SCR Developer Documentation

Release 1.2.0

SCR

Sep 14, 2023

Concepts

Welcome to the Developers Guide for the Scalable Checkpoint / Restart (SCR) library. This guide contains three main sections to help developers get up to speed with SCR:

Concepts This section defines essential concepts and to provide high-level explanation so that one may better understand the source code.

Files This section covers the contents of many of the files used in SCR. In order to not clutter the user's directory structure on the parallel file system, SCR writes its internal files to hidden "`.scr`" subdirectories. See Section *Example of SCR files and directories* for an example of where these files are written.

Program Flow This section describes high-level program flow of various library routines and commands.

Testing This section describes testing practices related to SCR.

The SCR library enables MPI applications to utilize distributed storage on Linux clusters to attain high file I/O bandwidth for checkpointing, restarting, and writing large datasets. With SCR, jobs run more efficiently, recompute less work upon a failure, and reduce load on shared resources like the parallel file system. It provides the most benefit to large-scale jobs that write large datasets. SCR utilizes tiered storage in a cluster to provide applications with the following capabilities:

- guidance for the optimal checkpoint frequency,
- scalable checkpoint bandwidth,
- scalable restart bandwidth,
- scalable output bandwidth,
- asynchronous data transfers to the parallel file system.

SCR originated as a production-level implementation of a multi-level checkpoint system of the type analyzed by [Vaidya] SCR caches checkpoints in scalable storage, which is faster but less reliable than the parallel file system. It applies a redundancy scheme to the cache such that checkpoints can be recovered after common system failures. It also copies a subset of checkpoints to the parallel file system to recover from less common but more severe failures. In many failure cases, a job can be restarted from a checkpoint in cache, and writing and reading datasets in cache can be orders of magnitude faster than the parallel file system.

When writing a cached dataset to the parallel file system, SCR can transfer data asynchronously. The application may continue once the data has been written to the cache while SCR copies the data to the parallel file system in the background. SCR supports general output datasets in addition to checkpoint datasets.

SCR consists of two components: a library and a set of commands. The application registers its dataset files with the SCR API, and the library maintains the dataset cache. The SCR commands are typically invoked from the job batch script. They are used to prepare the cache before a job starts, automate the process of restarting a job, and copy datasets from cache to the parallel file system upon a failure.

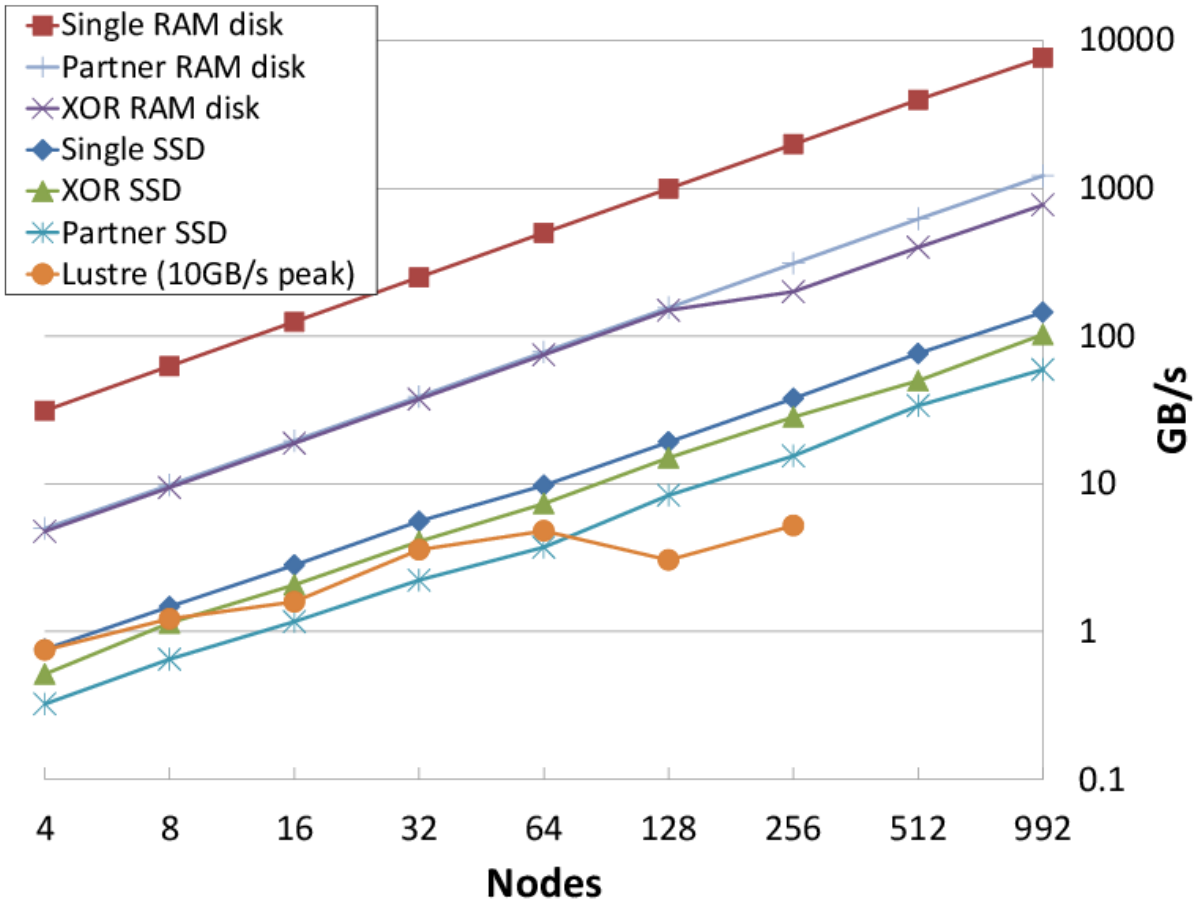

Fig. 1: Aggregate write bandwidth on Coastal

CHAPTER 1

Support

The main repository for SCR is located at:

https://github.com/LLNL/scr.

From this site, you can download the source code and manuals for the current release of SCR.

For information about the project including active research efforts, please visit:

https://computation.llnl.gov/project/scr

To contact the developers of SCR for help with using or porting SCR, please visit:

https://computation.llnl.gov/project/scr/contact.php

There you will find links to join our discussion mailing list for help topics, and our announcement list for getting notifications of new SCR releases.

CHAPTER 2

Contents

2.1 Overview

This section covers basic concepts and terms used throughout the SCR documentation and source code.

2.1.1 Intro to the SCR API

This section provides an overview of how one may integrate the SCR API into an application. For a more detailed discussion, please refer to the user manual.

SCR is designed to support MPI applications that write application-level checkpoints, primarily as a file-per-process. In a given checkpoint, each process may actually write zero or more files, but the current implementation assumes that each process writes roughly the same amount of data. The checkpointing code for such applications may look like the following:

```
int main(int argc, char* argv[]) {
  MPI_Init(argc, argv);

  for (t = 0; t < TIMESTEPS; t++) {
    /* ... do work ... */

    /* every so often, write a checkpoint */
    if (t % CHECKPOINT_FREQUENCY == 0)
      checkpoint();
  }

  MPI_Finalize();
  return 0;
}

void checkpoint() {
  /* rank 0 creates a directory on the file system,
   * and then each process saves its state to a file */
```

(continues on next page)

(continued from previous page)

```
  /* get rank of this process */
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* rank 0 creates directory on parallel file system */
  if (rank == 0)
    mkdir(checkpoint_dir);

  /* hold all processes until directory is created */
  MPI_Barrier(MPI_COMM_WORLD);

  /* build file name of checkpoint file for this rank */
  char checkpoint_file[256];
  sprintf(checkpoint_file, "%s/rank_%d".ckpt",
    checkpoint_dir, rank
  );

  /* each rank opens, writes, and closes its file */
  FILE* fs = open(checkpoint_file, "w");
  if (fs != NULL) {
    fwrite(checkpoint_data, ..., fs);
    fclose(fs);
  }
}
```

The following code exemplifies the changes necessary to integrate SCR. Each change is numbered for further discussion below.

```
int main(int argc, char* argv[]) {
  MPI_Init(argc, argv);

  /**** change #1 ****/
  SCR_Init();

  for (t = 0; t < TIMESTEPS; t++) {
    /* ... do work ... */

    /**** change #2 ****/
    int need_checkpoint;
    SCR_Need_checkpoint(&need_checkpoint);
    if (need_checkpoint)
      checkpoint();
  }

  /**** change #3 ****/
  SCR_Finalize();

  MPI_Finalize();
  return 0;
}

void checkpoint() {
  /* rank 0 creates a directory on the file system,
   * and then each process saves its state to a file */
```

(continues on next page)

(continued from previous page)

```
  /**** change #4 ****/
  SCR_Start_checkpoint();

  /* get rank of this process */
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /**** change #5 ****/
  /*
      if (rank == 0)
        mkdir(checkpoint_dir);

      /* hold all processes until directory is created */
      MPI_Barrier(MPI_COMM_WORLD);
  */

  /* build file name of checkpoint file for this rank */
  char checkpoint_file[256];
  sprintf(checkpoint_file, "%s/rank_%d".ckpt",
    checkpoint_dir, rank
  );

  /**** change #6 ****/
  char scr_file[SCR_MAX_FILENAME];
  SCR_Route_file(checkpoint_file, scr_file);

  /**** change #7 ****/
  /* each rank opens, writes, and closes its file */
  FILE* fs = open(scr_file, "w");
  if (fs != NULL) {
    fwrite(checkpoint_data, ..., fs);
    fclose(fs);
  }

  /**** change #8 ****/
  SCR_Complete_checkpoint(valid);
}
```

First, as shown in change #1, one must call `SCR_Init()` to initialize the SCR library before it can be used. SCR uses MPI, so SCR must be initialized after MPI has been initialized. Similarly, as shown in change #3, it is good practice to shut down the SCR library by calling `SCR_Finalize()`. This must be done before calling `MPI_Finalize()`. As shown in change #2, the application may rely on SCR to determine when to checkpoint by calling `SCR_Need_checkpoint()`. SCR can be configured with information on failure rates and checkpoint costs for the particular host platform, so this function provides a portable method to guide an application toward an optimal checkpoint frequency.

Then, as shown in change #4, the application must inform SCR when it is starting a new checkpoint by calling `SCR_Start_checkpoint()`. Similarly, it must inform SCR when it has completed the checkpoint with a corresponding call to `SCR_Complete_checkpoint()` as shown in change #8. When calling `SCR_Complete_checkpoint()`, each process sets the `valid` flag to indicate whether it wrote all of its checkpoint files successfully. SCR manages checkpoint directories, so the `mkdir` operation is removed in change #5.

Between the call to `SCR_Start_checkpoint()` and `SCR_Complete_checkpoint()`, the application must register each of its checkpoint files by calling `SCR_Route_file()` as shown in change #6. SCR "routes" the file by replacing any leading directory on the file name with a path that points to another directory in which SCR caches data for the checkpoint. As shown in change #7, the application must use the exact string returned by `SCR_Route_file()` to open its checkpoint file.

All SCR functions are collective, except for `SCR_Route_file()`. Additionally, SCR imposes the following semantics:

1. A process of a given MPI rank may only access files previously written by itself or by processes having the same MPI rank in prior runs. We say that a rank "owns" the files it writes. A process is never guaranteed access to files written by other MPI ranks.
2. During a checkpoint, a process may only access files of the current checkpoint between calls to `SCR_Start_checkpoint()` and `SCR_Complete_checkpoint()`. Once a process calls `SCR_Complete_checkpoint()` it is no longer guaranteed access to any file it registered during that checkpoint via a call to `SCR_Route_file()`.
3. During a restart, a process may only access files from its "most recent" checkpoint, and it must access those files between calls to `SCR_Init()` and `SCR_Start_checkpoint()`. That is, a process cannot access its restart files until it calls `SCR_Init()`, and once it calls `SCR_Start_checkpoint()`, it is no longer guaranteed access to its restart files. SCR selects which checkpoint is considered to be the "most recent".

These semantics enable SCR to cache checkpoint files on devices that are not globally visible to all processes, such as node-local storage. Further, these semantics enable SCR to move, reformat, or delete checkpoint files as needed, such that it can manage this cache, which may be small.

2.1.2 Jobs, allocations, and runs

A large-scale simulation often must be restarted multiple times in order to run to completion. It may be interrupted due to a failure, or it may be interrupted due to time limits imposed by the resource scheduler. We use the term *allocation* to refer to an assigned set of compute resources that are available to the user for a period of time. A resource manager typically assigns an id to each resource allocation, which we refer to as the *allocation id*. SCR uses the allocation id in some directory and file names. Within an allocation, a user may execute a simulation one or more times. We call each execution a *run*. For MPI applications, each run corresponds to a single invocation of `mpirun` or its equivalent. Finally, multiple allocations may be required to complete a given simulation. We refer to this series of one or more allocations as a *job*. To summarize, one or more runs occur within an allocation, and one or more allocations occur within a job.

2.1.3 Group, store, and redundancy descriptors

SCR duplicates `MPI_COMM_WORLD` and stores a copy in `scr_comm_world`. Each process also caches its rank and the size of `scr_comm_world` in `scr_my_rank_world` and `scr_ranks_world`, respectively. This communicator is created during `SCR_Init()`, and it is freed during `SCR_Finalize()`. The variables are defined in `scr_globals.h` and declared and initialized in `scr_globals.c`.

The SCR library must group processes of the parallel job in various ways. For example, if power supply failures are common, it is necessary to identify the set of processes that share a power supply. Similarly, it is necessary to identify all processes that can access a given storage device, such as an SSD mounted on a compute node. To represent these groups, the SCR library uses a *group descriptor*. Details of group descriptors are given in Section *Group descriptors*.

The library creates two groups by default: `NODE` and `WORLD`. The `NODE` group consists of all processes on the same compute node, and `WORLD` consists of all processes in the run. The user or system administrator can create additional groups via configuration files.

The SCR library must also track details about each class of storage it can access. For each available storage class, SCR needs to know the associated directory prefix, the group of processes that share a device, the capacity of the device, and other details like whether the associated file system can support directories. SCR tracks this information in a *store descriptor*. Each store descriptor refers to a group descriptor, which specifies how processes are grouped with respect to that class of storage. Group descriptors must exist before the store descriptors can be created. For a given storage class, it is assumed that all compute nodes refer to the class using the same directory prefix. Each store descriptor is referenced by its directory prefix. Details of store descriptors are given in Section *Store descriptors*.

The library creates one store descriptor by default: `/tmp`. The assumption is made that `/tmp` is mounted as a local file system on each compute node. On Linux clusters, `/tmp` is often RAM disk or a local hard drive. Additional store descriptors can be defined by the user or system administrator in configuration files.

Finally, SCR defines *redundancy descriptors* to associate a redundancy scheme with a class of storage and a group of processes that are likely to fail at the same time. It also tracks details about the particular redundancy scheme used, and the frequency with which it should be applied. Redundancy descriptors reference both store and group descriptors, so these must exist before the SCR library creates its internal redundancy descriptors. Details about redundancy descriptors are given in Section *Redundancy descriptors*.

The library creates a default redundancy descriptor. It assumes that processes on the same node are likely to fail at the same time. It also assumes that checkpoints can be cached in `/tmp`, which is assumed to be storage local to each compute node. It applies an `XOR` redundancy scheme using a group size of 8. Additional redundancy descriptors may be defined by the user or system administrator in configuration files.

All of these descriptors (group, store, and redundancy) are defined by the system administrator or user in system or user configuration files. Additionally, some predefined descriptors are created by the library.

2.1.4 Control, cache, and prefix directories

SCR manages numerous files and directories to cache checkpoint data and to record its internal state. There are three fundamental types of directories: control, cache, and prefix directories. For a detailed illustration of how these files and directories are arranged, see the example presented in Section *directories_example*.

The *control directory* is where SCR writes files to store its internal state about the current run. This directory is expected to be stored in node-local storage. SCR writes multiple, small files in the control directory, and it may access these files frequently. It is best to configure this directory to be stored in a node-local RAM disk.

To construct the full path of the control directory, SCR incorporates a control base directory name along with the user name and allocation id associated with the resource allocation, such that the control directory is of the form: `<controlbase>/<username>/scr.<allocationid>`. This enables multiple users, or multiple jobs by the same user, to run at the same time without conflicting for the same control directory. The control base directory is hard-coded into the SCR library at configure time, but this value may be overridden via a system configuration file. The user may not change the control base directory.

SCR directs the application to write checkpoint files to subdirectories within a *cache directory*. SCR also stores its redundancy data files in these subdirectories. The device serving the cache directory must be large enough to hold the data for one or more checkpoints plus the associated redundancy data. Multiple cache directories may be utilized in the same run, which enables SCR to use more than one class of storage within a run (e.g., RAM disk and SSD). Cache directories should be located on scalable storage.

To construct the full path of a cache directory, SCR incorporates a cache base directory name with the user name and allocation id associated with the resource allocation, such that the cache directory is of the form: `<cachebase>/<username>/scr.<allocationid>`. A set of valid cache base directories is hard-coded into the SCR library at configure time, but this set can be overridden in a system configuration file. Out of this set, the user may select a subset of cache base directories that should be used during a run. A cache directory may be the same as the control directory.

The user must configure the maximum number of checkpoints that SCR should keep in each cache directory. It is up to the user to ensure that the capacity of the device associated with the cache directory is large enough to hold the specified number of checkpoints.

For now, SCR only handles checkpoint output sets. However, there are plans to eventually support general, non-checkpoint output sets. SCR refers to each application output set as a *dataset*, where a checkpoint is a dataset having a certain property, namely that it can be used to restart the simulation. SCR assigns a unique sequence number to each dataset called the *dataset id*. SCR also assigns a unique sequence number to each checkpoint called the *checkpoint id*. It assigns dataset ids starting from 1 and counts up with each successive dataset written by the application. Similarly,

it starts checkpoint ids from 1 and increments the checkpoint id with each successive checkpoint. The library manages these counters in the `scr_dataset_id` and `scr_checkpoint_id` global variables defined in `scr.c`.

Within a cache directory, a dataset is written to its own subdirectory called the *dataset directory*. SCR associates each dataset with a *redundancy descriptor*. The redundancy descriptor describes which cache directory should be used, which redundancy scheme to apply, and how often it should be used (Section *Redundancy descriptors*). A single run employs a set of one or more redundancy descriptors, and each descriptor is assigned a unique integer index counting up from 0. When starting a new checkpoint, SCR selects a redundancy descriptor, and then it creates a corresponding dataset directory within the cache directory. The full path of the dataset directory is of the form: `<cachebase>/<username>/scr.<allocationid>/scr.dataset.<datasetid>`.

Finally, the *prefix directory* is a directory on the parallel file system that the user specifies. SCR copies datasets to the prefix directory for permanent storage (Section *Fetch, Flush, and scavenge*). The prefix directory should be accessible from all compute nodes, and the user must ensure that the prefix directory name is unique for each job. For each dataset stored in the prefix directory, SCR creates and manages a *dataset directory*. The dataset directory holds all files associated with a particular dataset, including application files and SCR redundancy files. SCR maintains an index file within the prefix directory, which records information about each dataset stored there.

Note that the term "dataset directory" is overloaded. In some cases, we use this term to refer to a directory in cache and in other cases we use the term to refer to a directory within the prefix directory on the parallel file system. In any particular case, the meaning should be clear from the context.

2.1.5 Example of SCR files and directories

To illustrate how various files and directories are arranged in SCR, consider the example shown in Figure *Example of SCR Directories*. In this example, a user named "`user1`" runs a 4-task MPI job with one task per compute node. The base directory for the control directory is `/tmp`, the base directory for the cache directory is `/ssd`, and the prefix directory is `/p/lscratchb/user1/simulation123`. The control and cache directories are storage devices local to the compute node.

The full path of the control directory is `/tmp/user1/scr.1145655`. This is derived from the concatenation of the base directory (`/tmp`), the user name (`user1`), and the allocation id (`1145655`). SCR keeps files to persist its internal state in the control directory, including filemap files (Section *Filemap files*) and the transfer file (Section *Transfer file*).

Similarly, the cache directory is `/ssd/user1/scr.1145655`, which is derived from the concatenation of the cache base directory (`/ssd`), the user name (`user1`), and the allocation id (`1145655`). Within the cache directory, SCR creates a subdirectory for each dataset. The dataset directory name includes the dataset id. In this example, there are two datasets currently stored in cache, (`scr.dataset.17` and `scr.dataset.18`). The application dataset files and SCR redundancy files are stored within their corresponding dataset directory. On the node running MPI rank 0, there is one application dataset file (`rank_0.ckpt`) and one XOR redundancy data file (`1_of_4_in_0.xor`).

Finally, the full path of the prefix directory is `/p/lscratchb/user1/simulation123`. This is a path on the parallel file system that is specified by the user, and it is unique to the particular simulation the user is running (`simulation123`). The prefix directory contains dataset directories. It also contains a hidden `.scr` directory where SCR writes the index file to record info for each of the datasets (Section *Index file*). The SCR library also writes the halt file (Section *Halt file*), the flush file (Section *Flush file*), and the nodes file (Section *Nodes file*) to the hidden `.scr` directory.

While the user provides the prefix directory, SCR defines the name of each dataset directory to be "`scr.dataset.<id>`" where `<id>` is the dataset id. In this example, there are multiple datasets stored on the parallel file system (corresponding to dataset ids 10, 12, and 18).

Within each dataset directory, SCR stores the files written by the application. SCR also creates a hidden `.scr` subdirectory, and this hidden directory contains redundancy files and filemap files. SCR also stores a summary file (Section *Summary file*) and rank2file map files (Section *Rank2file map*) in which it records information needed to fetch files from the directory in order to restart the job from the checkpoint.

