory. This *keep* feature thus allows for multiple uses of checkpoint files. This function returns a non-zero value if unsuccessful.

### **int tcs\_finalize\_(void)**

This function releases the resources allocated by the *tcs\_init\_* function. Users should call it at the end of their job. It returns a non-zero value if unsuccessful.

### **int tcs\_postmessage\_(const char \*msg, int \*length)**

This function posts the contents of the *msg* parameter to the RMS DB in the *checkpoint\_jobs.message* field. Only the most recent message posted will be retained. The TCS monitor (see <http://www.psc.edu/machines/tcs/#monitor>) will retrieve the contents of this field and display them in the legend at the bottom, after the user's name and job ID. Thus, you can use this function to track the progress of your program. Please note that your message will be visible to all users of the TCS monitor. The *length* parameter indicates the length of the total message string. It can be specified as 0 if the message string is null-terminated.

If *msg* points to a string with the format "mailto:userid@machine some message" where "userid@machine" is a valid email address, then, when this function is called, an email message will be sent to the specified address with "some message" as the subject of the email message, as well as to the RMS database as described above. Since there is some performance penalty for using this function, particularly when using it to send email messages, you may want to use it sparingly. The function returns a non-zero value if unsuccessful.

## **int tcs\_drainoperation\_(void)**

If a user's job has short checkpointable intervals (*e.g.* requires only 15 minutes to complete the calculations in each "time step") the user may want to write checkpoint files only after every "N" intervals. Typically the TCS machine administrators will know well in advance when the machine is going to be "drained" of running jobs. In such situations the TCS administrator can post a notice to running jobs, via the RMS database, notifying them of an impending drain operation. This function queries the state of that flag. It returns a positive integer if there is an impending drain operation, zero if no such notice has been posted, and a negative integer if it was unable to determine that status. If it returns a positive integer then the program should write a checkpoint as soon as possible to avoid loss of work.

## **Appendix B: Sample User Code**

Following is the source code for an example, written in C, illustrating the proper usage of the PSC checkpoint system.

```
//
// TCS IO library test C program
//
// Authors: Nathan Stone
//
// Copyright 2001, Pittsburgh Supercomputing Center (PSC),
// an organizational unit of Carnegie Mellon University.
//
// Permission to use and copy this software and its documentation
// without fee for personal use or use within your organization is
// hereby granted, provided that the above copyright notice is
// preserved in all copies and that that copyright and this permission
// notice appear in any supporting documentation. Permission to
// redistribute this software to other organizations or individuals is
// NOT granted; that must be negotiated with the PSC. Neither the
// PSC nor Carnegie Mellon University makes any representations about
// the suitability of this software for any purpose. It is provided
// "as is" without express or implied warranty.
//
// $Id: test_c.c,v 1.26 2001/09/28 16:36:49 nstone Exp $
//
#include <mpi.h>
#include <stdio.h>
#include "tcsio.h"

int main(int argc, char**argv){
    int rank;
    int flun;
    int plun;
    int fnum;
    int pnum;
    long numInt;
    int intSize;
    int keep;
    int i;
```

```

char message[80];

printf("Starting main...\n");

keep = 0; // do not stage checkpoint files out to $TMPDIR
flun = -1;
plun = -1;
fnum = 235;
pnum = 101;
numInt = 1;
intSize = sizeof(int);

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank );

// initialize the TCS IO library
if (0>tcs_init_(0)){
    printf("Failed to initialize TCS library\n");
    return -1;
}

// recover a failed previous job (of the same jobID)
if (0!=tcs_jobrestarted_()){
    if (0==rank){
        printf("This is a restarted job...\n");
    }

    flun = tcs_open_read_("fields");
    if (0>flun){
        printf("Failed to acquire fields LUN\n");
        return -1;
    }

    plun = tcs_open_read_("particles");
    if (0>plun){
        printf("Failed to acquire particles LUN\n");
        return -1;
    }

    if (0!=tcs_read_(&flun, &fnum, &numInt, &intSize)){
        printf("Failed to read from fields LUN %d\n",flun);
        return -1;
    }
    printf("Got fields checkpoint value: %d\n", fnum);

    if (0!=tcs_read_(&plun, &pnum, &numInt, &intSize)){
        printf("Failed to read from particles LUN %d\n",plun);
        return -1;
    }
    printf("Got particles checkpoint value: %d\n", pnum);

    if (0!=tcs_close_(&flun, &keep)){
        printf("Failed to close fields LUN %d\n", flun);
        return -1;
    }

    if (0!=tcs_close_(&plun, &keep)){

