

## Mass Storage on the Terascale Computing System

**Nathan T.B. Stone** <nstone@psc.edu>, **J. Ray Scott** <scott@psc.edu>, **John Kochmar** <kochmar@psc.edu>, **Jason Sommerfield** <jasons@psc.edu>, **Ravi Subramanya** <ravi@psc.edu>, **R. Reddy** <rreddy@psc.edu>, **Katie Vargo** <katie@psc.edu>

Pittsburgh Supercomputing Center, Pittsburgh, PA 15213

Phone: +1 412 268-4960, Fax: +1 412 268-5832

### Abstract

On August 3rd, 2000, the National Science Foundation announced the award of \$45 million to the Pittsburgh Supercomputing Center to provide "terascale" computing capability for U.S. researchers in all science and engineering disciplines.

This Terascale Computing System (TCS) will be built using commodity components. The computational engines will be Compaq Alpha CPU's in a four processors per node configuration. The final system will have over 682 quad processor nodes, for a total of more than 2700 Alpha CPUs. Each node will have 4 gigabytes of memory, for a total system memory of over 2.5 terabytes. All the nodes will be interconnected by a high speed, very low latency Quadrics switch fabric, constructed in a full "fat-tree" topology.

Given the very high component count in this system, it is important to architect the solution to be tolerant of failures. Part of this architecture is the efficient saving of a program's memory state to disk. This checkpointing of the program memory should be easy for the programmer to invoke and sufficiently fast to allow for frequent checkpoints, yet it should not severely impact the performance of the compute nodes or file servers. There will be flexibility in the recovery so that spare nodes can be automatically swapped in for failed nodes and the job restarted from the most recent checkpoint without significant user or system management intervention. It is estimated that the file servers required to collect and store these checkpoints and other temporary storage for executing jobs will collectively have ~27 terabytes of disk storage. The file servers will maintain a disk cache that will be migrated to a Hierarchical Storage Manager serving over 300 terabytes.

This paper will discuss the hardware and software architectural design of the TCS machine. The software architecture of this system rests upon Compaq's "AlphaServer SC" software, which includes administration, control, accounting, and scheduling software. We will describe our accounting, scheduling, and monitoring systems and their relation to the software included in AlphaServer SC.

### Hardware Architecture

The computational core of the Terascale Computing System (TCS) will consist of over 682 Compaq ES40-series quad Alpha processor nodes. Each node will be equipped with 4 gigabytes of main memory and 36 gigabytes of disk storage.