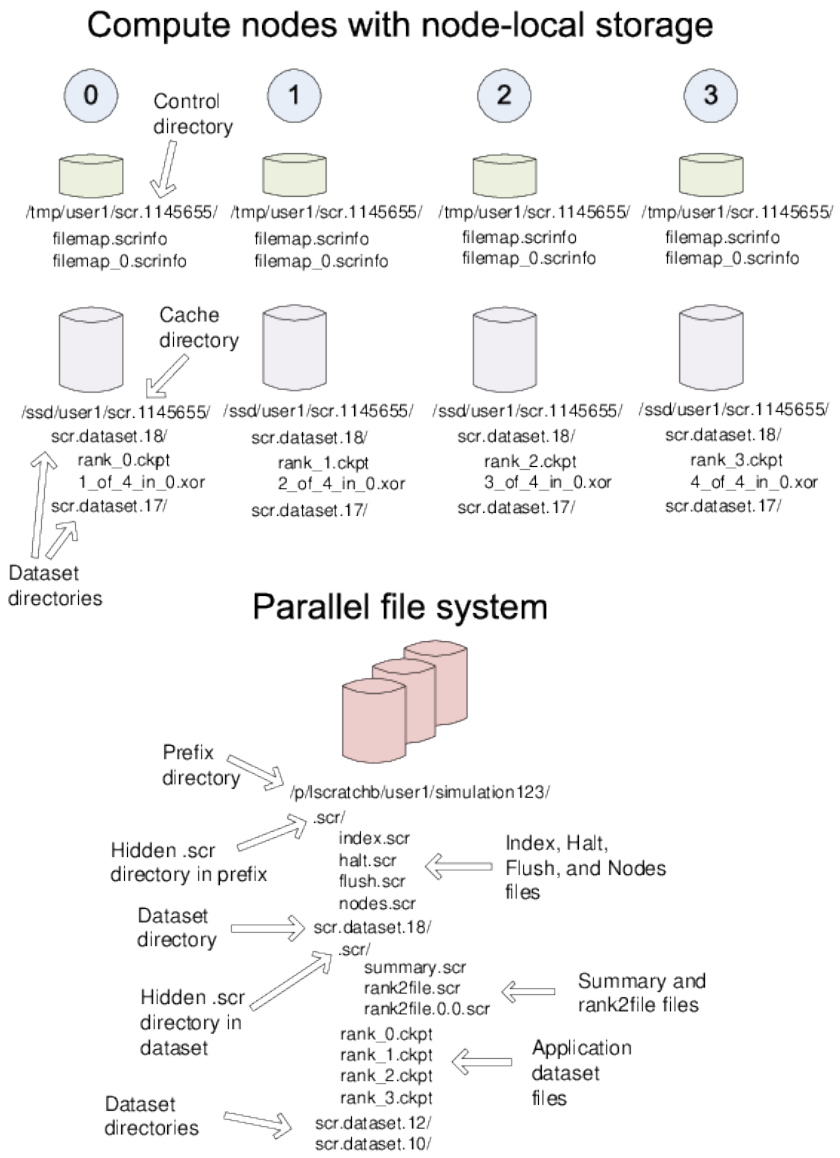

Fig. 1: Example of SCR Directories

2.1.6 Scalable checkpoint

In practice, it is common for multiple processes to fail at the same time, but most often this happens because those processes depend on a single, failed component. It is not common for multiple, independent components to fail simultaneously. By expressing the groups of processes that are likely to fail at the same time, the SCR library can apply redundancy schemes to withstand common, multi-process failures. We refer to a set of processes likely to fail at the same time as a *failure group*.

SCR must also know which groups of processes share a given storage device. This is useful so the group can coordinate its actions when accessing the device. For instance, if a common directory must be created before each process writes a file, a single process can create the directory and then notify the others. We refer to a set of processes that share a storage device as a *storage group*.

Given knowledge of failure and storage groups, the SCR library implements three redundancy schemes which trade off performance, storage space, and reliability:

`Single` - each checkpoint file is written to storage accessible to the local process

`Partner` - each checkpoint file is written to storage accessible to the local process, and a full copy of each file is written to storage accessible to a partner process from another failure group

`XOR` - each checkpoint file is written to storage accessible to the local process, XOR parity data are computed from checkpoints of a set of processes from different failure groups, and the parity data are stored among the set.

With `Single`, SCR writes each checkpoint file in storage accessible to the local process. It requires sufficient space to store the maximum checkpoint file size. This scheme is fast, but it cannot withstand failures that disable the storage device. For instance, when using node-local storage, this scheme cannot withstand failures that disable the node, such as when a node loses power or its network connection. However, it can withstand failures that kill the application processes but leave the node intact, such as application bugs and file I/O errors.

With `Partner`, SCR writes checkpoint files to storage accessible to the local process, and it also copies each checkpoint file to storage accessible to a partner process from another failure group. This scheme is slower than `Single`, and it requires twice the storage space. However, it is capable of withstanding failures that disable a storage device. In fact, it can withstand failures of multiple devices, so long as a device and the device holding the copy do not fail simultaneously.

With `XOR`, SCR defines sets of processes where members within a set are selected from different failure groups. The processes within a set collectively compute XOR parity data which is stored in files along side the application checkpoint files. This algorithm is based on the work found in [Ross], which in turn was inspired by [RAID5]. This scheme can withstand multiple failures so long as two processes from the same set do not fail simultaneously.

Computationally, `XOR` is more expensive than `Partner`, but it requires less storage space. Whereas `Partner` must store two full checkpoint files, `XOR` stores one full checkpoint file plus one XOR parity segment, where the segment size is roughly $1/(N-1)$ times the size of a checkpoint file for a set of size N. Larger sets demand less storage, but they also increase the probability that two processes in the same set will fail simultaneously. Larger sets may also increase the cost of recovering files in the event of a failure.

2.1.7 Scalable restart

So long as a failure does not violate the redundancy scheme, a job can restart within the same resource allocation using the cached checkpoint files. This saves the cost of writing checkpoint files out to the parallel file system only to read them back during the restart. In addition, SCR provides support for the use of spare nodes. A job can allocate more nodes than it needs and use the extra nodes to fill in for any failed nodes during a restart. SCR includes a set of scripts which encode much of the restart logic (Section *Perl modules*).

Upon encountering a failure, SCR relies on the MPI library, the resource manager, or some other external service to kill the current run. After the run is killed, and if there are sufficient healthy nodes remaining, the same job can

be restarted within the same allocation. In practice, such a restart typically amounts to issuing another "`srun`" or "`mpirun`" in the job batch script.

Of the set of nodes used by the previous run, the restarted run should use as many of the same nodes as it can to maximize the number of files available in cache. A given MPI rank in the restarted run does not need to run on the same node that it ran on in the previous run. SCR distributes cached files among processes according to the process mapping of the restarted run.

By default, SCR inspects the cache for existing checkpoints when a job starts. It attempts to rebuild all datasets in cache, and then it attempts to restart the job from the most recent checkpoint. If a checkpoint fails to rebuild, SCR deletes it from cache. To disable restarting from cache, set the `SCR_DISTRIBUTE` parameter to 0. When disabled, SCR deletes all files from cache and restarts from a checkpoint on the parallel file system.

An example restart scenario is illustrated in Figure *1* in which a 4-node job using the `Partner` scheme allocates 5 nodes and successfully restarts within the allocation after a node fails.

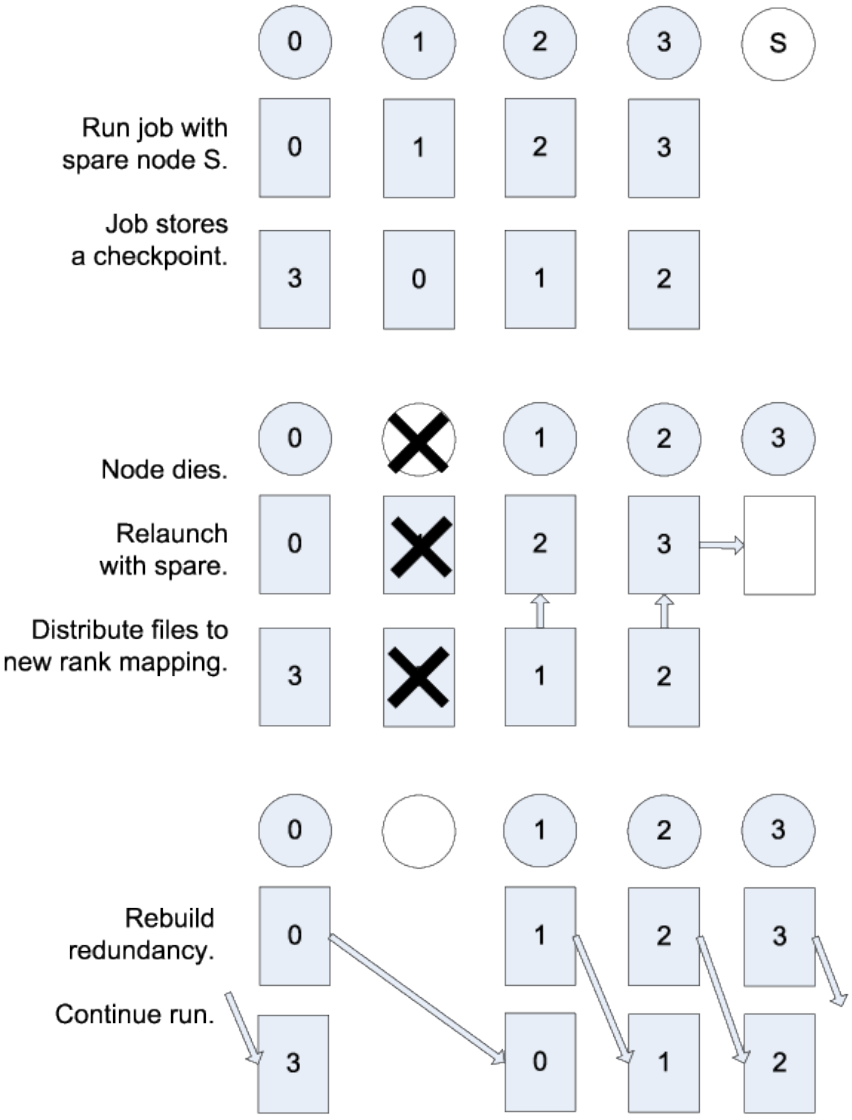

Fig. 2: Example restart after a failed node with `Partner`

2.1.8 Catastrophic failures

There are some failures from which the SCR library cannot recover. In such cases, the application is forced to fall back to the latest checkpoint successfully written to the parallel file system. Such catastrophic failures include the following:

Multiple node failure which violates the redundancy scheme. If multiple nodes fail in a pattern which violates the cache redundancy scheme, data are irretrievably lost.

Failure during a checkpoint. Due to cache size limitations, some applications can only fit one checkpoint in cache at a time. For such cases, a failure may occur after the library has deleted the previous checkpoint but before the next checkpoint has completed. In this case, there is no valid checkpoint in cache to recover.

Failure of the node running the job batch script. The logic at the end of the allocation to scavenge the latest checkpoint from cache to the parallel file system executes as part of the job batch script. If the node executing this script fails, the scavenge logic will not execute and the allocation will terminate without copying the latest checkpoint to the parallel file system.

Parallel file system outage. If the application fails when writing output due to an outage of the parallel file system, the scavenge logic may also fail when it attempts to copy files to the parallel file system.

There are other catastrophic failure cases not listed here. Checkpoints must be written to the parallel file system with some moderate frequency so as not to lose too much work in the event of a catastrophic failure. Section *Fetch, Flush, and scavenge* provides details on how to configure SCR to make occasional writes to the parallel file system.

By default, the current implementation stores only the most recent checkpoint in cache. One can change the number of checkpoints stored in cache by setting the `SCR_CACHE_SIZE` parameter. If space is available, it is recommended to increase this value to at least 2.

2.1.9 Fetch, flush, and scavenge

SCR manages the transfer of datasets between the prefix directory on the parallel file system and the cache. We use the term *fetch* to refer to the action of copying a dataset from the parallel file system to cache. When transferring data in the other direction, there are two terms used: *flush* and *scavenge*. Under normal circumstances, the library directly copies files from cache to the parallel file system, and this direct transfer is known as a flush. However, sometimes a run is killed before the library can complete this transfer. In these cases, a set of SCR commands is executed after the final run to ensure that the latest checkpoint is copied to the parallel file system before the current allocation expires. We say that these scripts scavenge the latest checkpoint.

Furthermore, the library supports two types of flush operations: *synchronous* and *asynchronous*. We say a flush is synchronous when the library blocks the application until the flush has completed. SCR also supports a flush in which the library starts the transfer but immediately returns control to the application. An external mechanism (e.g., another process) copies the dataset to the parallel file system in the background. At some later point, the library checks to verify that the transfer has completed. We say this type of flush is asynchronous.

Each time an SCR job starts, SCR first inspects the cache and attempts to distribute files for a scalable restart as discussed in Section Scalable restart <restart. If the cache is empty or the distribute operation fails or is disabled, SCR attempts to fetch a checkpoint from the prefix directory to fill the cache. SCR reads the index file and attempts to fetch the most recent checkpoint, or otherwise the checkpoint that is marked as current within the index file. For a given checkpoint, SCR records whether the fetch attempt succeeds or fails in the index file. SCR does not attempt to fetch a checkpoint that is marked as being incomplete nor does it attempt to fetch a checkpoint for which a previous fetch attempt has failed. If SCR attempts but fails to fetch a checkpoint, it prints an error and continues the run.

To disable the fetch operation, set the `SCR_FETCH` parameter to 0. If an application disables the fetch feature, the application is responsible for reading its checkpoint set directly from the parallel file system upon a restart.

To withstand catastrophic failures, it is necessary to write checkpoint sets out to the parallel file system with some moderate frequency. In the current implementation, the SCR library writes a checkpoint set out to the parallel file

system after every 10 checkpoints. This frequency can be configured by setting the SCR_FLUSH parameter. When this parameter is set, SCR decrements a counter with each successful checkpoint. When the counter hits 0, SCR writes the current checkpoint set out to the file system and resets the counter to the value specified in SCR_FLUSH. SCR preserves this counter between scalable restarts, and when used in conjunction with SCR_FETCH, it also preserves this counter between fetch and flush operations such that it is possible to maintain periodic checkpoint writes across runs. Set SCR_FLUSH to 0 to disable periodic writes in SCR. If an application disables the periodic flush feature, the application is responsible for writing occasional checkpoint sets to the parallel file system.

By default, SCR computes and stores a CRC32 checksum value for each checkpoint file during a flush. It then uses the checksum to verify the integrity of each file as it is read back into cache during a fetch. If data corruption is detected, SCR falls back to fetch an earlier checkpoint set. To disable this checksum feature, set the SCR_CRC_ON_FLUSH parameter to 0.

2.1.10 Configuration parameters

As detailed in the user manual, there are many configuration parameters for SCR. To read the value of a parameter, the SCR library and SCR commands that are written in C invoke the scr_param API which is defined in scr_param.h and implemented in scr_param.c. SCR commands that are written in PERL acquire parameter values through the scr_param.pm PERL module (Section *scripts/common/scr_param.pm.in*). Through either interface, SCR returns the first setting it finds for a parameter, searching in the following order:

1. Environment variables,
2. User configuration file,
3. System configuration file,
4. Default settings.

The user is not able to set some parameters. For these parameters, any setting specified via an environment variable or user configuration file is ignored.

When the library initializes the scr_param interface in an MPI job, rank 0 reads the configuration files (if they exist) and broadcasts the settings to all other processes through the scr_comm_world communicator. Thus, scr_comm_world must be defined before initializing the scr_param interface.

2.1.11 Global variables and portability

All global variables are declared in scr_globals.h and defined in scr_globals.c.

Most of the SCR library code uses basic C routines, POSIX functions, and MPI calls. It is written to be easily portable from one system to another. Code which is different from system to system should be abstracted behind a function and moved to scr_env.h/c. This practice simplifies the task of porting SCR to new systems.

2.2 File paths

2.2.1 Overview

The SCR library manipulates file and directory paths. To simplify this task, it uses the scr_path data structure. There are a number of functions to manipulate paths, including combining, slicing, simplification, computing relative paths, and converting to/from character strings.

This object stores file paths as a linked list of path components, where a component is a character string separated by '/' symbols. An empty string is a valid component, and they are often found as the first component in an absolute path, as in "/hello/world", or as the last component in a path ending with '/'. Components are indexed starting at 0.

2.2.2 Common functions

This section lists the most common functions used when dealing with paths. For a full listing, refer to comments in `scr_path.h`. The implementation can be found in `scr_path.c`.

Creating and freeing path objects

First, before using a path, one must allocate a path object.

```
scr_path* path = scr_path_new();
```

This allocates an empty (or "null") path having 0 components. One must free the path when done with it.

```
scr_path_delete(&path);
```

One may also create a path from a character string.

```
scr_path* path = scr_path_from_str("/hello/world");
```

This splits the path into components at '/' characters. In this example, the resulting path would have three components, consisting of the empty string, "hello", and "world". One can construct a path from a formatted string.

```
scr_path* path = scr_path_from_strf("/%s/%s/%d", dir1, dir2, id);
```

Or to make a full copy of a path as path2.

```
scr_path* path2 = scr_path_dup(path);
```

Querying paths and converting them to character string

One can determine the number of components in a path.

```
int components = scr_path_components(path);
```

A shortcut is available to identify a "null" path (i.e., a path with 0 components).

```
int is_null_flag = scr_path_is_null(path);
```

This function returns 1 if the path has 0 components and 0 otherwise. You can determine whether a path is absolute.

```
int is_absolute_flag = scr_path_is_absolute(path);
```

This returns 1 if the path starts with an empty string and 0 otherwise. The character representation of such a path starts with a '/' character or otherwise it is the empty string.

To get the number of characters in a path.

```
size_t len = scr_path_strlen(path);
```

This count includes '/' characters, but like the `strlen` function, it excludes the terminating NULL character.

One can convert a path and return it as a newly allocated character string.

```
char* str = scr_path_strdup(path);
scr_free(&str);
```

The caller is responsible for freeing the returned string.

Or one can copy the path into a buffer as a character string.

```
char buf[bufsize];
scr_path_strcpy(buf, bufsize, path);
```

Combining paths

There are functions to prepend and append entries to a path. To prepend entries of path2 to path1 (does not affect path2).

```
scr_path_prepend(path1, path2);
```

Similarly to append path2 to path1.

```
scr_path_append(path1, path2);
```

Or one can insert entries of path2 into path1 at an arbitrary location.

```
scr_path_insert(path1, offset, path2);
```

Here `offset` can be any value in the range $[0, N]$ where N is the number of components in `path1`. With an offset of 0, the entries of path2 are inserted before the first component of path1. With an offset of $N - 1$, path2 in inserted before the last component of path1. An offset of N inserts path2 after the last component of path1.

In addition, one may insert a string into a path using functions ending with `_str`, e.g., `scr_path_prepend_str`. One may insert a formatted string into a path using functions ending with `_strf`, e.g., `scr_path_prepend_strf`.

Slicing paths

A number of functions are available to slice paths into smaller pieces. First, one can chop components from the start and end of a path.

```
scr_path_slice(path, offset, length);
```

This modifies `path` to keep length components starting from the specified offset. The offset can be negative to count from the back. A negative length means that all components are taken starting from the offset to the end of the path.

A shortcut to chop off the last component.

```
scr_path_dirname(path);
```

A shortcut that keeps only the last component.

```
scr_path_basename(path);
```

The following function cuts a path in two at the specified offset. All components starting at offset are returned as a newly allocated path. The original path is modified to contain the beginning components.

```
scr_path* path2 = scr_path_cut(path1, offset);
```

The above functions modify the source path. If one wants to take a piece of a path without modifying the source, you can use the following function. To create a new path which is a substring of a path.

```
scr_path* path2 = scr_path_sub(path, offset, length);
```

The offset and length values have the same meaning as in `scr_path_slice`.

Other path manipulation

A common need when dealing with paths is to simplify them to some reduced form. The following function eliminates all ".", "..", consecutive '/', and trailing '/' characters.

```
scr_path_reduce(path);
```

As an example, the above function converts a path like "/hello/world/../foo/bar/.././" to "/hello/foo".

Since it is common to start from a string, reduce the path, and convert back to a string, there is a shortcut that allocates a new, reduced path as a string.

```
char* reduced_str = scr_path_strdup_reduce_str(str);
scr_free(&reduced_str);
```

The caller is responsible for freeing the returned string.

Another useful function is to compute one path relative to another.

```
scr_path* path = scr_path_relative(src, dst);
```

This function computes `dst` as a path relative to `src` and returns the result as a newly allocated path object. For example, if `src` is "/hello/world" and `dst` is "/hello/foo", the returned path would be "../foo".

2.3 Hash

2.3.1 Overview

A frequently used data structure is the `scr_hash` object. This data structure contains an unordered list of elements, where each element contains a key (a string) and a value (another hash). Each element in a hash has a unique key. Using the key, one can get, set, and unset elements in a hash. There are functions to iterate through the elements of a hash. There are also functions to pack and unpack a hash into a memory buffer, which enables one to transfer a hash through the network or store the hash to a file.

Throughout the documentation and comments in the source code, a hash is often displayed as a tree structure. The key belonging to a hash element is shown as a parent node, and the elements in the hash belonging to that element are displayed as children of that node. For example, consider the following tree:

```
+- RANK
   +- 0
   |  +- FILES
   |  |  +- 2
   |  +- FILE
   |     +- foo_0.txt
   |     |  +- SIZE
   |     |  |  +- 1024
   |     |  +- COMPLETE
   |     |     +- 1
   |     +- bar_0.txt
   |        +- SIZE
```

(continues on next page)

(continued from previous page)

```
|        |  +- 2048
|        +- COMPLETE
|           +- 1
+- 1
   +- FILES
   |  +- 1
   +- FILE
      +- foo_1.txt
         +- SIZE
         |  +- 3072
         +- COMPLETE
            +- 1
```

The above example represents a hash that contains a single element with key `RANK`. The hash associated with the `RANK` element contains two elements with keys `0` and `1`. The hash associated with the `0` element contains two elements with keys `FILES` and `FILE`. The `FILES` element, in turn, contains a hash with a single element with a key `2`, which finally contains a hash having no elements.

Often when displaying these trees, the guidelines are not shown and only the indentation is used, like so:

```
RANK
  0
    FILES
      2
    FILE
      foo_0.txt
        SIZE
          1024
        COMPLETE
          1
      bar_0.txt
          SIZE
            2048
          COMPLETE
            1
  1
    FILES
      1
    FILE
      foo_1.txt
        SIZE
          3072
        COMPLETE
          1
```

2.3.2 Common functions

This section lists the most common functions used when dealing with hashes. For a full listing, refer to comments in `scr_hash.h`. The implementation can be found in `scr_hash.c`.

Hash basics

First, before using a hash, one must allocate a hash object.

```
scr_hash* hash = scr_hash_new();
```

And one must free the hash when done with it.

```
scr_hash_delete(&hash);
```

Given a hash object, you may insert an element, specifying a key and another hash as a value.

```
scr_hash_set(hash, key, value_hash);
```

If an element already exists for the specified key, this function deletes the value currently associated with the key and assigns the specified hash as the new value. Thus it is not necessary to unset a key before setting it – setting a key simply overwrites the existing value.

You may also perform a lookup by specifying a key and the hash object to be searched.

```
scr_hash* value_hash = scr_hash_get(hash, key);
```

If the hash has a key by that name, it returns a pointer to the hash associated with the key. If the hash does not have an element with the specified key, it returns NULL.

You can unset a key.

```
scr_hash_unset(hash, key);
```

If a hash value is associated with the specified key, it is freed, and then the element is deleted from the hash. It is OK to unset a key even if it does not exist in the hash.

To clear a hash (unsets all elements).

```
scr_hash_unset_all(hash);
```

To determine the number of keys in a hash.

```
int num_elements = scr_hash_size(hash);
```

To simplify coding, most hash functions accept NULL as a valid input hash parameter. It is interpreted as an empty hash. For example,

`scr_hash_delete(NULL);`	does nothing
`scr_hash_set(NULL, key, value_hash);`	does nothing and returns NULL
`scr_hash_get(NULL, key);`	returns NULL
`scr_hash_unset(NULL, key);`	does nothing
`scr_hash_unset_all(NULL);`	does nothing
`scr_hash_size(NULL);`	returns 0

Accessing and iterating over hash elements

At times, one needs to work with individual hash elements. To get a pointer to the element associated with a key (instead of a pointer to the hash belonging to that element).

```
scr_hash_elem* elem = scr_hash_elem_get(hash, key);
```

To get the key associated with an element.

```
char* key = scr_hash_elem_key(elem);
```

To get the hash associated with an element.

```
scr_hash* hash = scr_hash_elem_hash(elem);
```

It's possible to iterate through the elements of a hash. First, you need to get a pointer to the first element.

```
scr_hash_elem* elem = scr_hash_elem_first(hash);
```

This function returns NULL if the hash has no elements. Then, to advance from one element to the next.

```
scr_hash_elem* next_elem = scr_hash_elem_next(elem);
```

This function returns NULL when the current element is the last element. Below is some example code that iterates through the elements of hash and prints the key for each element:

```
scr_hash_elem* elem;
for (elem = scr_hash_elem_first(hash);
     elem != NULL;
     elem = scr_hash_elem_next(elem))
{
  char* key = scr_hash_elem_key(elem);
  printf("%s\n", key);
}
```

Key/value convenience functions

Often, it's useful to store a hash using two keys which act like a key/value pair. For example, a hash may contain an element with key `RANK`, whose hash contains a set of elements with keys corresponding to rank ids, where each rank id `0`, `1`, `2`, etc. has a hash, like so:

```
RANK
  0
    <hash for rank 0>
  1
    <hash for rank 1>
  2
    <hash for rank 2>
```

This case comes up so frequently that there are special key/value (_kv) functions to make this operation easier. For example, to access the hash for rank 0 in the above example, one may call

```
scr_hash* rank_0_hash = scr_hash_get_kv(hash, "RANK", "0");
```

This searches for the `RANK` element in the specified hash. If found, it then searches for the `0` element in the hash of the `RANK` element. If found, it returns the hash associated with the `0` element. If hash is NULL, or if hash has no `RANK` element, or if the `RANK` hash has no `0` element, this function returns NULL.