```

```

        printf("Failed to close particles LUN %d\n", plun);
        return -1;
    }
} else {
    if (0==rank){
        printf("This is NOT a restarted job...\n");
    }
}

// perform normal calculations
for (i=1; i<=60; i++){
    printf("Time step: %d\n", i);
    if (0==rank){
        //      sprintf(message, "mailto:myself@foo.bar.edu starting
iteration %d\0", i);
        sprintf(message, "starting iteration %d\0", i);
        tcs_postmessage_(message, 0);
    }

    sleep(1);
    // put up a barrier to keep checkpoint series from getting out of
sync
    // by more than one timestep
    MPI_Barrier(MPI_COMM_WORLD);

    if (0==i%10 || tcs_drainoperation_()){
        printf("Checkpointing...\n");

        flun = tcs_open_write_("fields");
        if (0>flun){
            printf("Failed to acquire fields LUN\n");
            return -1;
        }

        plun = tcs_open_write_("particles");
        if (0>plun){
            printf("Failed to acquire particles LUN\n");
            return -1;
        }

        if (0!=tcs_write_(&flun, &fnum, &numInt, &intSize)){
            printf("Failed to write to fields LUN %d\n",flun);
            return -1;
        }

        if (0!=tcs_write_(&plun, &pnum, &numInt, &intSize)){
            printf("Failed to write to particles LUN %d\n",plun);
            return -1;
        }

        if (0!=tcs_close_(&flun, &keep)){
            printf("Failed to close fields LUN %d\n", flun);
            return -1;
        }

        if (0!=tcs_close_(&plun, &keep)){
            printf("Failed to close particles LUN %d\n", plun);

```

```

        return -1;
    }
}
}
printf("Finalizing...\n");
if (0!=tcs_finalize()){
    printf("Error finalizing TCS library\n");
}
MPI_Finalize();
printf("Done.\n");
return 0;
}

```

## Appendix C: Sample User PBS Job Script

Following is an example of a PBS job script illustrating the proper invocation of the PSC recovery system and job execution.

```

#!/bin/sh
#
# Define your PBS directives here...
#
#PBS -l walltime=10:00

#
# Run the recovery for successive, restarted runs
# (On the first invocation, it will just return a zero...)
#
# !! Do this first !!
# Do not continue your job in the case of a non-zero return value!
#
tcsrecover
if [ $? -ne 0 ]; then
    echo "Recovery Failed!"
    exit 1
fi

#
# Check if this job is running under restart.
# If so, then avoid redundant setup operations.
#
tcsisrestart
if [ $? -ne 1 ]; then
    far rget input_files/ $STAGEDIR
fi

#
# Uncomment one of these to select a specific checkpoint plan
# (A1, A2, B1, B2, C1, C2 -- default is C1)
#
#TCS_PLAN=B2
#TCS_PLAN=C1
#export TCS_PLAN

#

```

```
# Uncomment this to set the number of nodes per XOR set
# (for C plans only -- default value is 8)
#
#TCS_NODES_XOR=4
#export TCS_NODES_XOR

#
# run the job
#
prun -t -N ${RMS_NODES} -n ${RMS_PROCS} ./a.out
```

## Appendix D: TCS System Shell Tools

The library includes a set of companion shell tools that are listed and described below. These tools provide an interactive interface to some of the functionality of the checkpoint and recovery system.

### **tcsrecover**

This tool should be used within a user's job script (see Appendix C). It retrieves all available checkpoint files and then either retrieves backup copies or regenerates from parity files any checkpoint files that were lost or otherwise irretrievable. It then redistributes the files to the appropriate nodes where the recovered job will run. In the case of a failed recovery it returns a non-zero value to the shell. It can also be used interactively, although the redistribution of checkpoint files in that case is of questionable utility. It automatically records the recovery time in the central database so that this time can be credited back to the user's grant allocation.

### **tcsisrestart**

This tool should be used within a user's job script (see Appendix C). It determines the status of the running job, whether or not it was restarted by the automated recovery system. It can also be used interactively to query the restart status of any job, given the OpenPBS job ID. It returns a positive integer if the job is running under restart, a zero if it is not under restart, and a negative integer if it failed to determine the restart status.

### **tcscp**

This tool was designed to facilitate high-performance file migration for active jobs, although it can also be used interactively. It utilizes a custom interface to the native Quadrics interconnect (ELAN3) to migrate files within the TCS machine. Its command-line invocation is comparable to the standard UNIX "cp" utility.

### **tcsinit**

This tool can only be used by user "root". It is the means by which a new entry is created in the central database in order to store the state information for a new job. This tool is invoked in the OpenPBS prologue script.

### **tcscalctime**

This tool can only be used by user "root". It is the means by which the checkpoint system calculates the amount of computation time that was lost due to any machine

failure. The “lost time” is recorded in the central database so that it can be credited back to the user’s grant allocation. This tool is invoked in the OpenPBS epilogue script.

### **tcsrmjob**

This tool can only be used by user “root”. While users’ jobs automatically clean up and release any resources allocated during their jobs, this tool provides a vehicle for ensuring proper cleanup following either improper checkpoint library usage or abnormal job termination not due to machine failure.

### **tcsping**

This tool can be used to determine if a TCS I/O daemon is running on a set of nodes. It takes a single command-line argument: an RMS-style regex host specification for hosts to check for the presence of a responsive “tcsiod”.