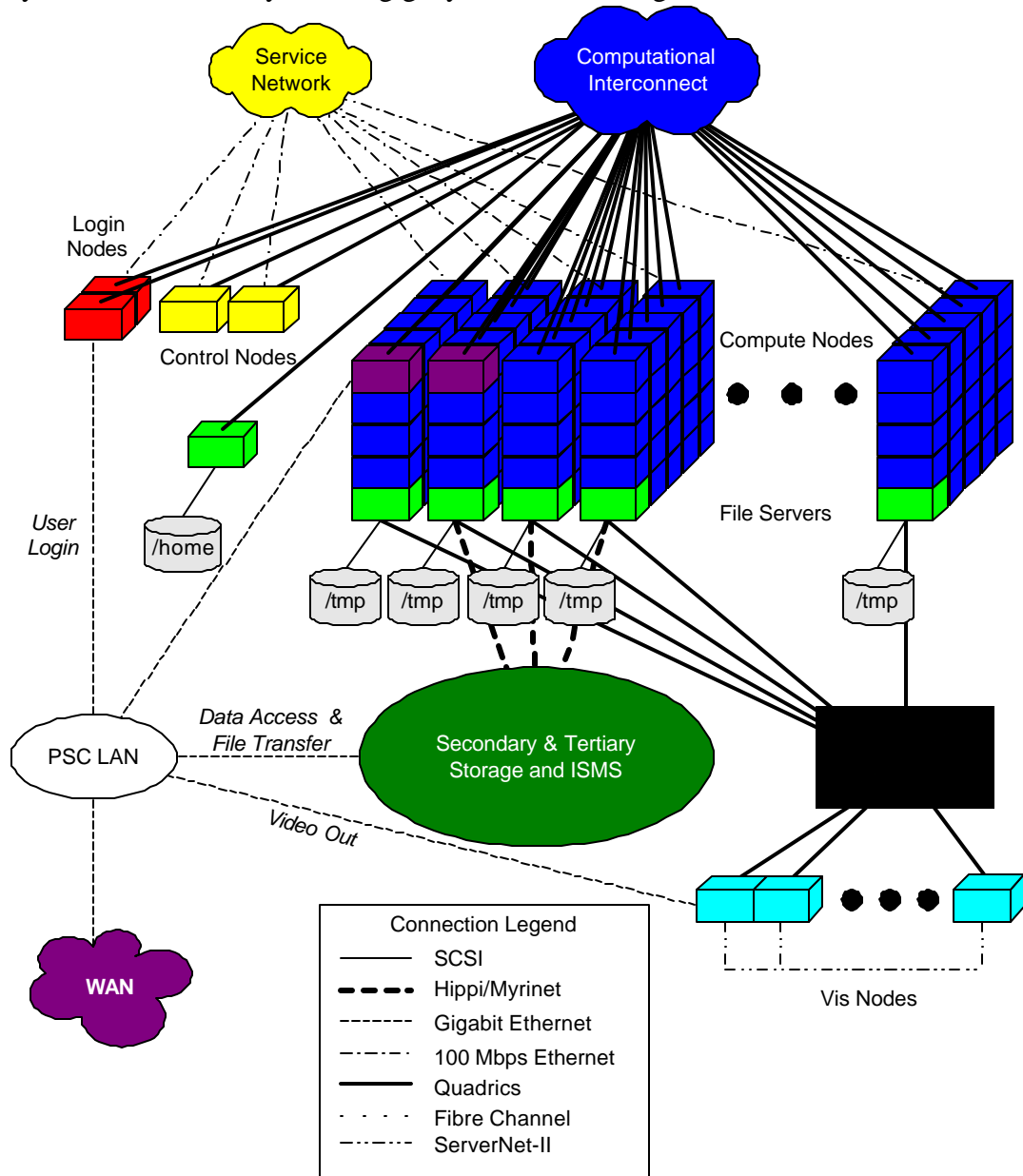


Figure 1: A logical view of the TCS machine architecture

Our administration of the system requires a logical distinction of the roles of each node in the TCS machine, as shown in Figure 1. The TCS machine will contain the following types of nodes: “login” nodes, accessible to remote users via secure connections; “compute” nodes, comprising most of the TCS machine; “file server” nodes, specially equipped to provide high-speed access to user-oriented storage (*e.g.* /home and /scratch); “visualization” nodes, specially equipped and connected to the WAN to provide real-time renderings and streaming visualizations to remote users; and “control” nodes, to

administer machine status, configuration, and accounting information. All nodes, with the possible exception of the visualization nodes, will share a high-bandwidth (hundreds of megabytes/sec), low-latency (few microsecond) Quadrics interconnect. The TCS will have an additional “maintenance” network used only for administrative configuration and control of the machine. A Hierarchical Storage Manager (HSM) will manage long-term storage on a separate system. Some file servers will have dedicated connectivity to the HSM.

## **File Systems and Storage**

We will be running Compaq’s “AlphaServer SC” software to manage the entire machine. AlphaServer SC utilizes Compaq’s Tru64 operating system and TruCluster software to facilitate administration of the nodes in the machine. In doing so, it administratively divides the pool of compute nodes into smaller “domains” of up to 32 nodes, all served by a common Cluster File System (CFS). All nodes within a CFS domain share their “cluster common” directories “/”, “/usr”, and “/var”. This provides a single, coherent administrative file system among these 32 nodes.

AlphaServer SC also includes a Parallel File System (PFS) utility for striping across separate logical volumes. It is expected that Compaq will release a higher performance file system targeted at utilizing the Quadrics interconnect for traffic, and in so doing present the opportunity to take full advantage of parallel access to multiple file servers. In this case we will investigate the performance issues related to forming truly global and parallel file systems via these products.