The following function behaves similarly to `scr_hash_get_kv` – it returns the hash for rank 0 if it exists. It differs in that it creates and inserts hashes and elements as needed such that an empty hash is created for rank 0 if it does not already exist.

```
scr_hash* rank_0_hash = scr_hash_set_kv(hash, "RANK", "0");
```

This function creates a RANK element if it does not exist in the specified hash, and it creates a 0 element in the RANK hash if it does not exist. It returns the hash associated with the 0 element, which will be an empty hash if the 0 element was created by the call. This feature lets one string together multiple calls without requiring lots of conditional code to check whether certain elements already exist. For example, the following code is valid whether or not hash has a RANK element.

```
scr_hash* rank_hash = scr_hash_set_kv(hash,      "RANK", "0");
scr_hash* ckpt_hash = scr_hash_set_kv(rank_hash, "CKPT", "10");
scr_hash* file_hash = scr_hash_set_kv(ckpt_hash, "FILE", "3");
```

Often, as in the case above, the *value* key is an integer. In order to avoid requiring the caller to convert integers to strings, there are functions to handle the value argument as an int type, e.g, the above segment could be written as

```
scr_hash* rank_hash = scr_hash_set_kv_int(hash,      "RANK",  0);
scr_hash* ckpt_hash = scr_hash_set_kv_int(rank_hash, "CKPT", 10);
scr_hash* file_hash = scr_hash_set_kv_int(ckpt_hash, "FILE",  3);
```

It's also possible to unset key/value pairs.

```
scr_hash_unset_kv(hash, "RANK", "0");
```

This call removes the 0 element from the RANK hash if one exists. If this action causes the RANK hash to be empty, it also removes the RANK element from the specified input hash.

In some cases, one wants to associate a single value with a given key. When attempting to change the value in such cases, it is necessary to first unset a key before setting the new value. Simply setting a new value will insert another element under the key. For instance, consider that one starts with the following hash

```
TIMESTEP
  20
```

If the goal is to modify this hash such that it changes to

```
TIMESTEP
  21
```

then one should do the following

```
scr_hash_unset(hash, "TIMESTEP");
scr_hash_set_kv_int(hash, "TIMESTEP", 21);
```

Simply executing the set operation without first executing the unset operation results in the following

```
TIMESTEP
  20
  21
```

Because it is common to have fields in a hash that should only hold one value, there are several utility functions to set and get such fields defined in scr_hash_util.h and implemented in scr_hash_util.c. For instance, here are a few functions to set single-value fields:

```
int scr_hash_util_set_bytecount(scr_hash* hash, const char* key, unsigned long count);
int scr_hash_util_set_crc32(scr_hash* hash, const char* key, uLong crc);
int scr_hash_util_set_int64(scr_hash* hash, const char* key, int64_t value);
```

These utility routines unset any existing value before setting the new value. They also convert the input value into an appropriate string representation. Similarly, there are corresponding get routines, such as:

```
int scr_hash_util_get_bytecount(const scr_hash* hash, const char* key, unsigned long*␣
↪count);
int scr_hash_util_get_crc32(const scr_hash* hash, const char* key, uLong* crc);
int scr_hash_util_get_int64(const scr_hash* hash, const char* key, int64_T* value);
```

If a value is set for the specified key, and if the value can be interpreted as the appropriate type for the output parameter, the get routine returns `SCR_SUCCESS` and copies the value to the output parameter. Otherwise, the routine does not return `SCR_SUCCESS` and does not modify the output parameter.

For example, to set and get the timestep value from the example above, one could do the following:

```
scr_hash_util_set_int64(hash, "TIMESTEP", 21);

int64_t current_timestep = -1;
if (scr_hash_util_get_int64(hash, "TIMESTEP", &current_timestep) == SCR_SUCCESS) {
  /* TIMESTEP was set, and it's value is now in current_timestep */
} else {
  /* TIMESTEP was not set, and current_timestep is still -1 */
}
```

The difference between these utility functions and the key/value (`_kv`) functions is that the key/value functions are used to set and get a hash that is referenced by a key/value pair whereas the utility functions set and get a scalar value that has no associated hash.

Specifying multiple keys with format functions

One can set many keys in a single call using a printf-like statement. This call converts variables like floats, doubles, and longs into strings. It enables one to set multiple levels of keys in a single call, and it enables one to specify the hash value to associate with the last element.

```
scr_hash_setf(hash, value_hash, "format", variables ...);
```

For example, if one had a hash like the following

```
RANK
  0
    CKPT
      10
        <current_hash>
```

One could overwrite the hash associated with the `10` element in a single call like so.

```
scr_hash_setf(hash, new_hash, "%s %d %s %d", "RANK", 0, "CKPT", 10);
```

Different keys are separated by single spaces in the format string. Only a subset of the printf format strings are supported.

There is also a corresponding getf version.

```
scr_hash* hash = scr_hash_getf(hash, "%s %d %s %d", "RANK", 0, "CKPT", 10);
```

Sorting hash keys

Generally, the keys in a hash are not ordered. However, one may order the keys with the following sort routines.

```
scr_hash_sort(hash, direction);
scr_hash_sort_int(hash, direction);
```

The first routine sorts keys by string, and the second sorts keys as integer values. The direction variable may be either SCR_HASH_SORT_ASCENDING or SCR_HASH_SORT_DESCENDING. The keys remain in sorted order until new keys are added. The order is not kept between packing and unpacking hashes.

Listing hash keys

One may get a sorted list of all keys in a hash.

```
int num_keys;
int* keys;
scr_hash_list_int(hash, &num_keys, &keys);
...
if (keys != NULL)
  free(keys);
```

This routine returns the number of keys in the hash, and if there is one or more keys, it allocates memory and returns the sorted list of keys. The caller is responsible for freeing this memory. Currently, one may only get a list of keys that can be represented as integers. There is no such list routine for arbitrary key strings.

Packing and unpacking hashes

A hash can be serialized into a memory buffer for network transfer or storage in a file. To determine the size of a buffer needed to pack a hash.

```
int num_bytes = scr_hash_pack_size(hash);
```

To pack a hash into a buffer.

```
scr_hash_pack(buf, hash);
```

To unpack a hash from a buffer into a given hash object.

```
scr_hash* hash = scr_hash_new();
scr_hash_unpack(buf, hash);
```

One must pass an empty hash to the unpack function.

Hash files

Hashes may be serialized to a file and restored from a file. To write a hash to a file.

```
scr_hash_file_write(filename, hash);
```

This call creates the file if it does not exist, and it overwrites any existing file.

To read a hash from a file (merges hash from file into given hash object).

```
scr_hash_file_read(filename, hash);
```

Many hash files are written and read by more than one process. In this case, locks can be used to ensure that only one process has access to the file at a time. A process blocks while waiting on the lock. The following call blocks the calling process until it obtains a lock on the file. Then it opens, reads, closes, and unlocks the file. This results in an atomic read among processes using the file lock.

```
scr_hash_read_with_lock(filename, hash)
```

To update a locked file, it is often necessary to execute a read-modify-write operation. For this there are two functions. One function locks, opens, and reads a file.

```
scr_hash_lock_open_read(filename, &fd, hash)
```

The opened file descriptor is returned, and the contents of the file are read (merged) in to the specified hash object. The second function writes, closes, and unlocks the file.

```
scr_hash_write_close_unlock(filename, &fd, hash)
```

One must pass the filename, the opened file descriptor, and the hash to be written to the file.

Sending and receiving hashes

There are several functions to exchange hashes between MPI processes. While most hash functions are implemented in `scr_hash.c`, the functions dependent on MPI are implemented in `scr_hash_mpi.c`. This is done so that serial programs can use hashes without having to link to MPI.

To send a hash to another MPI process.

```
scr_hash_send(hash, rank, comm)
```

This call executes a blocking send to transfer a copy of the specified hash to the specified destination rank in the given MPI communicator. Similarly, to receive a copy of a hash.

```
scr_hash_recv(hash, rank, comm)
```

This call blocks until it receives a hash from the specified rank, and then it unpacks the received hash into `hash` and returns.

There is also a function to simultaneously send and receive hashes, which is useful to avoid worrying about ordering issues in cases where a process must both send and receive a hash.

```
scr_hash_sendrecv(hash_send, rank_send, hash_recv, rank_recv, comm)
```

The caller provides the hash to be sent and the rank it should be sent to, along with a hash to unpack the received into and the rank it should receive from, as well as, the communicator to be used.

A process may broadcast a hash to all ranks in a communicator.

```
scr_hash_bcast(hash, root, comm)
```

As with MPI, all processes must specify the same root and communicator. The root process specifies the hash to be broadcast, and each non-root process provides a hash into which the broadcasted hash is unpacked.

Finally, there is a call used to issue a (sparse) global exchange of hashes, which is similar to an `MPI_Alltoallv` call.

```
scr_hash_exchange(hash_send, hash_recv, comm)
```

This is a collective call which enables any process in `comm` to send a hash to any other process in `comm` (including itself). Furthermore, the destination processes do not need to know from which processes they will receive data in advance. As input, a process should provide an empty hash for `hash_recv`, and it must structure `hash_send` in the following manner.

```
 <rank_X>
   <hash_to_send_to_rank_X>
<rank_Y>
  <hash_to_send_to_rank_Y>
```

Upon return from the function, `hash_recv` will be filled in according to the following format.

```
<rank_A>
  <hash_received_from_rank_A>
<rank_B>
  <hash_received_from_rank_B>
```

For example, if `hash_send` was the following on rank 0 before the call:

```
hash_send on rank 0:
1
  FILES
    1
  FILE
    foo.txt
2
  FILES
    1
  FILE
    bar.txt
```

Then after returning from the call, `hash_recv` would contain the following on ranks 1 and 2:

```
hash_recv on rank 1:
0
  FILES
    1
  FILE
    foo.txt
<... data from other ranks ...>

hash_recv on rank 2:
0
  FILES
    1
  FILE
    bar.txt
<... data from other ranks ...>
```

The algorithm used to implement this function assumes the communication is sparse, meaning that each process only sends to or receives from a small number of other processes. It may also be used for gather or scatter operations.

2.3.3 Debugging

Newer versions of TotalView enable one to dive on hash variables and inspect them in a variable window using a tree view. For example, when diving on a hash object corresponding to the example hash in the overview section, one would see an expanded tree in the variable view window like so:

```
+- RANK
   +- 0
   |  +- FILES = 2
   |  +- FILE
   |     +- foo_0.txt
   |     |  +- SIZE = 1024
   |     |  +- COMPLETE = 1
   |     +- bar_0.txt
   |        +- SIZE = 2048
   |        +- COMPLETE = 1
   +- 1
      +- FILES = 1
      +- FILE
         +- foo_1.txt
            +- SIZE = 3072
            +- COMPLETE = 1
```

When a hash of an element contains a single element whose own hash is empty, this display condenses the line to display that entry as a key = value pair.

If TotalView is not available, one may resort to printing a hash to `stdout` using the following function. The number of spaces to indent each level is specified in the second parameter.

```
scr_hash_print(hash, indent);
```

To view the contents of a hash file, there is a utility called `scr_print_hash_file` which reads a file and prints the contents to the screen.

```
scr_print_hash_file  myhashfile.scr
```

2.3.4 Binary format

This section documents the binary format used when serializing a hash.

Packed hash

A hash can be serialized into a memory buffer for network transfer or storage in a file. When serialized, all integers are stored in network byte order (big-endian format). Such a "packed" hash consists of the following format:

Format of a PACKED HASH:

Field Name	Datatype	Description
Count	`uint32_t`	Number of elements in hash
		A count of 0 means the hash is empty.
Elements	PACKED	Sequence of packed elements of length Count.
	ELEMENT	

Format of a PACKED ELEMENT:

Field Name	Datatype	Description
Key	NULL-terminated ASCII string	Key associated with element
Hash	PACKED	Hash associated with element
	HASH	

File format

A hash can be serialized and stored as a binary file. This section documents the file format for an `scr_hash` object. All integers are stored in network byte order (big-endian format). A hash file consists of the following sequence of bytes:

Field Name	Datatype	Description
Magic Number	`uint32_t`	Unique integer to help distinguish an SCR file from other types of files
		0x951fc3f5 (host byte order)
File Type	`uint16_t`	Integer field describing what type of SCR file this file is
		1 → file is an `scr_hash` file
File Version	`uint16_t`	Integer field that together with File Type defines the file format
		1 → `scr_hash` file is stored in version 1 format
File Size	`uint64_t`	Size of this file in bytes, from first byte of the header to the last byte in the file.
Flags	`uint32_t`	Bit flags for file.
Data	PACKED	Packed hash data (see Section *1.4.1*).
	HASH	
CRC32*	`uint32_t`	CRC32 of file, accounts for first byte of header to last byte of Data.
		*Only exists if `SCR_FILE_FLAGS_CRC32` bit is set in Flags.

2.4 Filemap

2.4.1 Overview

The `scr_filemap` data structure maintains the mapping between files, process ranks, and datasets (checkpoints). In a given dataset, each process may write zero or more files. SCR uses the filemap to record which rank writes which file in which dataset. The complete mapping is distributed among processes. Each process only knows a partial mapping. A process typically knows the mapping for its own files as well as the mapping for a few other processes that it provides redundancy data for.

The filemap tracks all files currently in cache, and it is recorded in a file in the control directory. Each process manages its own *filemap file*, so that a process may modify its filemap file without coordinating with other processes. In addition, the master process on each node maintains a *master filemap file*, which is written to a well-known name and records the file names of all of the per-process filemap files that are on the same node.

Before any file is written to the cache, a process adds an entry for the file to its filemap and then updates its filemap file on disk. Similarly, after a file is deleted from cache, the corresponding entry is removed from the filemap and the filemap file is updated on disk. Following this protocol, a file will not exist in cache unless it has a corresponding entry in the filemap file. On the other hand, an entry in the filemap file does not ensure that a corresponding file exists in cache – it only implies that the corresponding file *may* exist.

When an SCR job starts, the SCR library attempts to read the filemap files from the control directory to determine what datasets are stored in cache. The library uses this information to determine which datasets and which ranks it has data for. The library also uses this data to know which files to remove when deleting a dataset from cache, and it uses this data to know which files to copy when flushing a dataset to the parallel file system.

SCR internally numbers each checkpoint with two unique numbers: a *dataset id* and a *checkpoint id*. Functions that return dataset or checkpoint ids return -1 if there is no valid dataset or checkpoint contained in the filemap that matches a particular query.

The `scr_filemap` makes heavy use of the `scr_hash` data structure (Section *Hash*). The `scr_hash` is utilized in the `scr_filemap` API and its implementation.

2.4.2 Example filemap hash

Internally, filemaps are implemented as `scr_hash` objects. Here is an example hash for a filemap object containing information for ranks 20 and 28 and dataset ids 10 and 11.

```
DSET
  10
    RANK
      20
  11
    RANK
      20
      28
RANK
  20
    DSET
      10
        FILES
          2
        FILE
          /<path_to_foo_20.ckpt.10>/foo_20.ckpt.10
            <meta_data_for_foo_20.ckpt.10>
          /<path_to_foo_20.ckpt.10.xor>/foo_20.ckpt.10.xor
            <meta_data_for_foo_20.ckpt.10.xor>
        REDDESC
          <redundancy descriptor hash>
        DATADESC
          <dataset_descriptor_for_dataset_10>
      11
        FILES
          1
        FILE
          /<path_to_foo_20.ckpt.11>/foo_20.ckpt.11
            <meta_data_for_foo_20.ckpt.11>
        REDDESC
          <redundancy descriptor hash>
        DATADESC
          <dataset_descriptor_for_dataset_11>
  28
    DSET
      11
        FILES
          1
        FILE
          /<path_to_foo_28.ckpt.11>/foo_28.ckpt.11
            <meta_data_for_foo_28.ckpt.11>
        PARTNER
          atlas56
        REDDESC
          <redundancy descriptor hash>
```

The main data is kept under the RANK element at the top level. Rank ids are listed in the hash of the RANK element. Within each rank id, dataset ids are listed in the hash of a DSET element. Finally, each dataset id contains elements for the expected number of files (under FILES), the file names (under FILE), the redundancy descriptor hash (under REDDESC, see Section *Redundancy descriptors*) that describes the redundancy scheme applied to the dataset files, and a hash providing details about the dataset (under DATADESC). In addition, there may be arbitrary tags such as the PARTNER element.

Note that the full path to each file is recorded, and a meta data hash is also recorded for each file, which contains attributes specific to each file.

There is also a DSET element at the top level, which lists rank ids by dataset id. This information is used as an index to provide fast lookups for certain queries, such as to list all dataset ids in the filemap, to determine whether there are any entries for a given dataset id, and to lookup all ranks for a given dataset. This index is kept in sync with the information contained under the RANK element.

2.4.3 Common functions

This section describes some of the most common filemap functions. For a detailed list of all functions, see scr_filemap.h. The implementation can be found in scr_filemap.c.

Allocating, freeing, merging, and clearing filemaps

Create a new filemap object.

```
scr_filemap* map = scr_filemap_new();
```

Free a filemap object.

```
scr_filemap_delete(&map);
```

Copy entries from filemap_2 into filemap_1.

```
scr_filemap_merge(filemap_1, filemap_2);
```

Delete all entries from a filemap.

```
scr_filemap_clear(map);
```

Adding and removing data

Add an entry for a file for a given rank id and dataset id.

```
scr_filemap_add_file(map, dset, rank, filename);
```

Remove an entry for a file for a given rank id and dataset id.

```
scr_filemap_remove_file(map, dset, rank, filename);
```

Remove all info corresponding to a given dataset id.

```
scr_filemap_remove_dataset(map, dset);
```

Remove all info corresponding to a given rank.

```
scr_filemap_remove_rank(map, rank);
```

Remove all info corresponding to a given rank for a given dataset number.

```
scr_filemap_remove_rank_by_dataset(map, dset, rank);
```

Extract all info for a rank from specified map and return as a newly created filemap. This also deletes the corresponding info from the source filemap.

```
scr_filemap* rank_filemap = scr_filemap_extract_rank(map, rank);
```

Query functions

Get the number of datasets in a filemap.

```
int num_dsets = scr_filemap_num_datasets(map);
```

Get the most recent dataset (highest dataset id).

```
int dset = scr_filemap_latest_dataset(map);
```

Get the oldest dataset (lowest dataset id).

```
int dset = scr_filemap_oldest_dataset(map);
```

Get the number of ranks in a filemap.

```
int num_ranks = scr_filemap_num_ranks(map);
```

Get the number of ranks in a filemap for a given dataset.

```
int num_ranks = scr_filemap_num_ranks_by_dataset(map, dset);
```

Determine whether the map contains any data for a specified rank. Returns 1 if true, 0 if false.

```
scr_filemap_have_rank(map, rank);
```

Determine whether the map contains any data for a specified rank for a given dataset id. Returns 1 if true, 0 if false.

```
scr_filemap_have_rank_by_dataset(map, dset, rank);
```

For a given rank in a given dataset, there are two file counts that are of interest. First, there is the "expected" number of files. This refers to the number of files that a process wrote during the dataset. Second, there is the "actual" number of files the filemap contains data for. This distinction enables SCR to determine whether a filemap contains data for all files a process wrote during a given dataset.

For a given rank id and dataset id, get the number of files the filemap contains info for.

```
int num_files = scr_filemap_num_files(map, dset, rank);
```

Set the number of expected files for a rank during a given dataset.

```
scr_filemap_set_expected_files(map, dset, rank, num_expected_files);
```

Get the number of expected files for a rank during a dataset.

```
int num_expected_files = scr_filemap_get_expected_files(map, dset, rank);
```

Unset the number of expected files for a given rank and dataset.

```
scr_filemap_unset_expected_files(map, dset, rank);
```

List functions

There a number of functions to return a list of entries in a filemap. The function will allocate and return the list in an output parameter. The caller is responsible for freeing the list if it is not NULL.

Get a list of all dataset ids (ordered oldest to most recent).

```
int ndsets;
int* dsets;
scr_filemap_list_datasets(map, &ndsets, &dsets);
...
if (dsets != NULL)
  free(dsets);
```

Get a list of all rank ids (ordered smallest to largest).

```
int nranks;
int* ranks;
scr_filemap_list_ranks(map, &nranks, &ranks);
...
if (ranks != NULL)
  free(ranks);
```

Get a list of all rank ids for a given dataset (ordered smallest to largest).

```
int nranks;
int* ranks;
scr_filemap_list_ranks_by_dataset(map, dset, &nranks, &ranks);
...
if (ranks != NULL)
  free(ranks);
```

To get a count of files and a list of file names contained in the filemap for a given rank id in a given dataset. The list is in arbitrary order.

```
int nfiles;
char** files;
scr_filemap_list_files(map, ckpt, rank, &nfiles, &files);
...
if (files != NULL)
  free(files);
```

In this last case, the pointers returned in files point to the strings in the elements within the filemap. Thus, if any elements are deleted or changed, these pointers will be invalid and should not be dereferenced. In this case, a new list of files should be obtained.

When using the above functions, the caller is responsible for freeing memory allocated to store the list if it is not NULL.

Iterator functions

One may obtain a pointer to an `scr_hash_elem` object which can be used with the `scr_hash` functions to iterate through the values of a filemap. The iteration order is arbitrary.

To iterate through the dataset ids contained in a filemap.

```
scr_hash_elem* elem = scr_filemap_first_dataset(map);
```

To iterate through the ranks contained in a filemap for a given dataset id.

```
scr_hash_elem* elem = scr_filemap_first_rank_by_dataset(map, dset);
```

To iterate through the files contained in a filemap for a given rank id and dataset id.

```
scr_hash_elem* elem = scr_filemap_first_file(map, dset, rank);
```

Dataset descriptors

The filemap also records dataset descriptors for a given rank and dataset id. These descriptors associate attributes with a dataset (see Section *Datasets*).

To record a dataset descriptor for a given rank and dataset id.

```
scr_filemap_set_dataset(map, dset, rank, desc);
```

To get a dataset descriptor for a given rank and dataset id.

```
scr_dataset* desc = scr_dataset_new();
scr_filemap_get_dataset(map, dset, rank, desc);
```

To unset a dataset descriptor for a given rank and dataset id.

```
scr_filemap_unset_dataset(map, dset, rank);
```

File meta data

In addition to recording the filenames for a given rank and dataset, the filemap also records meta data for each file, including the expected size of the file and CRC32 checksums (see Section *Meta data*).

To record meta data for a file.

```
scr_filemap_set_meta(map, dset, rank, file, meta);
```

To get a meta data for a file.

```
scr_meta* meta = scr_meta_new();
scr_filemap_get_meta(map, dset, rank, file, meta);
```

To unset meta data for a file.

```
scr_filemap_unset_meta(map, dset, rank, file);
```

One must specify the same filename that was used during the call to `scr_filemap_add_file()`.

Redundancy descriptors

A redundancy descriptor is a data structure that describes the location and redundancy scheme that is applied to a set of dataset files in cache (Section *Redundancy descriptors*). In addition to knowing what dataset files are in cache, it's also useful to know what redundancy scheme is applied to that data. To do this, a redundancy descriptor can be associated with a given dataset and rank in the filemap.

Given a redundancy descriptor hash, associate it with a given dataset id and rank id.

```
scr_filemap_set_desc(map, dset, rank, desc);
```

Given a dataset id and rank id, get the corresponding descriptor.

```
scr_filemap_get_desc(map, dset, rank, desc);
```

Unset a redundancy descriptor.

```
scr_filemap_unset_desc(map, ckpt, rank)
```

Tags

One may also associate arbitrary key/value string pairs for a given dataset id and rank. It is the caller's responsibility to ensure the tag name does not collide with another key in the filemap.

To assign a tag (string) and value (another string) to a dataset.

```
scr_filemap_set_tag(map, dset, rank, tag, value);
```

To retrieve the value associated with a tag.

```
char* value = scr_filemap_get_tag(map, dset, rank, tag);
```

To unset a tag value.

```
scr_filemap_unset_tag(map, dset, rank, tag);
```

Accessing a filemap file

A filemap can be serialized to a file. The following functions write a filemap to a file and read a filemap from a file.

Write the specified filemap to a file.

```
scr_filemap_write(filename, map);
```

Read contents from a filemap file and merge into specified filemap object.

```
scr_filemap_read(filename, map);
```

2.5 Datasets

2.5.1 Overview

The scr_dataset data structure associates various attributes with each dataset written by the application. It tracks information such as the dataset id, the creation time, the total number of bytes.

The scr_hash is utilized in the scr_dataset API and its implementation. Essentially, scr_dataset objects are specialized scr_hash objects that have certain well-defined keys (*fields*) and associated functions to access those fields.

2.5.2 Example dataset hash

Internally, dataset objects are implemented as scr_hash objects. Here is an example hash for a dataset object.

```
ID
  23
USER
  user1
JOBNAME
  simulation123
NAME
  dataset.23
SIZE
  524294000
FILES
  1024
CREATED
  1312850690668536
CKPT
  6
COMPLETE
  1
```

The ID field records the dataset id of the dataset as assigned by the scr_dataset_id variable at the time the dataset is created. The USER field records the username associated with the job within which the dataset was created, and the value of $SCR_JOB_NAME, if set, is recorded in the JOBNAME field. The NAME field records the name of the dataset. This is currently defined to be "dataset.<id>" where <id> is the dataset id. The total number of bytes in the dataset is recorded in the SIZE field, and the total number of files is recorded in FILES. The CREATED field records the time at which the dataset was created, in terms of microseconds since the Linux epoch. If the dataset is a checkpoint, the checkpoint id is recorded in the CKPT field. The COMPLETE field records whether the dataset is valid. It is set to 1 if the dataset is thought to be valid, and 0 otherwise.

These are the most common fields used in dataset objects. Not all fields are required, and additional fields may be used that are not shown here.

2.5.3 Common functions

This section describes some of the most common dataset functions. For a detailed list of all functions, see scr_dataset.h. The implementation can be found in scr_dataset.c.

Allocating and freeing dataset objects

Create a new dataset object.

```
scr_dataset* dataset = scr_dataset_new()
```

Free a dataset object.

```
scr_dataset_delete(&dataset);
```

Setting, getting, and checking field values

There are functions to set each field individually.

```
int scr_dataset_set_id(scr_dataset* dataset, int id);
int scr_dataset_set_user(scr_dataset* dataset, const char* user);
int scr_dataset_set_jobname(scr_dataset* dataset, const char* name);
int scr_dataset_set_name(scr_dataset* dataset, const char* name);
int scr_dataset_set_size(scr_dataset* dataset, unsigned long size);
int scr_dataset_set_files(scr_dataset* dataset, int files);
int scr_dataset_set_created(scr_dataset* dataset, int64_t created);
int scr_dataset_set_jobid(scr_dataset* dataset, const char* jobid);
int scr_dataset_set_cluster(scr_dataset* dataset, const char* name);
int scr_dataset_set_ckpt(scr_dataset* dataset, int id);
int scr_dataset_set_complete(scr_dataset* dataset, int complete);
```

If a field was already set to a value before making this call, the new value overwrites any existing value.

And of course there are corresponding functions to get values.

```
int scr_dataset_get_id(const scr_dataset* dataset, int* id);
int scr_dataset_get_user(const scr_dataset* dataset, char** name);
int scr_dataset_get_jobname(const scr_dataset* dataset, char** name);
int scr_dataset_get_name(const scr_dataset* dataset, char** name);
int scr_dataset_get_size(const scr_dataset* dataset, unsigned long* size);
int scr_dataset_get_files(const scr_dataset* dataset, int* files);
int scr_dataset_get_created(const scr_dataset* dataset, int64_t* created);
int scr_dataset_get_jobid(const scr_dataset* dataset, char** jobid);
int scr_dataset_get_cluster(const scr_dataset* dataset, char** name);
int scr_dataset_get_ckpt(const scr_dataset* dataset, int* id);
int scr_dataset_get_complete(const scr_dataset* dataset, int* complete);
```

If the corresponding field is set, the get functions copy the value into the output parameter and return `SCR_SUCCESS`. If `SCR_SUCCESS` is not returned, the output parameter is not changed.

2.6 Meta data

2.6.1 Overview

The `scr_meta` data structure associates various properties with files written by the application and with redundancy data files written by SCR. It tracks information such as the type of file (application vs. SCR redundancy data), whether the application marked the file as valid or invalid, the expected file size, its CRC32 checksum value if computed, and the original string the application used to register the file. Because the meta data is stored within a filemap (Section *Filemap*), there is no need to store the dataset id or rank id which owns the file.

The scr_meta data structure makes heavy use of the scr_hash data structure (Section *Hash*). The scr_hash is utilized in the scr_meta API and its implementation. Essentially, scr_meta objects are specialized scr_hash objects, which have certain well-defined keys (*fields*) and associated functions to access those fields.

2.6.2 Example meta data hash

Internally, meta data objects are implemented as scr_hash objects. Here is an example hash for a meta data object containing information for a file named "rank_0.ckpt".

```
FILE
  rank_0.ckpt
TYPE
  FULL
COMPLETE
  1
SIZE
  524294
CRC
  0x1b39e4e4
CKPT
  6
RANKS
  4
ORIG
  ckpt.6/rank_0.ckpt
ORIGPATH
  /p/lscratchb/user3/simulation123/ckpt.6
ORIGNAME
  rank_0.ckpt
```

The FILE field records the file name this meta data associates with. In this example, the file name is recorded using a relative path. The TYPE field indicates whether the file is written by the application (FULL), whether it's a PARTNER copy of a file (PARTNER), or whether it's a redundancy file for an XOR set (XOR). The COMPLETE field records whether the file is valid. It is set to 1 if the file is thought to be valid, and 0 otherwise. The SIZE field records the size of the file in bytes. The CRC field records the CRC32 checksum value over the contents of the file. The CKPT field records the checkpoint id in which the file was written. The RANKS field record the number of ranks active in the run when the file was created. The ORIG field records the original string specified by the caller when the file was registered in the call to SCR_Route_File(). The ORIGPATH field records the absolute path to the original file at the time the file was registered, and the ORIGNAME field records just the name of the file when registered.

In this case, "rank_0.ckpt" was created during checkpoint id 6, and it was written in a run with 4 MPI tasks. It was written by the application, and it is marked as being complete. It consists of 524,294 bytes and its CRC32 value is 0x1b39e4e4. The caller referred to this file as "ckpt.6/rank_0.ckpt" when registering this file in SCR_Route_file(). Based on the current working directory at the time when SCR_Route_file was called, the absolute path to the file would have been "/p/lscratchb/user3/simulation123/ckpt.6" and its name would have been "rank_0.ckpt".

These are the most common fields used in meta data objects. Not all fields are required, and additional fields may be used that are not shown here.

2.6.3 Common functions

This section describes some of the most common meta data functions. For a detailed list of all functions, see scr_meta.h. The implementation can be found in scr_meta.c.

Allocating, freeing, and copying meta data objects

Create a new meta data object.

```
scr_meta* meta = scr_meta_new()
```

Free a meta data object.

```
scr_meta_delete(&meta)
```

Make an exact copy of `meta_2` in `meta_1`.

```
scr_meta_copy(meta_1, meta_2)
```

Setting, getting, and checking field values

There are functions to set each field individually.

```
scr_meta_set_complete(meta, complete)
scr_meta_set_ranks(meta, ranks)
scr_meta_set_checkpoint(meta, ckpt)
scr_meta_set_filesize(meta, filesize)
scr_meta_set_filetype(meta, filetype)
scr_meta_set_filename(meta, filename)
scr_meta_set_crc32(meta, crc32)
scr_meta_set_orig(meta, string)
scr_meta_set_origpath(meta, path)
scr_meta_set_origname(meta, name)
```

If a field was already set to a value before making this call, the new value overwrites any existing value.

And of course there are corresponding functions to get values.

```
scr_meta_get_complete(meta, complete)
scr_meta_get_ranks(meta, ranks)
scr_meta_get_checkpoint(meta, ckpt)
scr_meta_get_filesize(meta, filesize)
scr_meta_get_filetype(meta, filetype)
scr_meta_get_filename(meta, filename)
scr_meta_get_crc32(meta, crc32)
scr_meta_get_orig(meta, string)
scr_meta_get_origpath(meta, path)
scr_meta_get_origname(meta, name)
```

If the corresponding field is set, the get functions copy the value into the output parameter and return `SCR_SUCCESS`. If `SCR_SUCCESS` is not returned, the output parameter is not changed.

Many times one simply wants to verify that a field is set to a particular value. The following functions return `SCR_SUCCESS` if a field is set and if that field matches the specified value.

```
scr_meta_check_ranks(meta, ranks)
scr_meta_check_checkpoint(meta, ckpt)
scr_meta_check_filesize(meta, filesize)
scr_meta_check_filetype(meta, filetype)
scr_meta_check_filename(meta, filename)
```

Similar to the above functions, the following function returns `SCR_SUCCESS` if the complete field is set and if its value is set to 1.

```
scr_meta_check_complete(meta)
```

2.7 Group descriptors

2.7.1 Overview

A group descriptor is a data structure that describes a group of processes. Each group is given a name, which is used as a key to refer to the group. For each group name, a process belongs to at most one group, which is a subset of all processes in the job.

There are two pre-defined groups: `WORLD` which contains all processes in `MPI_COMM_WORLD` and `NODE` which contains all processes on the same node. SCR determines which processes are on the same node by splitting processes into groups that have the same value for `scr_my_hostname`, which is set by calling `scr_env_hostname()`.

Additional groups may be defined via entries in the system or user configuration files. It is necessary to define additional groups when failure modes or storage devices span multiple compute nodes. For example if network switch failures are common, then one could define a group to specify which nodes share a network switch to enable SCR to protect against such failures.

The group descriptor is a C struct. During the run, the SCR library maintains an array of group descriptor structures in a global variable named `scr_groupdescs`. It records the number of descriptors in this list in a variable named `scr_ngroupdescs`. It builds this list during `SCR_Init()` by calling `scr_groupdescs_create()` which constructs the list from a third variable called `scr_groupdescs_hash`. This hash variable is initialized from entries in the configuration files while processing SCR parameters. The group structures are freed in `SCR_Finalize()` by calling `scr_groupdescs_free()`.

2.7.2 Group descriptor struct

Here is the definition for the C struct.

```
typedef struct {
  int      enabled;      /* flag indicating whether this descriptor is active */
  int      index;        /* each descriptor is indexed starting from 0 */
  char*    name;         /* name of group */
  MPI_Comm comm;         /* communicator of processes in same group */
  int      rank;         /* local rank of process in communicator */
  int      ranks;        /* number of ranks in communicator */
} scr_groupdesc;
```

The `enabled` field is set to 0 (false) or 1 (true) to indicate whether this particular group descriptor may be used. Even though a group descriptor may be defined, it may be disabled. The `index` field records the index within the `scr_groupdescs` array. The `name` field is a copy of the group name. The `comm` field is a handle to the MPI communicator that defines the group the process is a member of. The `rank` and `ranks` fields cache the rank of the process in this communicator and the number of processes in this communicator, respectively.

2.7.3 Example group descriptor configuration file entries

Here are some examples of configuration file entries to define new groups.

```
GROUPS=zin1  POWER=psu1  SWITCH=0
GROUPS=zin2  POWER=psu1  SWITCH=1
GROUPS=zin3  POWER=psu2  SWITCH=0
GROUPS=zin4  POWER=psu2  SWITCH=1
```

Group descriptor entries are identified by a leading `GROUPS` key. Each line corresponds to a single compute node, where the hostname is the value of the `GROUPS` key. There must be one line for every compute node in the allocation. It is recommended to specify groups in the system configuration file.

The remaining values on the line specify a set of group name / value pairs. The group name is the string to be referenced by store and checkpoint descriptors. The value can be an arbitrary character string. The only requirement is that for a given group name, nodes that form a group must specify identical values.

In the above example, there are four compute nodes: zin1, zin2, zin3, and zin4. There are two groups defined: `POWER` and `SWITCH`. Nodes zin1 and zin2 belong to the same `POWER` group, as do nodes zin3 and zin4. For the `SWITCH` group, nodes zin1 and zin3 belong to the same group, as do nodes zin2 and zin4.

2.7.4 Common functions

This section describes some of the most common group descriptor functions. These functions are defined in `scr_groupdesc.h` and implemented in `scr_groupdesc.c`.

Creating and freeing the group descriptors array

To initialize the `scr_groupdescs` and `scr_ngroupdescs` variables from the `scr_groupdescs_hash` variable:

```
scr_groupdescs_create();
```

Free group descriptors array.

```
scr_groupdescs_free();
```

Lookup group descriptor by name

To lookup a group descriptor by name.

```
scr_groupdesc* group = scr_groupdescs_from_name(name);
```

This returns NULL if the specified group name is not defined. There is also a function to return the index of a group within `scr_groupdescs`.

```
int index = scr_groupdescs_index_from_name(name);
```

This returns an index value in the range $[0, \texttt{scr\_ngroupdescs})$ if the specified group name is defined and it returns -1 otherwise.

2.8 Store descriptors

2.8.1 Overview

A store descriptor is a data structure that describes a class of storage. Each store is given a name, which is used as a key to refer to the storage device.

All storage devices available to SCR must be specified via entries in the system or user configuration files. These entries specify which nodes can access the device, the capacity of the device, and other capabilities such as whether it supports the creation of directories.

The store descriptor is a C struct. During the run, the SCR library maintains an array of store descriptor structures in a global variable named `scr_storedescs`. It records the number of descriptors in this list in a variable named `scr_nstoredescs`. It builds this list during `SCR_Init()` by calling `scr_storedescs_create()` which constructs the list from a third variable called `scr_storedescs_hash`. This hash variable is initialized from entries in the configuration files while processing SCR parameters. The store structures are freed in `SCR_Finalize()` by calling `scr_storedescs_free()`.

2.8.2 Store descriptor struct

Here is the definition for the C struct.

```
typedef struct {
  int      enabled;   /* flag indicating whether this descriptor is active */
  int      index;     /* each descriptor is indexed starting from 0 */
  char*    name;      /* name of store */
  int      max_count; /* maximum number of datasets to be stored in device */
  int      can_mkdir; /* flag indicating whether mkdir/rmdir work */
  char*    type;      /* AXL xfer type string (bbapi, sync, pthread, etc..) */
  char*    view;      /* indicates whether store is node-local or global */
  MPI_Comm comm;      /* communicator of processes that can access storage */
  int      rank;      /* local rank of process in communicator */
  int      ranks;     /* number of ranks in communicator */
} scr_storedesc;
```

The `enabled` field is set to 0 (false) or 1 (true) to indicate whether this particular store descriptor may be used. Even though a store descriptor may be defined, it may be disabled. The `index` field records the index within the `scr_storedescs` array. The `name` field is a copy of the store name. The `comm` field is a handle to the MPI communicator that defines the group the processes that share access to the storage device that the local process uses. The `rank` and `ranks` fields cache the rank of the process in this communicator and the number of processes in this communicator, respectively. `type` is the name of the AXL (https://github.com/ECP-VeloC/AXL) transfer type used internally to copy the files into the storage descriptor. Some transfer types are:

sync: A basic synchronous file copy pthread: Multi-threaded file copy bbapi: Use the IBM Burst Buffer API (if available) dw: Use the Cray DataWarp API (if available)

2.8.3 Example store descriptor configuration file entries

SCR must know about the storage devices available on a system. SCR requires that all processes be able to access the prefix directory, and it assumes that `/tmp` is storage local to each compute node. Additional storage can be described in configuration files with entries like the following.

```
STORE=/tmp           GROUP=NODE   COUNT=1
STORE=/ssd           GROUP=NODE   COUNT=3  TYPE=bbapi
STORE=/dev/persist   GROUP=NODE   COUNT=1  ENABLED=1  MKDIR=0
STORE=/p/lscratcha   GROUP=WORLD  TYPE=pthread
```

Store descriptor entries are identified by a leading `STORE` key. Each line corresponds to a class of storage devices. The value associated with the `STORE` key is the directory prefix of the storage device. This directory prefix also serves as the name of the store descriptor. All compute nodes must be able to access their respective storage device via the specified directory prefix.

The remaining values on the line specify properties of the storage class. The `GROUP` key specifies the group of processes that share a device. Its value must specify a group name. The `COUNT` key specifies the maximum number of checkpoints that can be kept in the associated storage. The user should be careful to set this appropriately depending on the storage capacity and the application checkpoint size. The `COUNT` key is optional, and it defaults to the value of the `SCR_CACHE_SIZE` parameter if not specified. The `ENABLED` key enables (1) or disables (0) the store descriptor. This key is optional, and it defaults to 1 if not specified. The `MKDIR` key specifies whether the device supports the creation of directories (1) or not (0). This key is optional, and it defaults to 1 if not specified. `TYPE` is the AXL transfer type used to copy files into the store descriptor. Values for *TYPE* include:

sync: A basic synchronous file copy pthread: Multi-threaded file copy bbapi: Use the IBM Burst Buffer API (if available) dw: Use the Cray DataWarp API (if available)

`TYPE` is optional, and will default to pthread if not specified.

In the above example, there are four storage devices specified: `/tmp`, `/ssd`, `/dev/persist`, and `/p/lscratcha`. The storage at `/tmp`, `/ssd`, and `/dev/persist` specify the `NODE` group, which means that they are node-local storage. Processes on the same compute node access the same device. The storage at `/p/lscratcha` specifies the `WORLD` group, which means that all processes in the job can access the device. In other words, it is a globally accessible file system.

2.8.4 Common functions

This section describes some of the most common store descriptor functions. These functions are defined in `scr_storedesc.h` and implemented in `scr_storedesc.c`.

Creating and freeing the store descriptors array

To initialize the `scr_storedescs` and `scr_nstoredescs` variables from the `scr_storedescs_hash` variable:

```
scr_storedescs_create();
```

Free store descriptors array.

```
scr_storedescs_free();
```

Lookup store descriptor by name

To lookup a store descriptor by name.

```
int index = scr_storedescs_index_from_name(name);
```

This returns an index value in the range $[0, \texttt{scr\_nstoredescs})$ if the specified store name is defined and it returns -1 otherwise.

Create and delete directories on storage device

These functions are collective over the group of processes that share access to the same storage device. To create a directory on a storage device.

```
int scr_storedesc_dir_create(const scr_storedesc* s, const char* dir);
```

To delete a directory.

```
int scr_storedesc_dir_delete(const scr_storedesc* s, const char* dir);
```

2.9 Redundancy descriptors

2.9.1 Overview

A redundancy descriptor is a data structure that describes how a dataset is cached. It tracks information such as the cache directory that is used, the redundancy scheme that is applied, and the frequency with which this combination should be used. The data structure also records information on the group of processes that make up a redundancy set, such as the number of processes in the set, as well as, a unique integer that identifies the set, called the *group id*.

There is both a C struct and an equivalent specialized hash for storing redundancy descriptors. The hash is primarily used to persist group information across runs, such that the same process group can be reconstructed in a later run (even if the user changes configuration parameters between runs). These hashes are stored in filemap files. The C struct is used within the library to cache additional runtime information such as an MPI communicator for each group and the location of certain MPI ranks.

During the run, the SCR library maintains an array of redundancy descriptor structures in a global variable named `scr_reddescs`. It records the number of descriptors in this list in a variable named `scr_nreddescs`. It builds this list during `SCR_Init()` using a series of redundancy descriptor hashes defined in a third variable named `scr_reddesc_hash`. The hashes in this variable are constructed while processing SCR parameters.

2.9.2 Redundancy descriptor struct

Here is the definition for the C struct.

```
typedef struct {
  int      enabled;        /* flag indicating whether this descriptor is active */
  int      index;          /* each descriptor is indexed starting from 0 */
  int      interval;       /* how often to apply this descriptor, pick largest such
                            * that interval evenly divides checkpoint id */
  int      store_index;    /* index into scr_storedesc for storage descriptor */
  int      group_index;    /* index into scr_groupdesc for failure group */
  char*    base;           /* base cache directory to use */
  char*    directory;      /* full directory base/dataset.id */
  int      copy_type;      /* redundancy scheme to apply */
  void*    copy_state;     /* pointer to extra state depending on copy type */
  MPI_Comm comm;           /* communicator holding procs for this scheme */
  int      groups;         /* number of redundancy sets */
  int      group_id;       /* unique id assigned to this redundancy set */
  int      ranks;          /* number of ranks in this set */
  int      my_rank;        /* caller's rank within its set */
} scr_reddesc;
```

The enabled field is set to 0 (false) or 1 (true) to indicate whether this particular redundancy descriptor may be used. Even though a redundancy descriptor may be defined, it may be disabled. The index field records the index number of this redundancy descriptor. This corresponds to the redundancy descriptor's index in the scr_reddescs array. The interval field describes how often this redundancy descriptor should be selected for different checkpoints. To choose a redundancy descriptor to apply to a given checkpoint, SCR picks the descriptor that has the largest interval value which evenly divides the checkpoint id.

The store_index field tracks the index of the store descriptor within the scr_storedescs array that describes the storage used with this redundancy descriptor. The group_index field tracks the index of the group descriptor within the scr_groupdescs array that describes the group of processes likely to fail at the same time. The redundancy scheme will protect against failures for this group using the specified storage device.

The base field is a character array that records the cache base directory that is used. The directory field is a character array that records the directory in which the dataset subdirectory is created. This path consists of the cache directory followed by the redundancy descriptor index directory, such that one must only append the dataset id to compute the full path of the corresponding dataset directory.