The vast majority of user IO will be accomplished within the realm of the file servers. We expect to create the file serving nodes by equipping up to 64 nodes with additional disk controllers to provide nearly 30 terabytes of user storage. A variety of software and management systems are being investigated for providing high-performance access to that user storage across the machine as a whole. At the hardware level, the file servers will share the Quadrics interconnect, allowing high-performance, non-contending network paths for file traffic between any two nodes. These storage systems will each host not only user home directories but will also host scratch space for active or near-active jobs (*i.e.* for staging or recovering files).

Data that is written with the intention of being used in case of machine failure, so called “checkpoint” data, as well as other data written to the scratch space will be allowed to “expire” after a user-specified amount of time, up to some administratively defined maximum time. Data that the user wishes to keep must be migrated from scratch space to the Hierarchical Storage Manager (HSM).

Our system includes an HSM that contains over 150 terabytes of media in multiple tape silos. The HSM is designated as the sole residence of any files that are to be persistently saved after job termination (*e.g.* computational snapshots and visualization files). Some file servers will be equipped with high-bandwidth HIPPI connectivity to our HSM. These nodes, “IO Concentrators,” will manage the traffic involved in staging files to and

from the HSM. Users can invoke staging commands either online or within the scripts submitted for job submission.

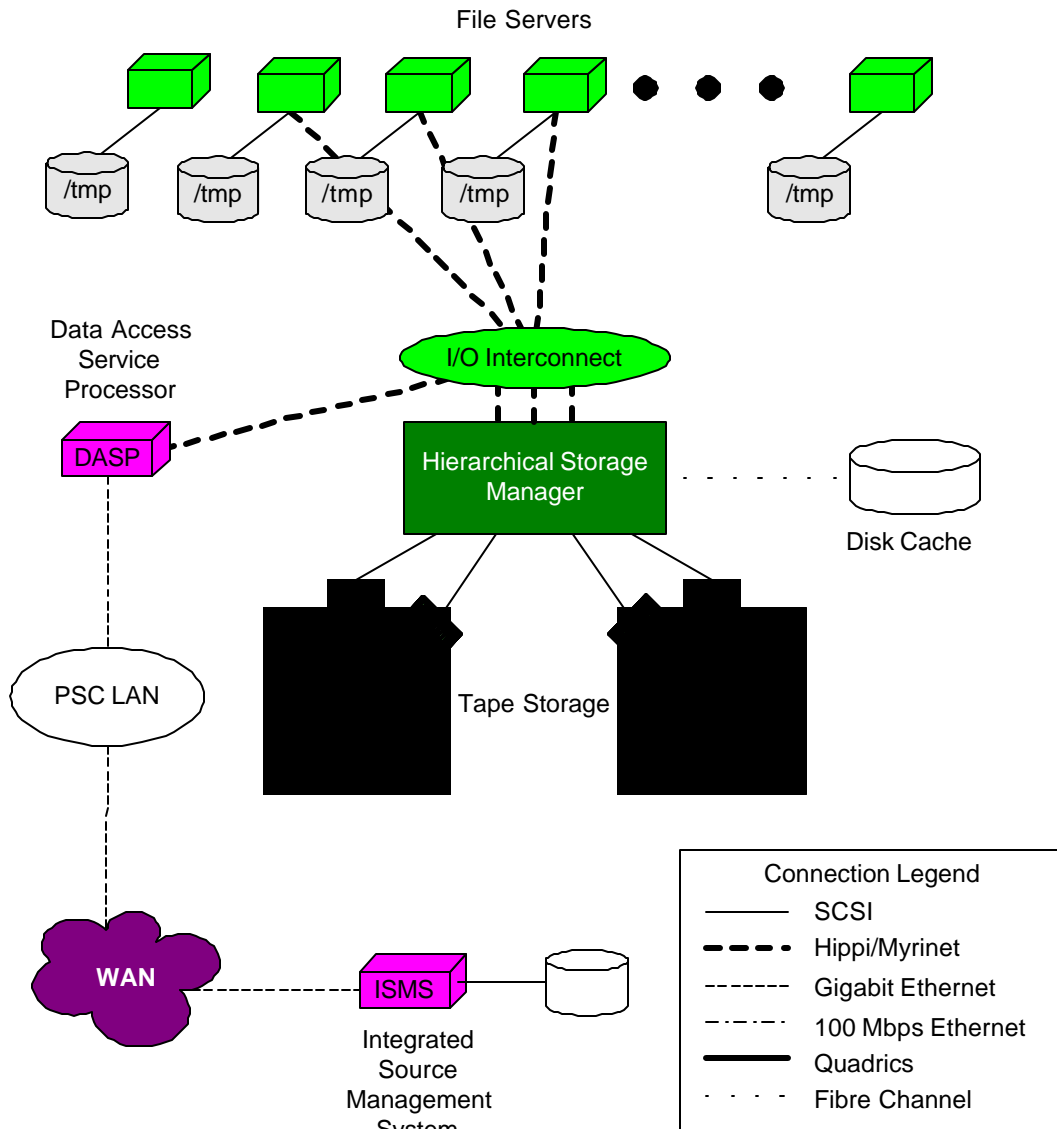


Figure 2: TCS storage architecture

## Hardware Performance

In prototyping the eventual layout of the file systems for the TCS machine, PSC has been engaging in performance testing and optimization of file system components and configurations. The storage capacities involved require a large component count that again necessitates robustness to allow for failures. However robustness added to the storage infrastructure in the form of Redundant Array of Inexpensive Disk (RAID) controllers constrains the performance of the storage subsystem. In many cases the aggregate throughput of an entire robust system is similar to that of a single raw drive. Experimentation and tuning of both software and hardware RAID technologies provides realistic expectations of the performance of robust storage.