The copy_type field specifies the type of redundancy scheme that is applied. It may be set to one of: SCR_COPY_NULL, SCR_COPY_SINGLE, SCR_COPY_PARTNER, or SCR_COPY_XOR. The copy_state field is a void* that points to any extra state that is needed depending on the redundancy scheme.

The remaining fields describe the group of processes that make up the redundancy set for a particular process. For a given redundancy descriptor, the entire set of processes in the run is divided into distinct groups, and each of these groups is assigned a unique integer id called the group id. The set of group ids may not be contiguous. Each process knows the total number of groups, which is recorded in the groups field, as well as, the id of the group the process is a member of, which is recorded in the group_id field.

Since the processes within a group communicate frequently, SCR creates a communicator for each group. The comm field is a handle to the MPI communicator that defines the group the process is a member of. The my_rank and ranks fields cache the rank of the process in this communicator and the number of processes in this communicator, respectively.

If the redundancy scheme requires additional information to be kept in the redundancy descriptor, it allocates additional memory and records a pointer to it via the copy_state pointer.

Extra state for PARTNER

The SCR_COPY_PARTNER scheme allocates the following structure:

```
typedef struct {
  int       lhs_rank;       /* rank which is one less (with wrap to highest) within␣
↪set */
  int       lhs_rank_world; /* rank of lhs process in comm world */
  char*     lhs_hostname;   /* hostname of lhs process */
  int       rhs_rank;       /* rank which is one more (with wrap to lowest) within␣
↪set */
  int       rhs_rank_world; /* rank of rhs process in comm world */
  char*     rhs_hostname;   /* hostname of rhs process */
} scr_reddesc_partner;
```

For SCR_COPY_PARTNER, the processes within a group form a logical ring, ordered by their rank in the group. Each process has a left and right neighbor in this ring. The left neighbor is the process whose rank is one less than the current process, and the right neighbor is the process whose rank is one more. The last process in the group wraps back around to the first. SCR caches information about the ranks to the left and right of a process. The lhs_rank, lhs_rank_world, and lhs_hostname fields describe the rank to the left of the process, and the rhs_rank, rhs_rank_world, and rhs_hostname fields describe the rank to the right. The lhs_rank and rhs_rank fields record the ranks of the neighbor processes in comm. The lhs_rank_world

and rhs_rank_world fields record the ranks of the neighbor processes in scr_comm_world. Finally, the lhs_hostname and rhs_hostname fields record the hostnames where those processes are running.

Extra state for XOR

The SCR_COPY_XOR scheme allocates the following structure:

```
typedef struct {
  scr_hash* group_map;      /* hash that maps group rank to world rank */
  int       lhs_rank;       /* rank which is one less (with wrap to highest) within␣
↪set */
  int       lhs_rank_world; /* rank of lhs process in comm world */
  char*     lhs_hostname;   /* hostname of lhs process */
  int       rhs_rank;       /* rank which is one more (with wrap to lowest) within␣
↪set */
  int       rhs_rank_world; /* rank of rhs process in comm world */
  char*     rhs_hostname;   /* hostname of rhs process */
} scr_reddesc_xor;
```

The fields here are similar to the fields of SCR_COPY_PARTNER with the exception of an additional group_map field, which records a hash that maps a group rank to its rank in MPI_COMM_WORLD.

2.9.3 Example redundancy descriptor hash

Each redundancy descriptor can be stored in a hash. Here is an example redundancy descriptor hash.

```
ENABLED
  1
INDEX
  0
INTERVAL
  1
BASE
  /tmp
DIRECTORY
  /tmp/user1/scr.1145655/index.0
TYPE
  XOR
HOP_DISTANCE
  1
SET_SIZE
  8
GROUPS
  1
GROUP_ID
  0
GROUP_SIZE
  4
GROUP_RANK
  0
```

Most field names in the hash match field names in the C struct, and the meanings are the same. The one exception is GROUP_RANK, which corresponds to my_rank in the struct. Note that not all fields from the C struct are recorded in the hash. At runtime, it's possible to reconstruct data for the missing struct fields using data from the hash. In particular, one may recreate the group communicator by calling MPI_Comm_split() on scr_comm_world specifying

the GROUP_ID value as the color and specifying the GROUP_RANK value as the key. After recreating the group communicator, one may easily find info for the left and right neighbors.

2.9.4 Example redundancy descriptor configuration file entries

SCR must be configured with redundancy schemes. By default, SCR protects against single compute node failures using XOR, and it caches one checkpoint in /tmp. To specify something different, edit a configuration file to include checkpoint descriptors. Checkpoint descriptors look like the following.

```
# instruct SCR to use the CKPT descriptors from the config file
SCR_COPY_TYPE=FILE

# the following instructs SCR to run with three checkpoint configurations:
# - save every 8th checkpoint to /ssd using the PARTNER scheme
# - save every 4th checkpoint (not divisible by 8) to /ssd using XOR with
#   a set size of 8
# - save all other checkpoints (not divisible by 4 or 8) to /tmp using XOR with
#   a set size of 16
CKPT=0 INTERVAL=1 GROUP=NODE   STORE=/tmp TYPE=XOR     SET_SIZE=16
CKPT=1 INTERVAL=4 GROUP=NODE   STORE=/ssd TYPE=XOR     SET_SIZE=8
CKPT=2 INTERVAL=8 GROUP=SWITCH STORE=/ssd TYPE=PARTNER
```

First, one must set the SCR_COPY_TYPE parameter to "FILE". Otherwise, an implied checkpoint descriptor is constructed using various SCR parameters including SCR_GROUP, SCR_CACHE_BASE, SCR_COPY_TYPE, and SCR_SET_SIZE.

Checkpoint descriptor entries are identified by a leading CKPT key. The values of the CKPT keys must be numbered sequentially starting from 0. The INTERVAL key specifies how often a checkpoint is to be applied. For each checkpoint, SCR selects the descriptor having the largest interval value that evenly divides the internal SCR checkpoint iteration number. It is necessary that one descriptor has an interval of 1. This key is optional, and it defaults to 1 if not specified. The GROUP key lists the failure group, i.e., the name of the group of processes likely to fail. This key is optional, and it defaults to the value of the SCR_GROUP parameter if not specified. The STORE key specifies the directory in which to cache the checkpoint. This key is optional, and it defaults to the value of the SCR_CACHE_BASE parameter if not specified. The TYPE key identifies the redundancy scheme to be applied. This key is optional, and it defaults to the value of the SCR_COPY_TYPE parameter if not specified.

Other keys may exist depending on the selected redundancy scheme. For XOR schemes, the SET_SIZE key specifies the minimum number of processes to include in each XOR set.

2.9.5 Common functions

This section describes some of the most common redundancy descriptor functions. The implementation can be found in scr.c.

Initializing and freeing redundancy descriptors

Initialize a redundancy descriptor structure (clear its fields).

```
struct scr_reddesc desc;
scr_reddesc_init(&desc)
```

Free memory associated with a redundancy descriptor.

```
scr_reddesc_free(&desc)
```

Redundancy descriptor array

Allocate and fill in `scr_reddescs` array using redundancy descriptor hashes provided in `scr_reddesc_hash`.

```
scr_reddescs_create()
```

Free the list of redundancy descriptors.

```
scr_reddescs_free()
```

Select a redundancy descriptor for a specified checkpoint id from among the `ndescs` descriptors in the array of descriptor structs pointed to by `descs`.

```
struct scr_reddesc* desc = scr_reddesc_for_checkpoint(ckpt, ndescs, descs)
```

Converting between structs and hashes

Convert a redundancy descriptor struct to its equivalent hash.

```
scr_reddesc_store_to_hash(desc, hash)
```

This function clears any entries in the specified hash before setting fields according to the struct.

Given a redundancy descriptor hash, build and fill in the fields for its equivalent redundancy descriptor struct.

```
scr_reddesc_create_from_hash(desc, index, hash)
```

This function creates a communicator for the redundancy group and fills in neighbor information relative to the calling process. Note that this call is collective over `scr_comm_world`, because it creates a communicator. The index value specified in the call is overridden if an index field is set in the hash.

Interacting with filemaps

Redundancy descriptor hashes are cached in filemaps. There are functions to set, get, and unset a redundancy descriptor hash in a filemap for a given dataset id and rank id (Section *Filemap redundancy descriptors*). There are additional functions to extract info from a redundancy descriptor hash that is stored in a filemap.

For a given dataset id and rank id, return the base directory associated with the redundancy descriptor stored in the filemap.

```
char* basedir = scr_reddesc_base_from_filemap(map, dset, rank)
```

For a given dataset id and rank id, return the path associated with the redundancy descriptor stored in the filemap in which dataset directories are to be created.

```
char* dir = scr_reddesc_dir_from_filemap(map, dset, rank)
```

For a given dataset id and rank id, fill in the specified redundancy descriptor struct using the redundancy descriptor stored in the filemap.

```
scr_reddesc_create_from_filemap(map, dset, rank, desc)
```

Note that this call is collective over `scr_comm_world`, because it creates a communicator.

2.10 Redundancy schemes

2.10.1 SINGLE

With the `SINGLE` redundancy scheme, SCR keeps a single copy of each dataset file. It tracks meta data on application files, but no redundancy data is stored. The communicator in the redundancy descriptor is a duplicate of `MPI_COMM_SELF`. During a restart, files are distributed according to the current process mapping, but if a failure leads to a loss of files, affected datasets are simply deleted from the cache.

Although the lack of redundancy prevents `SINGLE` from being useful in cases where access to storage is lost, in practice many failures are due to application-level software which do not impact storage accessibility. This scheme is also useful when writing to highly reliable storage.

2.10.2 PARTNER

TODO: This scheme assumes node-local storage.

With `PARTNER` a full copy of every dataset file is made. The partner process is always selected from a different failure group, so that it's unlikely for a process and its partner to fail at the same time.

When creating the redundancy descriptor, SCR splits `scr_comm_world` into subcommunicators each of which contains at most one process from each failure group. Within this communicator, each process picks the process whose rank is one more than its own (right-hand side) to send its copies, and it stores copies of the files from the process whose rank is one less (left-hand side). Processes at the end of the communicator wrap around to find partners. The hostname, the rank within the redundancy communicator, and the global rank of the left and right neighbors are stored in the `copy_state` field of the redundancy descriptor. This is all implemented in `scr_reddesc_create()` and `scr_reddesc_create_partner()` in `scr_reddesc.c`.

When applying the redundancy scheme, each process sends its files to its right neighbor. The meta data for each file is transferred and stored, as well as, the redundancy descriptor hash for the process. Each process writes the copies to the same dataset directory in which it wrote its own original files. Note that if a process and its partner share access to the same storage, then this scheme should not be used.

Each process also records the name of the node for which it is serving as the partner. This information is used during a scavenge in order to target the partner node for a copy if the source node has failed. This is a useful optimization when the cache is in node-local storage.

During the distribute phase of a restart, a process may obtain its files from either the original copy or the partner copy. If neither is available, the distribute fails and the dataset is deleted from cache. If the distribute phase succeeds, the `PARTNER` scheme is immediately applied again to restore the redundancy.

During the first round of a scavenge, only original files are copied from cache to the parallel file system. If the scavenge fails to copy data from some nodes, the second round attempts to target just the relevant partner nodes which it identified from the partner key recorded in the filemap. This optimization avoids unnecessarily copying every files twice, the original plus its copy.

2.11 XOR

The XOR redundancy scheme divides the set of processes in the run into subsets, called *XOR sets*. For each dataset, each process in a set computes and stores redundancy data in an *XOR file*. This file is stored in the dataset subdirectory within the cache directory along side any files that the application process writes.

The XOR redundancy scheme is designed such that the dataset files for any single member of a set can be reconstructed using the dataset files and XOR files from all remaining members. Thus, all dataset files can be recovered, even if the files for one process from each set are lost. On the other hand, if any set loses files for two or more processes, the XOR redundancy scheme cannot recover all files.

The processes within each set are ordered, and each process has a *rank* in the set, counting up from 0. The process whose rank in the set is one less than the rank of the current process is called the *left neighbor*, and the process whose rank is one more is the *right neighbor*. The last rank wraps back to the first to form a ring. At run time, the library caches the XOR set in the MPI communicator associated with a redundancy descriptor. Each process also caches information abouts its left and right neighbor processes in the redundancy descriptor.

SCR ensures that it does not select two processes from the same failure group to be in the same XOR set. The SCR_SET_SIZE parameter determines the minimum number of processes to include in a set. The selection algorithm is implemented in `scr_reddesc_create()` and `scr_reddesc_create_xor()` in `scr_reddesc.c`, as well as `scr_set_partners()` in `scr_util_mpi.c`.

2.11.1 XOR algorithm

The XOR redundancy scheme applies the algorithm described in [Ross] (which is based on [RAID5]). Assuming that each process writes one file and that the files on all processes are the same size, this algorithm is illustrated in Figure *1*. Given N processes in the set, each file is logically partitioned into $N-1$ chunks, and an empty, zero-padded chunk is logically inserted into the file at alternating positions depending on the rank of the process. Then a reduce-scatter is computed across the set of logical files. The resulting chunk from this reduce-scatter is the data that the process stores in its XOR file.

In general, different processes may write different numbers of files, and file sizes may be arbitrary. In Figure *Extension to multiple files*, we illustrate how to extend the algorithm for the general case. First, we logically concatenate all of the files a process writes into a single file. We then compute the minimum chunk size such that $N-1$ chunks are equal to or larger than the largest logical file. Finally, we pad the end of each logical file with zeros, such that each logical file extends to the number of bytes contained in $N-1$ chunks. This extension is efficient when all processes write about the same amount of data.

In practice, to read from this logical file, we first open each physical file, and then we call `scr_read_pad_n()`. As input, this function takes an array of file names, an array of file sizes, and an array of opened file descriptors, along with an integer defining how many elements are in each array, as well as, an offset and the number of bytes to read. It returns data as if the set of files were concatenated as a single file in the order specified by the arrays. This read also pads the end of the concatenated file with zeros if the read extends past the amount of real data. There is a corresponding `scr_write_pad_n()` function to issue writes to this logical file. These functions are implemented in `scr_io.c`.

This way, we can operate as though each process has exactly one file, where each file has the same length and is evenly divisible by $N-1$. For an efficient reduce-scatter implementation, we use an algorithm that achieves the following goals:

1. Evenly distributes the work among all processes in the set.
2. Structures communication so that a process always receives data from its left neighbor and sends data to its right neighbor. This is useful to eliminate network contention.
3. Only reads data from each checkpoint file once, and only writes data to the XOR file once. This minimizes file accesses, which may be slow.

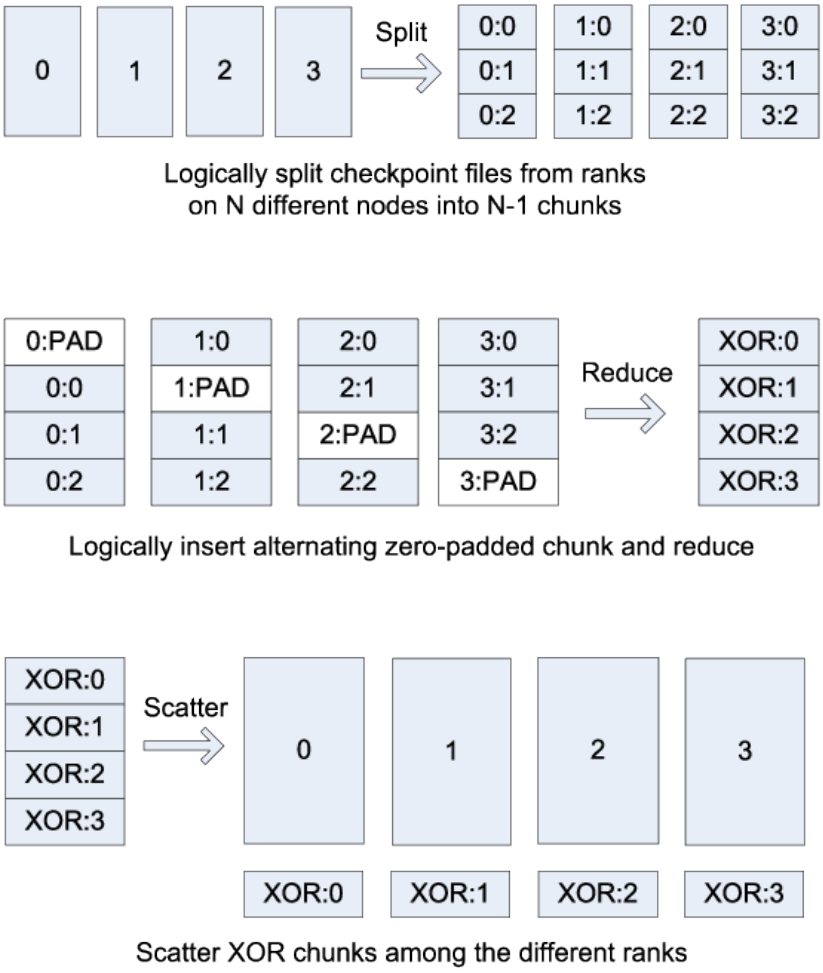

Fig. 3: XOR reduce-scatter

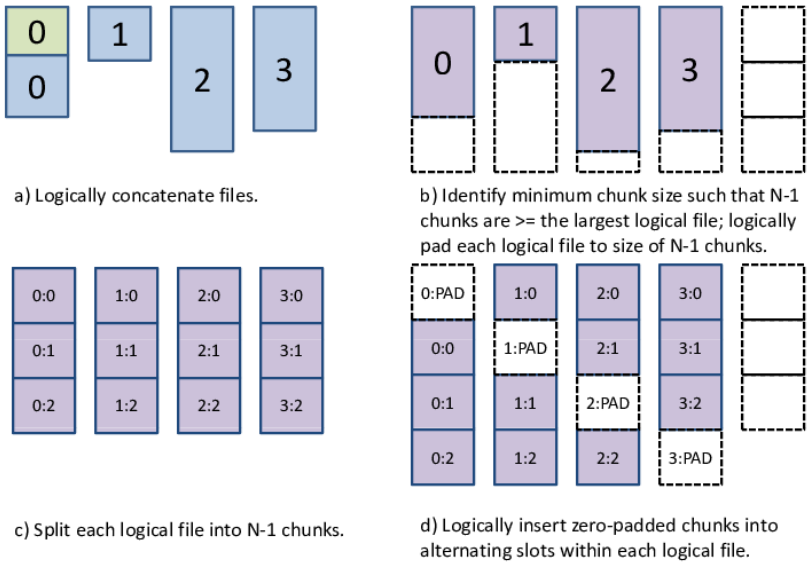

Fig. 4: Extension to multiple files

4. Operates on the data in small pieces, so that the working set fits within the processor's cache.

To accomplish this, we divide each chunk into a series of smaller pieces, and we operate on each piece in phases. In the first phase, we compute the reduce-scatter result for the first piece of all chunks. Then, in the second phase, we compute the reduce-scatter result for the second piece of all chunks, and so on. In each phase, the reduce-scatter computation is pipelined among the processes. The first phase of this reduce-scatter algorithm is illustrated in Figure *XOR reduce-scatter implementation*. This algorithm is implemented in `scr_reddesc_apply_xor()` in `scr_reddesc_apply.c`.

2.11.2 XOR file

The `XOR` file contains a header, which is stored as a hash, followed by the `XOR` chunk data, which is stored as binary data. The header provides information on the process that wrote the file, meta data for the process's files, and the group of processes that belong to its `XOR` set. SCR also makes a copy of the meta data for a process's files in the header of the `XOR` file written by the process's right neighbor. This way, SCR can recover all meta data even if one `XOR` file is lost. An example header is shown below:

```
DSET
  COMPLETE
    1
  SIZE
    2097182
  FILES
    4
  ID
    6
  NAME
    scr.dataset.6
```

(continues on next page)

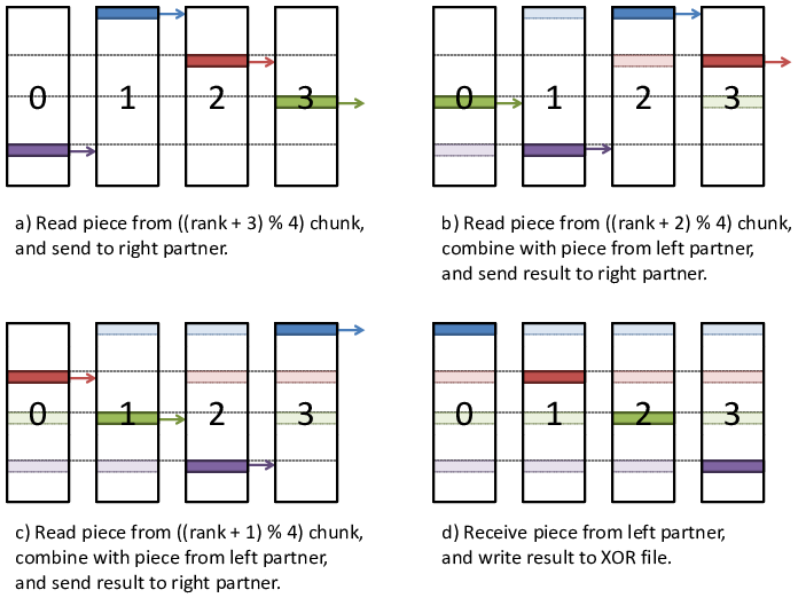

Fig. 5: XOR reduce-scatter implementation

(continued from previous page)

```
  CREATED
    1312850690668536
  USER
    user1
  JOBNAME
    simulation123
  JOBID
    112573
  CKPT
    6
RANKS
  4
GROUP
  RANKS
    4
  RANK
    0
      0
    1
      1
    2
      2
    3
      3
CHUNK
  174766
CURRENT
  RANK
    3
```

(continues on next page)

(continued from previous page)

```
  FILES
    1
  FILE
    0
      FILE
        rank_3.ckpt
      TYPE
        FULL
      RANKS
        4
      ORIG
        rank_3.ckpt
      PATH
        /p/lscratchb/user1/simulation123
      NAME
        rank_3.ckpt
      SIZE
        524297
      COMPLETE
        1
PARTNER
  RANK
    2
  FILES
    1
  FILE
    0
      FILE
        rank_2.ckpt
      TYPE
        FULL
      RANKS
        4
      ORIG
        rank_2.ckpt
      PATH
        /p/lscratchb/user1/simulation123
      NAME
        rank_2.ckpt
      SIZE
        524296
      COMPLETE
        1
```

The topmost `DSET` field records the dataset descriptor the `XOR` file belongs to, and the topmost `RANKS` field records the number of ranks in the run (i.e., the size of `scr_comm_world`). The `GROUP` hash records the set of processes in the `XOR` set. The number of processes in the set is listed under the `RANKS` field, and a mapping of a process's rank in the group to its rank in `scr_comm_world` is stored under the `RANK` hash. The size of the `XOR` chunk in number of bytes is specified in the `CHUNK` field.

Then, the meta data for the checkpoint files written by the process are recorded under the `CURRENT` hash, and a copy of the meta data for the checkpoint files written by the left neighbor are recorded under the `PARTNER` hash. Each hash records the rank of the process (in `scr_comm_world`) under `RANK`, the number of checkpoint files the process wrote under `FILES`, and a ordered list of meta data for each file under the `FILE` hash. Each checkpoint file is assigned an integer index, counting up from 0, which specifies the order in which the files were logically concatenated to compute the `XOR` chunk. The meta data for each file is then recorded under its index.

At times, XOR files from different processes reside in the same directory, so SCR specifies a unique name for the XOR file on each process. Furthermore, SCR encodes certain information in the file name to simplify the task of grouping files belonging to the same set. SCR assigns a unique integer id to each XOR set. To select this id, SCR computes the minimum rank in scr_comm_world of all processes in the set and uses that rank as the set id. SCR then incorporates a process's rank within its set, the size of its set, and its set id into its file name, such that the XOR file name is of the form: <grouprank+1>_of_<groupsize>_in_<groupid>.xor.

2.11.3 XOR rebuild

SCR provides two different methods to rebuild files using the XOR scheme. If a run is restarted and a dataset is stored in cache, then SCR rebuilds files during SCR_Init(). On the other hand, at the end of an allocation, SCR can rebuild files after scavenging a dataset from cache. This section discusses the method used in SCR_Init(). For discussion on rebuilding during a scavenge, see Sections *Scavenge* and *Program Flow>Scavenge*.

During SCR_Init() in a restarted run, SCR uses MPI to rebuild files in parallel. The processes in each set check whether they need to and whether they can rebuild any missing files. If so, the processes identify which rank in the set needs its files rebuilt. This rank is then set as the root of a reduction over the data in the remaining application files and XOR files to reconstruct the missing data. SCR implements a reduction algorithm that achieves the same goals as the reduce-scatter described in Section *0.1.1*. Namely, the implementation attempts to distribute work evenly among all processes, minimize network contention, and minimize file accesses. This algorithm is implemented in scr_reddesc_recover_xor() in scr_reddesc_recover.c. An example is illustrated in Figure *Pipelined XOR reduction to root*.

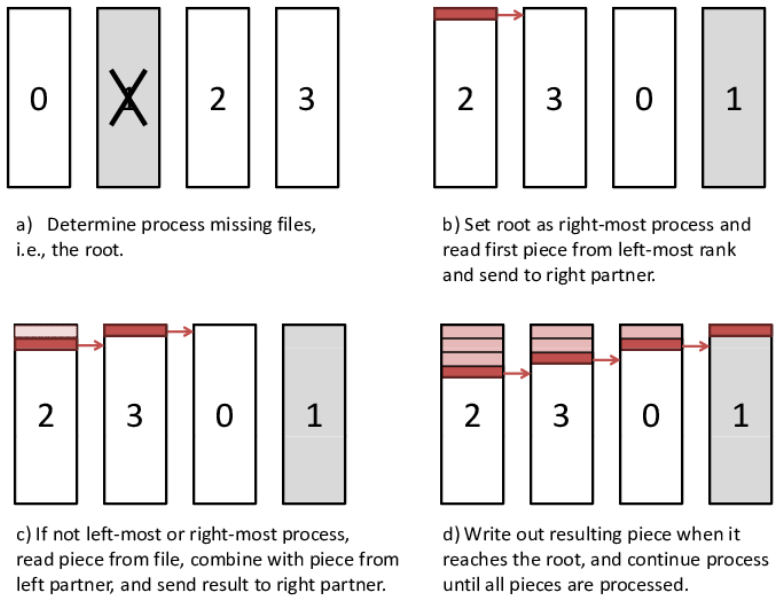

Fig. 6: Pipelined XOR reduction to root

2.12 Containers

NOTE: This feature is experimental and not yet complete, so it it not documented in the user guide.

SCR requires checkpoint data to be stored primarily as a file per process. However, writing a large number of files is inefficient or difficult to manage on some file systems. To alleviate this problem, SCR provides an abstraction called "containers". When writing data to or reading data from the prefix directory, SCR combines multiple application files into a container. Containers are disabled by default. To enable them, set the `SCR_USE_CONTAINERS` parameter to 1.

During a flush, SCR identifies the containers and the offsets within those containers where each file should be stored. SCR records the file-to-container mapping in the rank2file map, which it later references to extract files during the fetch operation.

A container has a maximum size, which is determined by the `SCR_CONTAINER_SIZE` parameter. This parameter defaults to 100GB. Application file data is packed sequentially within a container until the container is full, and then the remaining data spills over to the next container. The total number of containers required depends on the total number of bytes in the dataset and the container size. A container file name is of the form `ctr.<id>.scr`, where `<id>` is the container id which counts up from 0. All containers are written to the dataset directory within the prefix directory.

SCR combines files in an order such that all files on the same node are grouped sequentially. This limits the number of files that each compute node must access. For this purpose, SCR creates two global communicators during `SCR_Init`. Both are defined in `scr_globals.c`. The `scr_comm_node` communicator consists of all processes on the same compute node. The `scr_comm_node_across` communicator consists of all processes having the same rank within `scr_comm_node`. Note that some process has rank 0 in `scr_comm_node` for each node in the run. This process is called the "node leader".

To get the offset where each process should write its data, SCR first sums up the sizes of all files on the node via a reduce on `scr_comm_node`. The node leaders then execute a scan across nodes using the `scr_comm_node_across` communicator to get a node offset. A final scan within `scr_comm_node` produces the offset at which each process should write its data.

TODO: discuss setting in flush descriptor stored in filemap under dataset id and rank

TODO: discuss containers during a scavenge

TODO: should we copy redundancy data to containers as well?

Within a rank2file map file, the file-to-container map adds entries under the `SEG` key for each file. An example entry looks like the following:

```
rank_2.ckpt
  SEG
    0
      FILE
        .scr/ctr.1.scr
      OFFSET
        224295
      LENGTH
        75705
    1
      FILE
        .scr/ctr.2.scr
      OFFSET
        0
      LENGTH
        300000
    2
      FILE
        .scr/ctr.3.scr
      OFFSET
        0
```

(continues on next page)

(continued from previous page)

```
      LENGTH
        148591
```

The SEG key specifies file data as a list of numbered segments starting from 0. Each segment specifies the length of file data, and the name and offset at which it can be found within a container file. Reading all segments in order produces the full sequence of bytes that make up the file. The name of the container file is given as a relative path from the dataset directory.

In the above example, the container size is set to 300000. This size is smaller than normal to illustrate the various fields. The data for the `rank_2.ckpt` file is split among three segments. The first segment of 75705 bytes is in the container file named `.scr/ctr.1.scr` starting at offset 224295. The next segment is 300000 bytes and is in `.scr/ctr.2.scr` starting at offset 0. The final segment of 148591 bytes are in `.scr/ctr.3.scr` starting at offset 0.

2.13 Scavenge

At the end of an allocation, certain SCR commands inspect the cache to verify that the most recent checkpoint has been copied to the parallel file system. If not, these commands execute other SCR commands to scavenge this checkpoint before the allocation ends. In this section, we detail key concepts referenced as part of the scavenge operations. Detailed program flow for these operations is provided in Section *Program Flow>Scavenge*.

2.13.1 Rank filemap file

The `scr_copy` command is a serial program (non-MPI) that executes on a compute node and copies all files belonging to a specified dataset id from the cache to a specified dataset directory on the parallel file system. It is implemented in `scr_copy.c` whose program flow is described in Section <scr_copy>. The `scr_copy` command copies all application files and SCR redundancy data files. In addition, it writes a special filemap file for each rank to the dataset directory. The name of this filemap file is of the format: `<rank>.scrfilemap`. An example hash for such a filemap file is shown below:

```
DSET
  6
    RANK
      2
RANK
  2
    DSET
      6
        DSETDESC
          COMPLETE
            1
          SIZE
            2097182
          FILES
            4
          ID
            6
          NAME
            scr.dataset.6
          CREATED
            1312850690668536
          USER
```

(continues on next page)

(continued from previous page)

```
      user1
    JOBNAME
      simulation123
    JOBID
      112573
    CKPT
      6
  FILES
    2
  FILE
    3_of_4_in_0.xor
      META
        RANKS
          4
        COMPLETE
          1
        SIZE
          175693
        TYPE
          XOR
        FILE
          3_of_4_in_0.xor
        CRC
          0x2ef519a1
    rank_2.ckpt
      META
        COMPLETE
          1
        SIZE
          524296
        NAME
          rank_2.ckpt
        PATH
          /p/lscratchb/user1/simulation123
        ORIG
          rank_2.ckpt
        RANKS
          4
        TYPE
          FULL
        FILE
          rank_2.ckpt
        CRC
          0x738bb68f
```

It lists the files owned by a rank for a particular dataset. In this case, it shows that rank `2` wrote two files (`FILES=2`) as part of dataset id `6`. Those files are named `rank_2.ckpt` and `3_of_4_in_0.xor`.

This format is similar to the filemap hash format described in Section *Filemap*. The main differences are that files are listed using relative paths instead of absolute paths and there are no redundancy descriptors. The paths are relative so that the dataset directory on the parallel file system may be moved or renamed. Redundancy descriptors are cache-specific, so these entries are excluded.

2.13.2 Scanning files

After `scr_copy` copies files from the cache on each compute node to the parallel file system, the `scr_index` command runs to check whether all files were recovered, rebuild missing files if possible, and add an entry for the dataset to the SCR index file (Section *Index_file*). When invoking the `scr_index` command, the full path to the prefix directory and the name of the dataset directory are specified on the command line. The `scr_index` command is implemented in `scr_index.c`, and its program flow is described in Section <scr_index>.

The `scr_index` command first acquires a listing of all items contained in the dataset directory by calling `scr_read_dir`, which is implemented in `scr_index.c`. This function uses POSIX calls to list all files and subdirectories contained in the dataset directory. The hash returned by this function distinguishes directories from files using the following format.

```
DIR
  <dir1>
  <dir2>
  ...
FILE
  <file1>
  <file2>
  ...
```

The `scr_index` command then iterates over the list of file names and reads each file that ends with the "`.scrfilemap`" extension. These files are the filemap files written by `scr_copy` as described above. The `scr_index` command records the number of expected files for each rank into a single hash called the *scan hash*.

For each file listed in the rank filemap file, the `scr_index` command verifies the meta data from the rank filemap map against the original file (excluding CRC32 checks). If the file passes these checks, the command adds a corresponding entry for the file to the scan hash. This entry is formatted such that it can be used as an entry in the summary file hash (Section *Summary file*). If the file is an `XOR` file, it sets a `NOFETCH` flag under the `FILE` key, which instructs the SCR library to exclude this file during a fetch operation.

Furthermore, for each `XOR` file, the `scr_index` command extracts info about the `XOR` set from the file name and adds an entry under an `XOR` key in the scan hash. It records the `XOR` set id (under `XOR`), the number of members in the set (under `MEMBERS`), and the group rank of the current file in this set (under `MEMBER`), as well as, the global rank id (under `RANK`) and the name of the `XOR` file (under `FILE`). After this all of this, the scan hash might look like the following example:

```
DLIST
  <dataset_id>
    DSET
      COMPLETE
        1
      SIZE
        2097182
      FILES
        4
      ID
        6
      NAME
        scr.dataset.6
      CREATED
        1312850690668536
      USER
        user1
      JOBNAME
        simulation123
```

(continues on next page)

(continued from previous page)

```
    JOBID
      112573
    CKPT
      6
  RANK2FILE
    RANKS
      <num_ranks>
    RANK
      <rank1>
        FILES
          <num_expected_files_for_rank1>
        FILE
           <filename>
             SIZE
               <filesize>
             CRC
               <crc>
           <xor_filename>
             NOFETCH
             SIZE
               <filesize>
             CRC
               <crc>
           ...
      <rank2>
        FILES
          <num_expected_files_for_rank2>
        FILE
           <filename>
             SIZE
               <filesize>
             CRC
               <crc>
           <xor_filename>
             NOFETCH
             SIZE
               <filesize>
             CRC
               <crc>
           ...
      ...
  XOR
    <set1>
      MEMBERS
        <num_members_in_set1>
      MEMBER
        <member1>
          FILE
            <xor_filename_of_member1_in_set1>
          RANK
            <rank_id_of_member1_in_set1>
        <member2>
          FILE
            <xor_filename_of_member2_in_set1>
          RANK
            <rank_id_of_member2_in_set1>
        ...
```

(continues on next page)

(continued from previous page)

```
      <set2>
        MEMBERS
          <num_members_in_set2>
        MEMBER
          <member1>
            FILE
              <xor_filename_of_member1_in_set2>
            RANK
              <rank_id_of_member1_in_set2>
          <member2>
            FILE
              <xor_filename_of_member2_in_set2>
            RANK
              <rank_id_of_member2_in_set2>
          ...
      ...
```

2.13.3 Inspecting files

After merging data from all filemap files in the dataset directory, the `scr_index` command inspects the scan hash to identify any missing files. For each dataset, it determines the number of ranks associated with the dataset, and it checks that it has an entry in the scan hash for each rank. It then checks whether each rank has as an entry for each of its expected number of files. If any file is determined to be missing, the command adds an `INVALID` flag to the scan hash, and it lists all ranks that are missing files under the `MISSING` key. This operation may thus add entries like the following to the scan hash.

```
DLIST
  <dataset_id>
    INVALID
    MISSING
      <rank1>
      <rank2>
      ...
```

2.13.4 Rebuilding files

If any ranks are missing files, then the `scr_index` command attempts to rebuild files. Currently, only the `XOR` redundancy scheme can be used to rebuild files. The command iterates over each of the `XOR` sets listed in the scan hash, and it checks that each set has an entry for each of its members. If it finds an `XOR` set that is missing a member, or if it finds that a set contains a rank which is known to be missing files, the command constructs a string that can be used to fork and exec a process to rebuild the files for that process. It records these strings under the `BUILD` key in the scan hash. If it finds that one or more files cannot be recovered, it sets an `UNRECOVERABLE` flag in the scan hash. If the `scr_index` command determines that it is possible to rebuild all missing files, it forks and execs a process for each string listed under the `BUILD` hash. Thus this operation may add entries like the following to the scan hash.

```
DLIST
  <dataset_id>
    UNRECOVERABLE
    BUILD
      <cmd_to_rebuild_files_for_set1>
      <cmd_to_rebuild_files_for_set2>
      ...
```

2.13.5 Scan hash

After all of these steps, the scan hash is of the following form:

```
DLIST
  <dataset_id>
    UNRECOVERABLE
    BUILD
      <cmd_to_rebuild_files_for_set1>
      <cmd_to_rebuild_files_for_set2>
      ...
    INVALID
    MISSING
      <rank1>
      <rank2>
      ...
    RANKS
      <num_ranks>
    RANK
      <rank>
        FILES
          <num_files_to_expect>
        FILE
          <file_name>
            SIZE
              <size_in_bytes>
            CRC
              <crc32_string_in_0x_form>
          <xor_file_name>
            NOFETCH
            SIZE
              <size_in_bytes>
            CRC
              <crc32_string_in_0x_form>
          ...
      ...
    XOR
      <xor_setid>
        MEMBERS
          <num_members_in_set>
        MEMBER
          <member_id>
            FILE
              <xor_filename>
            RANK
              <rank>
          ...
      ...
```

After the rebuild attempt, the `scr_index` command writes a summary file in the dataset directory. To produce the hash for the summary file, the command deletes extraneous entries from the scan hash (`UNRECOVERABLE`, `BUILD`, `INVALID`, `MISSING`, `XOR`) and adds the summary file format version number.

2.14 Logging

If enabled, the SCR library and the SCR scripts log different events, recording things like:

- start time and end time of each application run
- time consumed to write each checkpoint/output dataset
- time consumed to transfer each dataset from cache to the file system
- any application restarts
- any node failures

This info can enable one to do things like:

- Gather stats about datasets, including the number of processes used, number of files, and total byte size of each dataset, which could help inform decisions about cache storage requirements for current/future systems.
- Gather stats about SCR and I/O system performance and variability.
- Compute stats about application interrupts on a machine.
- Compute optimal checkpoint frequency for each application. For this, there is a script that parses log entries from the text log file and computes the optimal checkpoint frequency using the Young or Daly formulas. The goal is to integrate a script like that into the scr_srun script to set an application checkpoint interval dynamically based on what the application is experiencing on each system: `scr_ckpt_interval.py`.

There are three working logging mechanisms. One can use them in any combination in a run:

- Text file written to the application's SCR prefix directory. This is most useful for end users.
- Messages written to syslog. This collects log messages from all jobs running on the system, so it is most useful to system support staff.
- Records written to a MySQL database. This could be used by either an end user or the system support staff. To create the MySQL database, see `scr.mysql`.

2.14.1 Settings

There are settings for each logging mechanism:

- `SCR_LOG_ENABLE` - If 0, this disables *all* logging no matter what other settings are. It is there as an easy way to turn off all logging. If set to 1, then logging depends on other settings below.
- Text-based logging:
  - `SCR_LOG_TXT_ENABLE` - If 1, the text log file is enabled.
- Syslog-based logging:
  - `SCR_LOG_SYSLOG_ENABLE` - If 1, syslog messages are enabled. There are some associated configure-time settings.
  - `-DSCR_LOG_SYSLOG_FACILITY=[facility]` : Facility for syslog messages (see `man openlog`), defaults to `LOG_LOCAL7`
  - `-DSCR_LOG_SYSLOG_LEVEL=[level]` : Level for syslog messages (see `man openlog`), defaults to `LOG_INFO`
  - `-DSCR_LOG_SYSLOG_PREFIX=[str]` : Prefix string to prepend to syslog messages, defaults to `"SCR"`
- MySQL-based logging:
  - `SCR_LOG_DB_ENABLE` - If 1, MySQL logging is enabled.
  - `SCR_LOG_DB_DEBUG` - If 1, echo SQL statements to `stdout` to help when debugging MySQL problems.

- SCR_LOG_DB_HOST - Hostname of MySQL server.
- SCR_LOG_DB_NAME - Database name on MySQL server.
- SCR_LOG_DB_USER - Username for accessing database.
- SCR_LOG_DB_PASS - Password for accessing database.

2.15 Index file

The index file records information about each of the datasets stored in the prefix directory on the parallel file system. It is stored in the prefix directory. Internally, the data of the index file is organized as a hash. Here are the contents of an example index file.

```
VERSION
  1
CURRENT
  scr.dataset.18
DIR
  scr.dataset.18
    DSET
      18
  scr.dataset.12
    DSET
      12
DSET
  18
    DIR
      scr.dataset.18
        COMPLETE
          1
        DSET
          ID
            18
          NAME
            scr.dataset.18
          CREATED
            1312853507675143
          USER
            user1
          JOBNAME
            simulation123
          JOBID
            112573
          CKPT
            18
          FILES
            4
          SIZE
            2097182
          COMPLETE
            1
        FLUSHED
          2011-08-08T18:31:47
  12
    DIR
      scr.dataset.12
```

(continues on next page)

(continued from previous page)

```
        FETCHED
          2011-08-08T18:31:47
        FLUSHED
          2011-08-08T18:30:30
        COMPLETE
          1
        DSET
          COMPLETE
            1
          SIZE
            2097182
          FILES
            4
          ID
            12
          NAME
            scr.dataset.12
          CREATED
            1312853406814268
          USER
            user1
          JOBNAME
            simulation123
          JOBID
            112573
          CKPT
            12
```

The VERSION field records the version number of file format of the index file. This enables future SCR implementations to change the format of the index file while still allowing SCR to read index files written by older implementations.

The CURRENT field specifies the name of a dataset directory. When restarting a job, SCR starts with this directory. It then works backwards from this directory, searching for the most recent checkpoint (the checkpoint having the highest id) that is thought to be complete and that has not failed a previous fetch attempt.

The DIR hash is a simple index which maps a directory name to a dataset id.

The information for each dataset is indexed by dataset id under the DSET hash. There may be multiple copies of a given dataset id, each stored within a different dataset directory within the prefix directory. For a given dataset id, each copy is indexed by directory name under the DIR hash. For each directory, SCR tracks whether the set of dataset files is thought to be complete (COMPLETE), the timestamp at which the dataset was copied to the parallel file system (FLUSHED), timestamps at which the dataset (checkpoint) was fetched to restart a job (FETCHED), and timestamps at which a fetch attempt of this dataset (checkpoint) failed (FAILED).

2.16 Summary file

The summary file tracks global properties of a dataset, such as dataset id, its size, the total number of files, and the time it was created. It is stored in the dataset directory on the parallel file system. Internally, the data of the summary file is organized as a hash. Here are the contents of an example summary file.

```
VERSION
  6
COMPLETE
```

(continues on next page)

(continued from previous page)

```
  1
DSET
  ID
    18
  NAME
    scr.dataset.18
  CREATED
    1312853507675143
  USER
    user1
  JOBNAME
    simulation123
  JOBID
    112573
  CKPT
    18
  FILES
    4
  SIZE
    2097182
  COMPLETE
    1
```

The `VERSION` field records the version number of file format of the summary file. This enables future SCR implementations to change the format of the summary file while still allowing SCR to read summary files written by older implementations.

A `COMPLETE` flag concisely denotes whether all files for this dataset are thought to be valid. The properties of the dataset are then contained within the `DSET` hash.

When fetching a checkpoint upon a restart, rank 0 reads the summary file and broadcasts its contents to the other ranks.

2.17 Rank2file map

The rank2file map tracks which files were written by which ranks during a particular dataset. This map contains information for every rank and file. For large jobs, it may consist of more bytes than can be loaded into any single MPI process. This information is scattered among multiple files that are organized as a tree. These files are stored in the dataset directory on the parallel file system. Internally, the data of the rank2file map is organized as a hash.

There is always a root file named `rank2file.scr`. Here are the contents of an example root rank2file map.

```
LEVEL
  1
RANKS
  4
RANK
  0
    OFFSET
      0
    FILE
      .scr/rank2file.0.0.scr
```

Note that there is no `VERSION` field. The version is implied from the summary file for the dataset. The `LEVEL` field lists the level at which the current rank2file map is located in the tree. The leaves of the tree are at level 0. The `RANKS` field specifies the number of ranks the current file (and its associated subtree) contains information for.

For levels that are above level 0, the RANK hash contains information about other rank2file map files to be read. Each entry in this hash is identified by a rank id, and then for each rank, a FILE and OFFSET are given. The rank id specifies which rank is responsible for reading content at the next level. The FILE field specifies the file name that is to be read, and the OFFSET field gives the starting byte offset within that file.

A process reading a file at the current level scatters the hash info to the designated "reader" ranks, and those processes read data for the next level. In this way, the task of reading the rank2file map is distributed among multiple processes in the job. The SCR library ensures that the maximum amount of data any process reads in any step is limited (currently 1MB).

File names at levels lower than the root have names of the form rank2file.<level>.<rank>.scr, where level is the level number within the tree and rank is the rank of the process that wrote the file.

Finally, level 0 contains the data that maps a rank to a list of files names. Here are the contents of an example rank2file map file at level 0.

```
RANK2FILE
  LEVEL
    0
  RANKS
    4
  RANK
    0
      FILE
        rank_0.ckpt
          SIZE
            524294
          CRC
            0x6697d4ef
    1
      FILE
        rank_1.ckpt
          SIZE
            524295
          CRC
            0x28eeb9e
    2
      FILE
        rank_2.ckpt
          SIZE
            524296
          CRC
            0xb6a62246
    3
      FILE
        rank_3.ckpt
          SIZE
            524297
          CRC
            0x213c897a
```

Again, the number of ranks that this file contains information for is recorded under the RANKS field.