We have been testing optimized logical volume definitions in an effort to establish the availability of low-, medium-, and high-bandwidth logical volumes utilizing the same disks. At the disk level, the sustainable data rate of individual drives varies on the physical location of the data. By intelligently allocating logical volumes on a drive, one can create logical volumes of differing performance. By writing portions of data to each of similarly performing volumes on multiple devices—“striping” the data—one can expect to leverage the performance characteristics of a group of volumes. Aligning each of these groups of volumes with different performance requirements within the system allows a better utilization of resources. This would allow checkpoint data, for example, to be dumped expediently to the highest performance volume group and the remainder of the same devices to be used for less demanding operations.

There are often significant differences between the published capabilities of raw devices and the aggregate throughput of a RAID system of these devices. The experimentation with both hardware and software RAID implementations is providing a realistic point of reference from which to qualify the published performance information on storage devices. Furthermore intelligent configuration is expected to enable the most effective and efficient usage of the storage infrastructure.

### **Remote Data Access**

The HSM will support connectivity to the WAN via a Data Access Service Processor (DASP). The DASP will be a dedicated node, responsible for providing HTTP and “direct” secure transport endpoints to remote users. Users who wish to migrate large data files (*e.g.* source code archives, visualization files, computational snapshots) into or out of the TCS machine and HSM must do so through the DASP. One potential software candidate we are investigating for the management and interface to data on the DASP is the Storage Resource Broker (SRB) from the San Diego Supercomputing Center (see <http://www.npaci.edu/DICE/SRB/>). SRB utilizes a high-performance database to contain and administer files or other data objects.

Another software candidate we are investigating is a tool called DocShare, developed at PSC. DocShare utilizes the file system, as opposed to a database, in its back-end archive implementation. It provides an interface accessible to remote users via their browser through which they can upload, download, and delete files in a hierarchical directory structure that they can create or modify. All file operations are archived at the server side via Concurrent Versions System (CVS) software, integrated into DocShare. Users can browse and compare previous versions of uploaded (or deleted) source files via CVSweb, also integrated into DocShare. To facilitate the browsing of complex source trees DocShare also contains an integrated version of the Linux Cross Reference (LXR) and Glimpse file system indexer. LXR/Glimpse allows users to search through their uploaded files and directories by directory name, file name, FORTRAN or C specifiers (as parsed by Glimpse), or free text searches. With these tools, the ISMS will provide users a platform independent interface to remotely share and manage their source files.

## **IO Schema**

Users of a system of this size will have diverse, high-performance IO requirements. We endeavor to provide our users with a set of IO tools with sufficient diversity and efficacy to meet their demands. For example, due to the large node count we expect hardware failures within the TCS system with an estimated Mean Time Between Failures (MTBF) of approximately 10 hours. At this rate, checkpointing user jobs for the purpose of automated recovery and restart is a necessity. Most users already have code to write out and read back a sufficient set of variables to restart their jobs in the case of a machine failure. Thus, one requirement is that we must provide the hardware and software infrastructure to enable users to write critical data to “checkpoint” files such that their jobs may be restarted by reading the contents of these files. Our IO and file system framework must provide the availability and robustness such that, once written to disk, these checkpoint files will be available to the restarted job in such a case.

Other examples of demanding user requirements include parallel access to a single file from all processes in a job and access to very large (multi gigabyte) files. In the following we present several different IO schema which we have conceived in an effort to provide our users with the tools they will need on a machine of this type: global/redirected file system access, high-speed checkpointing to local disk coupled with file migration or parity file generation.

### **Global/Redirected File System Access**

The first IO scheme involves a direct write of data in memory to a reliable remote medium. This can be achieved either via a global file system or IO redirection. In the case of a global file system, the user opens a file on a remote volume that is mounted on the compute node and writes to that file in the usual sense. In the case of IO redirection, system calls like `read()` and `write()` are superceded by a custom set of methods (e.g. `tcs_read_()`, `tcs_write_()`, etc.) and the traffic is sent over the interconnect to a remote node which performs the file operations. These approaches are characterized as a “fast drain” since they result in rapid export of in-memory data from compute nodes to reliable storage. These approaches are, however, subject to hardware and configuration constraints like the availability of a high-performance global file system.

In fact, for all IO schema TCS users will have to utilize a custom API, as opposed to the standard system IO library, but we will address this below (see IO Schema User Interface) after explaining the details of each of the IO schema.

### **Local Disk & Migration**

The second IO scheme involves a two-step process: first, writing data from memory directly to files on local disk on each compute node; and second, migrating the files to reliable storage. This is expected to enable the greatest memory-to-disk bandwidth (of order 60 gigabytes/sec across the TCS machine), thus enabling a rapid return to

computation with the least interruption--highly useful, for example, when checkpointing. However, data temporarily resident on local disk is most subject to loss by machine failure, hence the second necessary step of migrating files to reliable storage. If minimizing the interruption of computational time is of greatest importance, then the second step can be performed slowly, in the background. For this reason this approach is characterized as a “slow drain.”

There are two further variations on this scheme: 1) copying to a “neighbor” and 2) moving to a central storage location. In the first variation, compute nodes copy their files to another compute node. In this case recovery from a single-node failure is achieved by simply copying one file from its former designated “neighbor” to a new node that is added to the compute partition to take its place. This has the dual advantages of distributed file traffic at checkpoint time and minimal file traffic preceding a restart.

In the second variation, compute nodes copy their files to one of the file servers, which host large scratch space. This has the following advantages: it provides centralized storage and redundant file serving, a simple and consistent file migration plan for staging files prior to restart, and the ability to recover from multiple simultaneous compute node failures. But it has the disadvantage of being a bottleneck at the time of file migration and during restart, since multiple nodes must obtain their checkpoint files from a single, particular file server.