There are entries for specific ranks under the RANK hash, which is indexed by rank id within scr_comm_world. For a given rank, each file that rank wrote as part of the dataset is indexed by file name under the FILE hash. The file name specifies the relative path to the file starting from the dataset directory. For each file, SCR records the size of the file in bytes under SIZE, and SCR may also record the CRC32 checksum value over the contents of the file under the CRC field.

On restart, the reader rank that reads this hash scatters the information to the owner rank, so that by the end of processing the tree, all processes know which files to read.

2.18 Filemap files

To efficiently support multiple processes per node, several files are used to record the files stored in cache. Each process reads and writes its own filemap file, named `filemap_#.scrinfo`, where # is the rank of the process in `scr_comm_local`. Additionally, the master rank on each node writes a file named `filemap.scrinfo`, which lists the file names for all of the filemap files. These files are all written to the SCR control directory.

For example, if there are 4 processes on a node, then the following files would exist in the SCR control directory.

```
filemap.scrinfo
filemap_0.scrinfo
filemap_1.scrinfo
filemap_2.scrinfo
filemap_3.scrinfo
```

The contents of each `filemap_#.scrinfo` file would look something like the example in Section *Example filemap hash*. The contents of `filemap.scrinfo` would be the following:

```
Filemap
    /<path_to_filemap_0>/filemap_0.scrinfo
    /<path_to_filemap_1>/filemap_1.scrinfo
    /<path_to_filemap_2>/filemap_2.scrinfo
    /<path_to_filemap_3>/filemap_3.scrinfo
```

With this setup, the master rank on each node writes `filemap.scrinfo` once during `SCR_Init()` and each process is then free to access its own filemap file independently of all other processes running on the node. The full path to each filemap file is specified to enable these files to be located in different directories. Currently all filemap files are written to the control directory.

During restart or during a scavenge, it is necessary for a newly started process to build a complete filemap of all files on a node. To do this, the process first reads `filemap.scrinfo` to get the names of all filemap files, and then it reads each filemap file using code like the following:

```
/* read master filemap to get the names of all filemap files */
struct scr_hash* maps = scr_hash_new();
scr_hash_read("...filemap.scrinfo", maps);

/* create an empty filemap and read in each filemap file */
struct scr_hash_elem* elem;
scr_filemap* map = scr_filemap_new();
for (elem = scr_hash_elem_first(maps, "Filemap");
     elem != NULL
     elem = scr_hash_elem_next(elem))
{
  char* file = scr_hash_elem_key(elem);
  scr_filemap_read(file, map)
}
```

2.19 Flush file

The flush file tracks where cached datasets are located. It is stored in the prefix directory. Internally, the data of the flush file is organized as a hash. Here are the contents of an example flush file.

```
DSET
  18
    LOCATION
      PFS
      CACHE
    DIR
      scr.dataset.18
  17
    LOCATION
      CACHE
    DIR
      scr.dataset.17
```

Each dataset is indexed by dataset id under the `DSET` hash. Then, under the `LOCATION` hash, different flags are set to indicate where that dataset is stored. The `PFS` flag indicates that a copy of this dataset is stored on the parallel file system, while the `CACHE` flag indicates that the dataset is stored in cache. The same dataset may be stored in multiple locations at the same time. The `DIR` field specifies the dataset directory name that SCR should use when copying the dataset to the prefix directory on the parallel file system. At the end of a run, the flush and scavenge logic in SCR uses information in this file to determine whether or not the most recent checkpoint has been copied to the parallel file system.

2.20 Halt file

The halt file tracks various conditions that are used to determine whether or not a run should continue to execute. The halt file is kept in the prefix directory. It is updated by the library during the run, and it is also updated externally through the `scr_halt` command. Internally, the data of the halt file is organized as a hash. Here are the contents of an example halt file.

```
CheckpointsLeft
  7
ExitAfter
  1298937600
ExitBefore
  1298944800
HaltSeconds
  1200
ExitReason
  SCR_FINALIZE_CALLED
```

The `CheckpointsLeft` field provides a counter on the number of checkpoints that should be completed before SCR stops the job. With each checkpoint, the library decrements this counter, and the run stops if it hits 0.

The `ExitAfter` field records a timestamp (seconds since UNIX epoch). At various times, SCR compares the current time to this timestamp, and it halts the run as soon as the current time exceeds this timestamp.

The `ExitBefore` field combined with the `HaltSeconds` field inform SCR that the run should be halted at specified number of seconds before a specified time. Again, SCR compares the current time to the time specified by subtracting the `HaltSeconds` value from the `ExitBefore` timestamp (seconds since UNIX epoch). If the current time is equal to or greater than this time, SCR halts the run.

Finally, the `ExitReason` field records a reason the job is or should be halted. If SCR ever detects that this field is set, it halts the job.

A user can add, modify, and remove halt conditions on a running job using the `scr_halt` command. Each time an application completes a dataset, SCR checks settings in the halt file. If any halt condition is satisfied, SCR flushes the most recent checkpoint, and then each process calls `exit()`. Control is not returned to the application.

2.21 Nodes file

The nodes file is kept in the prefix directory, and it tracks how many nodes were used by the previous run. Internally, the data of the nodes file is organized as a hash. Here are the contents of an example nodes file.

```
NODES
  4
```

In this example, the previous run which ran on this node used 4 nodes. The number of nodes is computed by finding the maximum size of `scr_comm_level` across all tasks in the MPI job. The master process on each node writes the nodes file to the control directory.

Before restarting a run, SCR uses information in this file to determine whether there are sufficient healthy nodes remaining in the allocation to run the job. If this file does not exist, SCR assumes the job needs every node in the allocation. Otherwise, it assumes the next run will use the same number of nodes as the previous run, which is recorded in this file.

2.22 Transfer file

When using the asynchronous flush, the library creates a dataset directory within the prefix directory, and then it relies on an external task to actually copy data from the cache to the parallel file system. The library communicates when and what files should be copied by updating the transfer file. A `scr_transfer` daemon process running in the background on each compute node periodically reads this file to check whether any files needs to be copied. If so, it copies data out in small bursts, sleeping a short time between bursts in order to throttle its CPU and bandwidth usage. The code for this daemon is in `scr_transfer.c`. Here is what the contents of a transfer file look like:

```
FILES
  /tmp/user1/scr.1001186/index.0/dataset.1/rank_0.ckpt
    DESTINATION
      /p/lscratchb/user1/simulation123/scr.dataset.1/rank_0.ckpt
    SIZE
      524294
    WRITTEN
      524294
  /tmp/user1/scr.1001186/index.0/dataset.1/rank_0.ckpt.scr
    DESTINATION
      /p/lscratchb/user1/simulation123/scr.dataset.1/rank_0.ckpt.scr
    SIZE
      124
    WRITTEN
      124
PERCENT
  0.000000
BW
  52428800.000000
COMMAND
```

(continues on next page)

(continued from previous page)

```
  RUN
STATE
  STOPPED
FLAG
  DONE
```

The library specifies the list of files to be flushed by absolute file name under the `FILES` hash. For each file, the library specifies the size of the file (in bytes) under `SIZE`, and it specifies the absolute path where the file should be written to under `DESTINATION`.

The library also specifies limits for the `scr_transfer` process. The `PERCENT` field specifies the percentage of CPU time the `scr_transfer` process should spend running. The daemon monitors how long it runs for when issuing a write burst, and then it sleeps for an appropriate amount of time before executing the next write burst so that it stays below this threshold. The `BW` field specifies the amount of bandwidth the daemon may consume (in bytes/sec) while copying data. The daemon process monitors how much data it has written along with the time taken to write that data, and it adjusts its sleep periods between write bursts to keep below its bandwidth limit.

Once the library has specified the list of files to be transferred and set any limits for the `scr_transfer` process, it sets the `COMMAND` field to `RUN`. The `scr_transfer` process does not start to copy data until this `RUN` command is issued. The library may also specify the `EXIT` command, which causes the `scr_transfer` process to exit.

The `scr_transfer` process records its current state in the `STATE` field, which may be one of: `STOPPED` (waiting to do something) and `RUNNING` (actively flushing). As the `scr_transfer` process copies each file out, it records the number of bytes it has written (and fsync'd) under the `WRITTEN` field. When all files in the list have been copied, `scr_transfer` sets the `DONE` flag under the `FLAG` field. The library periodically looks for this flag, and once set, the library completes the flush by writing the summary file in the dataset directory and updating the index file in the prefix directory.

2.23 Perl modules

2.23.1 scripts/common/scr_hostlist.pm

Manipulates lists of hostnames. The elements in a set of hostnames are expected to have a common alpha prefix (machine name) followed by a number (node number). A hostlist can be specified in one of two forms:

compressed	"atlas[3,5-7,9-11]"	Perl string scalar
uncompressed	("atlas3","atals5","atlas6","atlas7","atlas9","atlas10","atlas11")	Perl list of string scalars

All functions in this module are global; no instance must be created.

Given a compressed hostlist, construct the corresponding uncompressed hostlist (preserves order and duplicates).

```
my @hostlist = scr_hostlist::expand($hostlist);
```

Given an uncompressed hostlist, construct a compressed hostlist (preserves duplicate hostnames, but sorts list by node number).

```
my $hostlist = scr_hostlist::compress(@hostlist);
```

Given references to two uncompressed hostlists, subtract second list from first and return remainder as an uncompressed hostlist.

```
my @hostlist = scr_hostlist::diff(\@hostlist1, \@hostlist2);
```

Given references to two uncompressed hostlists, return the intersection of the two as an uncompressed hostlist.

```
my @hostlist = scr_hostlist::intersect(\@hostlist1, \@hostlist2);
```

2.23.2 scripts/common/scr_param.pm.in

Reads and returns SCR configuration parameters, returning the first set value found by searching in the following order:

1. Environment variable,
2. User configuration file,
3. System configuration file,
4. Default (build-time constant).

When an instance is created, it attempts to read the user configuration file from `SCR_CONF_FILE` if that variable is set. Otherwise, it attempts to read the user configuration file from `<prefix>/.scrconf`, where `<prefix>` is the prefix directory specified in `SCR_PREFIX` or the current working directory if `SCR_PREFIX` is not set.

Some parameters cannot be set by a user, and for these parameters any settings in environment variables or the user configuration file are ignored.

The majority of parameters return scalar values, but some return an associated hash.

Allocate a new `scr_param` object.

```
my $param = new scr_param();
```

Given the name of an SCR parameter, return its scalar value.

```
my $val = $param->get($name);
```

Given the name of an SCR parameter, return a reference to its hash.

```
my $hashref = $param->get_hash($name);
```

To disable a user from setting a parameter, add it to the `no_user` hash within the module implementation. Parameters listed in this hash will not be affected by environment variables or user configuration file settings.

Compile-time constants should be listed in the `compile` hash.

2.24 Utilities

2.24.1 scripts/common/scr_glob_hosts.in

Uses `scr_hostlist.pm` to manipulate hostlists. Only accepts compressed hostlists for input.

Given a compressed hostlist, return number of hosts.

```
scr_glob_hosts --count "atlas[3,5-7,9-11]"
```

The example above returns "7", as there are seven hosts specified in the list.

Given a compressed hostlist, return the nth host.

```
scr_glob_hosts --nth 3 "atlas[3,5-7,9-11]"
```

The example above returns "atlas6", which is the third host.

Given two compressed hostlists, subtract one from the other and return remainder.

```
scr_glob_hosts --minus "atlas[3,5-7,9-11]":"atlas[5,7,20]"
```

The above example returns "atlas[3,6,9-11]", which has removed "atlas5" and "atlas7" from the first list.

Given two compressed hostlists, return intersection of the two.

```
scr_glob_hosts --intersection "atlas[3,5-7,9-11]":"atlas[5,7,20]"
```

The above example returns "atlas[5,7]", which is the list of common hosts between the two lists.

2.24.2 src/scr_flush_file.c

Utility to access info in SCR flush file. One must specify the prefix directory from which to read the flush file.

Read the flush file and return the latest checkpoint id.

```
scr_flush_file --latest /prefix/dir
```

The above command prints the checkpoint id of the most recent checkpoint in the flush file. It exits with a return code of 0 if it found a checkpoint id, and it exits with a return code of 1 otherwise.

Determine whether a specified checkpoint id needs to be flushed.

```
scr_flush_file --needflush 6 /prefix/dir
```

The command above checks whether the SCR_FLUSH_KEY_LOCATION_PFS key is set for the specified checkpoint id. If so, the command exits with 0, otherwise is exits with 1.

List the location(s) containing a copy of the dataset.

```
scr_flush_file --location 6 /prefix/dir
```

This command lists PFS for the parallel file system, and it lists CACHE for datasets stored in cache.

List the subdirectory where a dataset should be flushed to.

```
scr_flush_file --subdir 6 /prefix/dir
```

2.24.3 scripts/common/scr_list_dir.in

Returns full path to control or cache directory. Uses scr_param.pm. This command should be executed in an environment where SCR_CONF_FILE is set to the same value as the running job.

1. Uses scr_param.pm to read SCR_CNTL_BASE to get base control directory.
2. Uses scr_param.pm to read CACHE hash from config file to get info on cache directories.
3. Invokes "scr_env -user" to get the username if not specified on command line.

4. Invokes "scr_env -jobid" to get the jobid if not specified on command line.
5. Combines base, user, and jobid to build and output full path to control or cache directory.

2.24.4 scripts/TLCC/scr_list_down_nodes.in

Runs a series of tests over all specified nodes and builds list of nodes which fail one or more tests. Uses scr_hostlist.pm to manipulate hostlists. Uses scr_param.pm to read various parameters.

1. Invokes "scr_env -nodes" to get the current nodeset, if not specified on command line.
2. Invokes "scr_env -down" to ask resource manager whether any nodes are known to be down.
3. Invokes ping to each node thought to be up.
4. Uses scr_param.pm to read SCR_EXCLUDE_NODES, user may explicitly exclude nodes this way.
5. Adds any nodes explicitly listed on command line as being down.
6. Invokes scr_list_dir to get list of base directories for control directory.
7. Uses scr_param.pm to read CNTLDIR hash from config file to get expected capacity corresponding to each base directory.
8. Invokes scr_list_dir to get list of base directories for cache directory.
9. Uses scr_param.pm to read CACHEDIR hash from config file to get expected capacity corresponding to each base directory.
10. Invokes pdsh to run scr_check_node on each node that hasn't yet failed a test.
11. Optionally print list of down nodes to stdout.
12. Optionally log each down node with reason via scr_log_event if logging is enabled.
13. Exit with appropriate code to indicate whether any nodes are down.

2.24.5 scripts/common/scr_check_node.in

Runs on compute node to execute a series of checks to verify that node is still functioning.

1. Reads list of control directories and sizes from –cntl option.
2. Reads list of cache directories and sizes from –cache option.
3. Invokes ls -lt to check basic access for each directory.
4. If size is specified, invoke df to verify that each directory has sufficient space.
5. Invokes touch and rm -rf to create and delete a test file in each directory.

2.24.6 scripts/common/scr_prefix.in

Prints SCR prefix directory.

1. Reads $SCR_PREFIX if set.
2. Invokes pwd to get current working directory otherwise.

2.25 Launch a run

2.25.1 scripts/TLCC/scr_run.in

Prepares a resource allocation for SCR, launches a run, re-launches on failures, and scavenges and rebuilds files for most recent checkpoint if needed. Updates SCR index file in prefix directory to account for last checkpoint.

1. Interprets `$SCR_ENABLE`, calls `srun` and bails if set to 0.
2. Interprets `$SCR_DEBUG`, enables verbosity if set > 0.
3. Invokes `scr_test_runtime` to check that runtime dependencies are available.
4. Invokes "`scr_env -jobid`" to get jobid of current job.
5. Interprets `$SCR_NODELIST` to determine set of nodes job is using, sets and exports `$SCR_NODELIST` to value returned by "`scr_env -nodes`" if not set.
6. Invokes `$scr_prefix` to get prefix directory on parallel file system.
7. Interprets `$SCR_WATCHDOG`.
8. Invokes `scr_glob_hosts` to check that this command is running on a node in the nodeset, bails with error if not.
9. Invokes `scr_list_dir` to get control directory.
10. Issues a NOP `srun` command on all nodes to force each node to run SLURM prologue to delete old files from cache.
11. Invokes `scr_prerun` to prepare nodes for SCR run.
12. If `$SCR_FLUSH_ASYNC == 1`, invokes `scr_glob_hosts` to get count of number of nodes. and invokes `srun` to launch an `scr_transfer` process on each node.

ENTER LOOP

1. Invokes `scr_list_down_nodes` to determine list of bad nodes. If any node has been previously marked down, force it to continue to be marked down. We do this to avoid re-running on "bad" nodes, the logic being that if a node was identified as being bad in this resource allocation once already, there is a good chance that it is still bad (even if it currently seems to be healthy), so avoid it.
2. Invokes `scr_list_down_nodes` to print reason for down nodes, if any.
3. Count the number of nodes that the application needs. Invokes `scr_glob_hosts` to count number of nodes in `$SCR_NODELIST`, which lists all nodes in allocation. Interprets `$SCR_MIN_NODES` to use that value of set, otherwise invokes `scr_env -runnodes` to get number of nodes used in last run.
4. Invokes `scr_glob_host` to count number of nodes left in the allocation.
5. If number of nodes left is smaller than number needed, break loop.
6. Invokes `scr_glob_host` to ensure node running `scr_srun` script is not listed as a down node, if it is, break loop.
7. Build list of nodes to be excluded from run.
8. Optionally log start of run.
9. Invokes `srun` including node where the `scr_srun` command is running and excluding down nodes.
10. If watchdog is enabled, record pid of srun, invokes `sleep 10` so job shows up in squeue, invokes `scr_get_jobstep_id` to get SLURM jobstep id from pid, invokes `scr_watchdog` and records pid of watchdog process.

11. Invokes `scr_list_down_nodes` to get list of down nodes.
12. Optionally log end of run (and down nodes and reason those nodes are down).
13. If number of attempted runs is >= than number of allowed retries, break loop.
14. Invokes `scr_retries_halt` and breaks loop if halt condition is detected.
15. Invokes "`sleep 60`" to give nodes in allocation a chance to cleanup.
16. Invokes `scr_retries_halt` and breaks loop if halt condition is detected. We do this a second time in case a command to halt came in while we were sleeping.
17. Loop back.

EXIT LOOP

1. If `$SCR_FLUSH_ASYNC == 1`, invokes "`scr_halt -immediate`" to kill `scr_transfer` processes on each node.
2. Invokes `scr_postrun` to scavenge most recent checkpoint.
3. Invokes `kill` to kill watchdog process if it is running.

2.25.2 scripts/common/scr_test_runtime.in

Checks that various runtime dependencies are available.

1. Checks for `pdsh` command,
2. Checks for `dshbak` command,
3. Checks for `Date::Manip` perl module.

2.25.3 scripts/common/scr_prerun.in

Executes commands to prepare an allocation for SCR.

1. Interprets `$SCR_ENABLE`, calls `srun` and bails if set to 0.
2. Interprets `$SCR_DEBUG`, enables verbosity if set > 0.
3. Invokes `scr_test_runtime` to check for necessary run time dependencies.
4. Invokes `mkdir` to create `.scr` subdirectory in prefix directory.
5. Invokes `rm -f` to remove flush and nodes files from prefix directory.
6. Returns 0 if allocation is ready, 1 otherwise.

2.25.4 src/scr_retries_halt.c

Reads halt file and returns exit code depending on whether the run should be halted or not.

2.26 SCR_Init

During `SCR_Init()`, the library allocates and initializes data structures. It inspects the cache, and it distributes and rebuilds files for cached datasets. Otherwise, it attempts to fetch the most recent checkpoint from the parallel file system. This function is implemented in `scr.c`.

1. Interprets `$SCR_ENABLE`, if not enabled, bail out with error.
2. Calls `DTCMP_Init` if datatype compare library is available.
3. Create `scr_comm_world` by duplicating `MPI_COMM_WORLD` and set `scr_my_rank_world` and `scr_ranks_world`.
4. Call `scr_env_nodename` to get hostname, store in `scr_my_hostname`.
5. Call `getpagesize` to get memory page size, store in `scr_page_size`.
6. Initialize parameters – rank 0 reads any config files and broadcasts info to other ranks.
7. Check whether we are still enabled (a config file may disable us), and bail out with error if not.
8. Check that `scr_username` and `scr_jobid` are defined, which are used for logging purposes.
9. Call `scr_groupdescs_create` to create group descriptors (Section *0.1.1*).
10. Call `scr_storedescs_create` to create store descriptors (Section *0.1.2*).
11. Call `scr_reddescs_create` to create redundancy descriptors (Section *0.1.3*).
12. Check that we have a valid redundancy descriptor that we can use for each checkpoint.
13. Create `scr_comm_node` and `scr_comm_node_across`.
14. Log the start of this run, if logging is enabled.
15. Verify that `scr_prefix` is defined.
16. Define `scr_prefix_scr` to be `.scr` directory within prefix directory.
17. Create `.scr` directory in prefix directory.
18. Define control directory and save in `scr_cntl_prefix`.
19. Create the control directory.
20. Create each of the cache directories.

BARRIER

1. Define file names for halt, flush, nodes, transfer, and filemap files.
2. Delete any existing transfer file.
3. Create nodes file, write total number of nodes in the job (max of size of `scr_comm_level`).
4. Allocate a hash to hold halt status, and initialize halt seconds if needed.

BARRIER

1. Stop any ongoing asynchronous flush.
2. Check whether we need to halt and exit this run.
3. Enable `scr_flush_on_restart` and disable `scr_fetch` if `scr_global_restart` is set.
4. Create empty hash for filemap.
5. Master process for each store reads and distributes info from filemaps (Section *0.1.4*).
6. Call `scr_cache_rebuild` to rebuild datasets in cache (Section *0.1.5*).
7. If rebuild was successful, call `scr_flush_file_rebuild` to update flush file.
8. If rebuild successful, check whether dataset should be flushed.
9. If we don't have a checkpoint (rebuild failed), call `scr_cache_purge` to clear cache (delete all files).

10. If fetch is enabled, call `scr_fetch_sync` to read checkpoint from parallel file system (Section *0.1.12*).
11. If the fetch failed, clear the cache again.

BARRIER

1. Log end of initialization.
2. Start timer to record length of compute phase and log start of compute phase.

2.26.1 scr_groupdescs_create

The group descriptors are kept in the `scr_groupdescs` array (Section Group descriptors]). This function is implemented in `scr_groupdesc.c`.

1. Read `GROUPS` key from `scr_groupdesc_hash` which is set while processing parameters in `SCR_Init`.
2. Count number of groups and add two.
3. Allocate space for `scr_groupdesc` array.
4. Initialize each group descriptor.
5. Create descriptor for all tasks on the same node called `NODE`.
6. Create descriptor for all tasks in job called `WORLD`.
7. Create each group descriptor specified in `scr_groupdesc_hash` by calling `scr_groupdesc_create_by_str`. Rank 0 broadcasts the group name to determine the order.

The `scr_groupdesc_create_by_str` function creates groups of processes by splitting `scr_comm_world` into subcommunicators containing all procs that specify the same string. The real work is delegated to `scr_rank_str`, which is implemented in `scr_split.c`. It executes a bitonic sort on string names, and it returns the number of distinct groups across all procs and the group id to which the calling process belongs. This id is then used in a call to `MPI_Comm_split`.

2.26.2 scr_storedescs_create

The store descriptors are kept in the `scr_storedescs` array (Section *Store descriptors*). This function is implemented in `scr_storedesc.c`.