### **Local Disk & Parity**

The third IO scheme exploits the availability of local disk to maximize write bandwidth and minimize traffic over the QSW system interconnect to the shared file systems. In addition, this provides fault tolerance by generating a parity file for sets of processors. These parity files are written to a shared file system to enable recovery if one or more nodes in the compute partition fail. Asynchronous I/O is used to overlap the computation of the parity file with the write operation to the local disk. Multiple versions of checkpoint and parity files are maintained to avoid data corruption by truncated writes. The libraries handle the naming, location and creation of the checkpoint and parity files. Making a number of library write calls saves program state information. Each write call results in three distinct operations:

- Asynchronous write of the checkpoint file: User arrays are written to the checkpoint files on local disk using POSIX compliant system aio libraries.
- Computation of bit-wise xor using MPI collectives: While the asynchronous write of the checkpoint file is in progress, the PEs perform a collective bit-wise xor on the array across a set of processors. The processor set is configurable by the user and determines the number of parity files that are generated.
- Write of the parity file: Parity data is written to the shared file system using aio library calls.

Writing to the local disk reduces the load on the shared file system and reduces traffic on the QSW system interconnect. This will ensure greater aggregate write bandwidth from memory to disk. Usage of asynchronous IO allows the idle processors to compute and

write the parity files thereby increasing the resiliency to node failures. A recovery utility can then regenerate missing files automatically enabling a restart.

## IO Schema User Interface

The IO schema will be encapsulated behind a single Application Program Interface (API). This API is presented below.

```
//
// TCS user client library
//
// Authors: Nathan Stone
//
// Copyright 1999, 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 make any representations about
// the suitability of this software for any purpose. It is provided
// "as is" without express or implied warranty.
//
// $Id: tcsusr.h,v 1.5 2001/01/10 14:43:03 nstone Exp $
//

#ifdef _tcsusr_h
#define _tcsusr_h

#include <sys/types.h>

#ifdef __cplusplus
extern "C" {
#endif // __cplusplus

// initialize the TCS library
// - size => # of processors in the job
// - rank => processor number (e.g. "N" of "size")
int tcs_init_(int *rank, int *size);

// open a checkpoint file
// - prefix => prefix for checkpoint files (alpha-numeric only)
//   (!! no two open checkpoint files can have the same prefix !!)
// - on success returns a logical unit number (LUN) which is used as
//   an input argument to read/write/close
// - on failure returns -1
int tcs_open_write_(char *prefix);

// open a checkpoint file
// - prefix => prefix for checkpoint files (alpha-numeric only)
//   (!! no two open checkpoint files can have the same prefix !!)
```

```

// - rsrcid => resource ID of the job which generated the checkpoint
files
// - series => sequence number of the timestep to be recovered
// - on success returns a logical unit number (LUN) which is used as
//   an input argument to read/write/close
// - on failure returns -1
int tcs_open_read(char *prefix, long *rsrcid, int *series);

// close a checkpoint file
// - lun => logical unit returned from tcs_open_
int tcs_close_(int *lun);

// read from a checkpoint file
// - lun => logical unit returned from tcs_open_
// - A => pointer to data buffer
// - len => number of elements to be read
// - datasize => size of single element
int tcs_read_(int *lun, void *A, int *len, int *datasize);

// write arbitrary types to a checkpoint file
// - lun => logical unit returned from tcs_open_
// - A => pointer to data buffer
// - len => number of elements to be written
// - datasize => size of single element
int tcs_write(int *lun, void *A, const int *len, uint *datasize);

// terminate checkpoint operations
// - reapedays => number of days before checkpoint files will be deleted
int tcs_finalize_(int *reapedays);

#ifdef __cplusplus
}
#endif // __cplusplus

#endif // _tcsusr_h

```

This interface will make these schema easily accessible to FORTRAN, C, and C++ user codes via a small number of source code modifications. All implementations are hidden behind this same interface, selected via the TCS\_PLAN environment variable set by the user at job execution time. Certain machine-dependent configuration parameters (*e.g.* checkpoint directories) are defined in a host-based configuration file (*e.g.* /etc/tcsd.conf), while other job-dependent parameters (*e.g.* machine resource ID) will be defined by the scheduling system.

This custom IO library will utilize the input parameters from the scheduling system, the configuration files, and the user's environment to perform some necessary internal initializations when the user invokes the `tcs_init_()` method. The rank/size of the processor/job are as defined in the header file. This information is used in the file naming and storage conventions. The library will release all remaining resources when the user invokes the `tcs_finalize_()` method. The `reapedays` parameter allows users to set a non-zero number of days (up to a administratively-defined maximum) during which the files will be preserved, in the case that users wish to analyze the contents of these files.

This API provides two different open methods, `tcs_open_write_()` and `tcs_open_read_()`, the former is intended for writing checkpoint files during a normal job while the latter is intended for use by a job which restarts in checkpoint recovery mode. In such a case, the user must specify the resource ID of the failed job from which the current job is to recover and also the series (time step) from which to recover. (Ultimately the checkpoint recovery software will most likely automatically extract this latter parameter.) All checkpoint files are closed via the `tcs_close_()` method. Under certain schema, the close operation also triggers the migration of a completed checkpoint or XOR file.

Finally, the `tcs_read_()` and `tcs_write_()` methods encapsulate the IO methods implemented in each scheme. For example, in the first IO scheme (“IO Redirection”) the normal system read/write calls are supplanted by send/receive calls that send the data out over the high-speed interconnect and read/write to a remote file system. In the third IO scheme (“XOR Calculation”) the write calls become a combination of an asynchronous write and simultaneous XOR calculation across some predetermined number of nodes. The selection between these plans is a feature of the checkpoint library, controlled by the user’s environment.

### **Scheduling and Accounting**

AlphaServer SC also includes the Resource Management System (RMS), a uniform resource allocation, control, and monitoring interface. RMS provides the tools necessary to make the entire system appear, for purposes of job scheduling and resource management, like a single NUMA system.

The Resource Management System (RMS) provides a framework for defining compute partitions and managing the composition of these partitions, including dynamic reconfiguration of partitions to exclude failed nodes and include replacements for those nodes. RMS also monitors the status of hardware resources and maintains an accounting database for active and completed user jobs. PSC has integrated the Portable Batch System (PBS) and RMS in order to provide queuing and job monitoring services to our users.

The TCS will provide two RMS “partitions” for users: a small “login” partition and a large “parallel” partition. The login partition will provide TCS users an interactive shell from which they will edit and submit jobs to the parallel partition. This partition is expected to be no more than a few nodes in size. The parallel partition, which encompasses the remainder of the machine, will be further subdivided into two logical partitions: a small "interactive" partition and a large "batch" partition. The "interactive" partition will be available (via the “`qsub -i`” command) primarily during the day to provide a resource for users to debug their programs. This partition will be on the order of 32 nodes, and used for testing small programs that run only for short periods of time. The "batch" logical partition will be used (via the “`qsub`” command) exclusively for jobs that utilize the majority of the machine and run for long periods of time. The sizes of the

two logical partitions will be dynamically adjustable, and will depend on usage and need, with the batch partition occasionally using all of the nodes of the parallel partition for maximum batch job size.

### **Remote Monitoring**

PSC has a long history of providing scientific computing services to a pool of almost exclusively remote users. As such we have invested significant effort to ensure that our users can efficiently access the machine not only for job scheduling and control, as previously mentioned, but also for monitoring purposes. For example, PSC has a set of near real-time monitoring tools available via users' web browsers. Users with direct access to the system will, of course, have access to job-, queue-, and machine-oriented status tools. While we have authored our own tools, where appropriate, to deliver information about running jobs, machine parameters, and storage availability, we are also investigating other third party tools that may provide a meaningful supplement to our users.

One such example is the NPACI "Hot Page" web interface. Information about our Cray T3E is already available to all NPACI users via this interface, and we expect to provide similar functionality to our TCS machine. Another example is Compaq's "Insight Manager". Insight Manager monitors more than 1,000 management parameters to allow fault prediction and alerting, asset and configuration inventory, and performance monitoring. It can also be configured to automatically send alphanumeric pages in the event of a machine failure. Insight Manager can also maintain a database of serial numbers for each machine and its respective components, and collect CPU, memory and disk utilization for trend analysis. This type of "inventory control" although of primary interest to our support staff, may provide direct benefits to our users as well.