1. Read `STORE` key from `scr_storedesc_hash` which is set while processing parameters in `SCR_Init`.
2. Count number of store descriptors.
3. Allocate space for `scr_storedescs` array.
4. Sort store descriptors to ensure they are in the same order on all procs.
5. Create each store descriptor specified in `scr_storedesc_hash` by calling `scr_storedesc_create_from_hash`.
6. Create store descriptor for control directory and save it in `scr_storedesc_cntl`.

The `scr_storedesc_create` function sets all fields in the descriptor using default values or values defined in the hash. A key field is a communicator consisting of the group of processes that share the associated storage device. This communicator is used to coordinate processes when accessing the device. It is created by duplicating a communicator from a group descriptor.

2.26.3 scr_reddescs_create

The redundancy descriptors are kept in the `scr_reddescs` array (Section *Redundancy descriptors*). This function is implemented in `scr_reddesc.c`.

1. Read `CKPT` key from `scr_reddesc_hash` which is set while processing parameters in `SCR_Init`.
2. Count number of redundancy descriptors.
3. Allocate space for `scr_reddescs` array.
4. Sort redundancy descriptors to ensure they are in the same order on all procs.
5. Create each redundancy descriptor specified in `scr_reddesc_hash` by calling `scr_reddesc_create_from_hash`.

The `scr_reddesc_create_from_hash` function sets all fields in the descriptor from default values or values defined in the hash. Two key fields consist of an index to the store descriptor providing details on the class of storage to use and a communicator on which to compute redundancy data. To build the communicator, a new communicator is created by splitting `scr_comm_world` into subcommunicators consisting of processes from different failure groups.

2.26.4 scr_scatter_filemaps

During a restart, the master process for each control directory reads in all filemap data and distributes this data to the other processes sharing the control directory, if any. After this distribution phase, a process is responsible for each file it has filemap data for, and each file in cache is the responsibility of some process. We use this approach to handle cases where the number of tasks accessing the control directory in the current run is different from the number of tasks in the prior run. This function is implemented in `scr_cache_rebuild.c`.

1. Master reads master filemap file.
2. Master creates empty filemap and reads each filemap file listed in the master filemap. Deletes each filemap file as it's read.
3. Gather list of global rank ids sharing the store to master process.
4. If the filemap has data for a rank, master prepares hash to send corresponding data to that rank.
5. Master evenly distributes the remainder of the filemap data to all processes.
6. Distribute filemap data via `scr_hash_exchange()`.
7. Master writes new master filemap file.
8. Each process writes new filemap file.

2.26.5 scr_cache_rebuild

This section describes the logic to distribute and rebuild files in cache. SCR attempts to rebuild all cached datasets. This functionality is implemented in `scr_cache_rebuild.c`.

1. Start timer.
2. Delete any files from cache known to be incomplete.
3. Get list of dataset ids currently in cache.

LOOP

1. Identify dataset with lowest id across all procs yet to be rebuilt.
2. If there is no dataset id specified on any process, break loop.

3. Otherwise, log which dataset we are attempting to rebuild.
4. Distribute hash for this dataset and store in map object (Section *0.1.6*).
5. If we fail to distribute the hash to all processes, delete this dataset from cache and loop.
6. Distribute redundancy descriptors for this dataset and store in temporary redundancy descriptor object (Section *0.1.7*). This informs each process about the cache device and the redundancy scheme to use for this dataset.
7. If we fail to distribute the redundancy descriptors to all processes, delete this dataset from cache and loop.
8. Create dataset directory in cache according to redundancy descriptor.
9. Distribute files to the ranks that wrote them (Section *0.1.8*). The owner ranks may now be on different nodes.
10. Rebuild any missing files for this dataset using redundancy scheme specified in redundancy descriptor (Section *0.1.9*).
11. If the rebuild fails, delete this dataset from cache and loop.
12. Otherwise, the rebuild succeeded. Update `scr_dataset_id` and `scr_checkpoint_id` if the id for the current dataset is higher, so that we continue counting up from this number when assigning ids to later datasets.
13. Unset FLUSHING flag in flush file.
14. Free the temporary redundancy descriptor.

EXIT LOOP

1. Stop timer and log whether we were able to rebuild any dataset from cache.

2.26.6 scr_distribute_datasets

Given a filemap and dataset id, distribute dataset hash and store in filemap.

1. Create empty send hash for transferring dataset hashes.
2. Get list of ranks that we have files for as part of the specified dataset.
3. For each rank, lookup dataset hash from filemap and add to send hash.
4. Delete list of ranks.
5. Check that no rank identified an invalid rank. If the restarted run uses a smaller number of processes than the previous run, we may (but are not guaranteed to) discover this condition here.
6. Identify smallest rank that has a copy of the dataset hash.
7. Return with failure if no such rank exists.
8. Otherwise, broadcast hash from this rank.
9. Store dataset hash in filemap and write filemap to disk.
10. Delete send hash.

2.26.7 scr_distribute_reddescs

Given a filemap and dataset id, distribute redundancy descriptor that was applied to the dataset and store in filemap. This creates the same group and redundancy scheme that was applied to the dataset, even if the user may have configured new schemes for the current run.

1. Create empty send hash for transferring redundancy descriptor hashes.
2. Get list of ranks that we have files for as part of the specified dataset.

3. For each rank, lookup redundancy descriptor hash from filemap and add to send hash.
4. Delete list of ranks.
5. Check that no rank identified an invalid rank. If the restarted run uses a smaller number of processes than the previous run, we may (but are not guaranteed to) discover this condition here.
6. Execute sparse data exchange with `scr_hash_exchange`.
7. Check that each rank received its descriptor, return with failure if not.
8. Store redundancy descriptor hash in filemap and write filemap to disk.
9. Create redundancy descriptor by calling `scr_reddesc_create_from_filemap`.
10. Delete send and receive hashes from exchange.

2.26.8 scr_distribute_files

This section describes the algorithm used to distribute files for a specified dataset. SCR transfers files from their current location to the storage device accessible from the node where the owner rank is now running. The algorithm operates over a number of rounds. In each round, a process may send files to at most one other process. A process may only send files if it has all of the files written by the owner process. The caller specifies a filemap, a redundancy descriptor, and a dataset id as input. This implementation is in `scr_cache_rebuild.c`.

1. Delete all bad (incomplete or inaccessible) files from the filemap.
2. Get list of ranks that we have files for as part of the specified dataset.
3. From this list, set a start index to the position corresponding to the first rank that is equal to or greater than our own rank (looping back to rank 0 if we pass the last rank). We stagger the start index across processes in this way to help distribute load later.
4. Check that no rank identified an invalid rank while scanning for its start index. If the restarted run uses a smaller number of processes than the previous run, we may (but are not guaranteed to) discover this condition here.
5. Allocate arrays to record which rank we can send files to in each round.
6. Check that we have all files for each rank, and record the round in which we can send them. The round we pick here is affected by the start index computed earlier.
7. Issue sparse global exchange via `scr_hash_exchange` to inform each process in which round we can send it its files, and receive similar messages from other processes.
8. Search for minimum round in which we can retrieve our own files, and remember corresponding round and source rank. If we can fetch files from our self, we'll always select this option as it will be the minimum round.
9. Free the list of ranks we have files for.
10. Determine whether all processes can obtain their files, and bail with error if not.
11. Determine the maximum round any process needs to get its files.
12. Identify which rank we'll get our files from and issue sparse global exchange to distribute this info.
13. Determine which ranks want to receive files from us, if any, and record the round they want to receive their files in.
14. Get the directory name for this dataset.
15. Loop through the maximum number of rounds and exchange files.

LOOP ROUNDS

1. Check whether we can send files to a rank in this round, and if so, record destination and number of files.

2. Check whether we need to receive our files in this round, and if so, record source rank.
3. If we need to send files to our self, just move (rename) each file, update the filemap, and loop to the next round.
4. Otherwise, if we have files for this round but the the owner rank does not need them, delete them.
5. If we do not need to send or receive any files this round, loop to next round.
6. Otherwise, exchange number of files we'll be sending and/or receiving, and record expected number that we'll receive in our filemap.
7. If we're sending files, get a list of files for the destination.
8. Enter exchange loop.

LOOP EXCHANGE

1. Get next file name from our list of files to send, if any remaining.
2. Swap file names with partners.
3. If we'll receive a file in this iteration, add the file name to the filemap and write out our filemap.
4. Transfer file via `scr_swap_files()`. This call overwrites the outgoing file (if any) with the incoming file (if any), so there's no need to delete the outgoing file. If there is no incoming file, it deletes the outgoing file (if any). We use this approach to conserve storage space, since we assume the cache is small. We also transfer file metadata with this function.
5. If we sent a file, remove that file from our filemap and write out the filemap.
6. Decrement the number of files we have to send / receive by one. When both counts hit zero, break exchange loop.
7. Write updated filemap to disk.

EXIT LOOP EXCHANGE

1. Free list of files that we sent in this round.

EXIT LOOP ROUNDS

1. If we have more ranks than there were rounds, delete files for all remaining ranks.
2. Write out filemap file.
3. Delete bad files (incomplete or inaccessible) from the filemap.

2.26.9 scr_reddesc_recover

This function attempts to rebuild any missing files for a dataset. It returns `SCR_SUCCESS` on all processes if successful; it returns `!SCR_SUCCESS` on all processes otherwise. The caller specifies a filemap, a redundancy descriptor, and a dataset id as input. This function is implemented in in `scr_reddesc_recover.c`.

1. Attempt to rebuild files according to the redundancy scheme specified in the redundancy descriptor. Currently, only `XOR` can actually rebuild files (Section *0.1.10*).
2. If the rebuild failed, return with an error.
3. Otherwise, check that all processes have all of their files for the dataset.
4. If not, return with an error.
5. If so, reapply the redundancy scheme, if needed. No need to do this with `XOR`, since it does this step as part of the rebuild.

2.26.10 scr_reddesc_recover_xor_attempt

Before we attempt to rebuild files using the XOR redundancy scheme, we first check whether it is possible. If we detect that two or more processes from the same XOR set are missing files, we cannot recover all files and there is no point to rebuild any of them. We execute this check in `scr_reddesc_recover.c`. The caller specifies a filemap, a redundancy descriptor, and a dataset id as input.

1. Check whether we have our dataset files, and check whether we have our XOR file. If we're missing any of these files, assume that we're missing them all.
2. Count the number of processes in our XOR set that need their files. We can recover all files from a set so long as no more than a single member needs its files.
3. Check whether we can recover files for all sets, if not bail with an error.
4. If the current process is in a set which needs to be rebuilt, identify which rank needs its files and call `scr_reddesc_recover_xor()` to rebuild files (Section *0.1.11*).
5. Check that the rebuild succeeded on all tasks, return error if not, otherwise return success.

2.26.11 scr_reddesc_recover_xor

We invoke this routine within each XOR set that is missing files. The caller specifies a filemap, a redundancy descriptor, and a dataset id as input, as well as, the rank of the process in the XOR set that is missing its files. We refer to the process that needs to rebuild its files as the *root*. This function is implemented in `scr_reddesc_recover.c`

ALL

1. Get pointer to XOR state structure from `copy_state` field of redundancy descriptor.
2. Allocate empty hash to hold the header of our XOR file.

NON-ROOT

1. Get name of our XOR file.
2. Open XOR file for reading.
3. Read header from file.
4. From header, get hash of files we wrote.
5. From this file hash, get the number of files we wrote.
6. Allocate arrays to hold file descriptor, file name, and file size for each of our files.
7. Get path of dataset directory from XOR file name.
8. Open each of our files for reading and store file descriptor, file name, and file size of each file in our arrays.
9. If the failed rank is to our left, send it our header. Our header stores a copy of the file hash for the rank to our left under the PARTNER key.
10. If the failed rank is to our right, send it our file hash. When the failed rank rebuilds its XOR file, it needs to record our file hash in its header under the PARTNER key.

ROOT

1. Receive XOR header from rank to our right.
2. Rename PARTNER key in this header to CURRENT. The rank to our right stored a copy of our file hash under PARTNER.
3. Receive file hash from rank to our left, and store it under PARTNER in our header.

4. Get our file hash from `CURRENT` key in the header.
5. From our file hash, get the number of files we wrote during the dataset.
6. Allocate arrays to hold file descriptor, file name, and file size for each of our files.
7. Build the file name for our `XOR` file, and add `XOR` file to the filemap.
8. For each of our files, get meta data from file hash, then get file name and file size from meta data. Add file name to filemap, and record file name and file size in arrays.
9. Record the number of files we expect to have in the filemap, including the `XOR` file.
10. Write out filemap.
11. Open `XOR` file for writing.
12. Open each of our dataset files for writing, and record file descriptors in our file descriptor array.
13. Write out `XOR` header to `XOR` file.

ALL

1. Read `XOR` chunk size from header.
2. Allocate buffers to send and receive data during reduction.
3. Execute pipelined `XOR` reduction to root to reconstruct missing data as illustrated in Figure *Pipelined XOR reduction to root*. For a full description of the redundancy scheme, see Section *XOR algorithm*.
4. Close our `XOR` file.
5. Close each of our dataset files.

ROOT

1. For each of our dataset files and our `XOR` file, update filemap.
2. Write filemap to disk.
3. Also compute and record CRC32 checksum for each file if `SCR_CRC_ON_COPY` is set.

ALL

1. Free data buffers.
2. Free arrays for file descriptors, file names, and file sizes.
3. Free `XOR` header hash.

2.26.12 scr_fetch_sync

This section describes the loop used to fetch a checkpoint from the parallel file system. SCR starts with the most recent checkpoint on the parallel file system as specified in the index file. If SCR fails to fetch this checkpoint, it then works backwards and attempts to fetch the next most recent checkpoint until it either succeeds or runs out of checkpoints. It acquires the list of available checkpoints from the index file. This functionality is implemented within `scr_fetch.c`.

1. Start timer.
2. Rank 0 reads index file from prefix directory, bail if failed to read file.

LOOP

1. Rank 0 selects a target directory name. Start with directory marked as current if set, and otherwise use most recent checkpoint specified in index file. For successive iterations, attempt the checkpoint that is the next most recent.

2. Rank 0 records fetch attempt in index file.
3. Rank 0 builds full path to dataset.
4. Broadcast dataset path from rank 0.
5. Attempt to fetch checkpoint from selected directory.
6. If fetch fails, rank 0 deletes "current" designation from dataset and marks dataset as "failed" in index file.
7. If fetch succeeds, rank 0 updates "current" designation to point to this dataset in index file, break loop.

EXIT LOOP

1. Delete index hash.
2. Stop timer and print statistics.

2.27 SCR_Need_checkpoint

Determines whether a checkpoint should be taken. This function is implemented in `scr.c`.

1. If not enabled, bail out with error.
2. If not initialized, bail out with error.
3. Increment the `scr_need_checkpoint_id` counter. We use this counter so the user can specify that the application should checkpoint after every so many calls to `SCR_Need_checkpoint`.
4. Check whether we need to halt. If so, then set need checkpoint flag to true.
5. Rank 0 checks various properties to make a decision: user has called `SCR_Need_checkpoint` an appropriate number of times, or the max time between consecutive checkpoints has expired, or the ratio of the total checkpoint time to the total compute time is below a threshold.
6. Rank 0 broadcasts the decision to all other tasks.

2.28 SCR_Start_checkpoint

Prepares the cache for a new checkpoint. This function is implemented in `scr.c`.

1. If not enabled, bail out with error.
2. If not initialized, bail out with error.
3. If this is being called from within a Start/Complete pair, bail out with error.
4. Issue a barrier here so that processes don't delete checkpoint files from the cache before we're sure that all processes will actually make it this far.

BARRIER

1. Stop timer of compute phase, and log this compute section.
2. Increment `scr_dataset_id` and `scr_checkpoint_id`.
3. Get redundancy descriptor for this checkpoint id.
4. Start timer for checkpoint phase, and log start of checkpoint.
5. Get a list of all datasets in cache.
6. Get store descriptor associated with redundancy descriptor.

7. Determine how many checkpoints are currently in the cache directory specified by the store descriptor.
8. Delete oldest datasets from this directory until we have sufficient room for this new checkpoint. When selecting checkpoints to delete, skip checkpoints that are being flushed. If the only option is a checkpoint that is being flushed, wait for it to complete then delete it.
9. Free the list of checkpoints.
10. Rank 0 fills in the dataset descriptor hash and broadcasts it.
11. Store dataset hash in filemap.
12. Add flush descriptor entries to filemap for this dataset.
13. Store redundancy descriptor in filemap.
14. Write filemap to disk.
15. Create dataset directory in cache.

2.29 SCR_Route_file

Given a name of a file, return the string the caller should use to access this file. This function is implemented in `scr.c`.

1. If not enabled, bail out with error.
2. If not initialized, bail out with error.
3. Lookup redundancy descriptor for current checkpoint id.
4. Direct path to dataset directory in cache according to redundancy descriptor.
5. If called from within a Start/Complete pair, add file name to filemap. Record original file name as specified by caller, the absolute path to the file and the number of ranks in the job in the filemap. Update filemap on disk.
6. Otherwise, we assume we are in a restart, so check whether we can read the file, and return error if not. The goal in this case is to provide a mechanism for a process to determine whether it can read its checkpoint file from cache during a restart.
7. Return success.

2.30 SCR_Complete_checkpoint

Applies redundancy scheme to checkpoint files, may flush checkpoint to parallel file system, and may exit run if the run should be halted. This function is implemented in `scr.c`.

1. If not enabled, bail out with error.
2. If not initialized, bail out with error.
3. If not called from within Start/Complete pair, bail out with error.
4. Record file size and valid flag for each file written during checkpoint.
5. Write out meta data for each file registered in filemap for this dataset id.
6. Compute total data size across all procs and determine whether all procs specified a valid write.
7. Update filemap and write to disk.

8. Verify that flush is valid by checking that all files belong to same subdirectory and compute container offsets if used.
9. Apply redundancy scheme specified in redundancy descriptor (Section *0.5.1* or Section *0.5.2*).
10. Stop timer measuring length of checkpoint, and log cost of checkpoint.
11. If checkpoint was successful, update our flush file, check whether we need to halt, and check whether we need to flush.
12. If checkpoint was not successful, delete it from cache.
13. Check whether any ongoing asynchronous flush has completed.

BARRIER

1. Start timer for start of compute phase, and log start of compute phase.

2.30.1 scr_reddesc_apply_partner

Algorithm to compute `PARTNER` redundancy scheme. Caller provides a filemap, a redundancy descriptor, and a dataset id. This function is implemented in `scr_reddesc_apply.c`.

1. Get pointer to partner state structure from `copy_state` field in redundancy descriptor.
2. Read list of files for this rank for the specified checkpoint.
3. Inform our right-hand partner how many files we'll send.
4. Record number of files we expect to receive from our left-hand partner in our filemap.
5. Remember the node name where our left-hand partner is running (used during scavenge).
6. Record the redundancy descriptor hash for our left-hand partner. Each process needs to be able to recover its own redundancy descriptor hash after a failure, so we make a copy in our partner's filemap.
7. Write filemap to disk.
8. Get checkpoint directory we'll copy partner's files to.
9. While we have a file to send or receive, loop.

LOOP

1. If we have a file to send, get the file name.
2. Exchange file names with left-hand and right-hand partners.
3. If our left-hand partner will be sending us a file, add the file name to our filemap, and write out our filemap.
4. Exchange files by calling `scr_swap_files()`, and update filemap with meta data for file.

EXIT LOOP

1. Write filemap to disk.
2. Free the list of file names.

2.30.2 scr_reddesc_apply_xor

Algorithm to compute `XOR` redundancy scheme. Caller provides a filemap, a redundancy descriptor, and a dataset id. The `XOR` set is the group of processes defined by the communicator specified in the redundancy descriptor. This function is implemented in `scr_reddesc_apply.c`.

1. Get pointer to `XOR` state structure from `copy_state` field in redundancy descriptor.

2. Allocate a buffers to send and receive data.
3. Count the number of files this process wrote during the specified dataset id. Allocate space to record a file descriptor, the file name, and the size of each file.
4. Record the redundancy descriptor hash for our left-hand partner in our filemap. Each process needs to be able to recover its own redundancy descriptor hash after a failure, so each process sends a copy to his right-hand partner.
5. Allocate a hash to hold the header of our `XOR` redundancy file.
6. Record the global ranks of the MPI tasks in our `XOR` set.
7. Record the dataset id in our header.
8. Open each of our files, get the size of each file, and read the meta data for each file.
9. Create a hash and record our rank, the number of files we have, and the meta data for each file.
10. Send this hash to our right-hand partner, and receive equivalent hash from left-hand partner.
11. Record our hash along with the hash from our left-hand partner in our `XOR` header hash. This way, the meta data for each file is recorded in the headers of two different `XOR` files.
12. Determine chunk size for RAID algorithm (Section *XOR algorithm*) and record this size in the `XOR` header.
13. Determine full path name for `XOR` file.
14. Record `XOR` file name in our filemap and update the filemap on disk.
15. Open the `XOR` file for writing.
16. Write header to file and delete header hash.
17. Execute RAID algorithm and write data to `XOR` file (Section *XOR algorithm*).
18. Close and fsync our `XOR` file and close each of our dataset files.
19. Free off scratch space memory and MPI buffers.
20. Write out meta data file for `XOR` file.
21. If `SCR_CRC_ON_COPY` is specified, compute CRC32 checksum of `XOR` file.

2.31 SCR_Finalize

Shuts down the SCR library, flushes most recent checkpoint to the parallel file system, and frees data structures. This function is implemented in `scr.c`.

1. If not enabled, bail out with error.
2. If not initialized, bail out with error.
3. Stop timer measuring length of compute phase.
4. Add reason for exiting to halt file. We assume the user really wants to stop once the application calls `SCR_Finalize`. We add a reason to the halt file so we know not to start another run after we exit from this one.
5. Complete or stop any ongoing asynchronous flush.
6. Flush most recent checkpoint if we still need to.
7. Disconnect logging functions.
8. Free internal data structures.

9. Call `DTCMP_Finalize` if used.

2.32 Flush

This section describes the process of a synchronous flush.

2.32.1 scr_flush_sync

This is implemented in `scr_flush_sync.c`.

1. Return with failure if flush is disabled.
2. Return with success if specified dataset id has already been flushed.
3. Barrier to ensure all procs are ready to start.
4. Start timer to record flush duration.
5. If async flush is in progress, wait for it to stop. Then check that our dataset still needs to be flushed.
6. Log start of flush.
7. Add FLUSHING marker for dataset in flush file to denote flush started.
8. Get list of files to flush, identify containers, create directories, and create containers (Section *0.1.2*). Store list in new hash.
9. Flush data to files or containers (Section *0.1.7*).
10. Write summary file (Section *0.1.9*).
11. Get total bytes from dataset hash in filemap.
12. Delete hashes of data and list of files.
13. Removing FLUSHING marker from flush file.
14. Stop timer, compute bandwidth, log end.

2.32.2 scr_flush_prepare

Given a filemap and dataset id, prepare and return a list of files to be flushed, also create corresponding directories and container files. This is implemented in `scr_flush.c`.

1. Build hash of files, directories, and containers for flush (Section *0.1.3*).
2. Create directory tree for dataset (Section *0.1.6*).
3. Create container files in `scr_flush_create_containers`.
    1. Loop over each file in file list hash. If the process writes to offset 0, have it open, create, truncate, and close the container file.

2.32.3 scr_flush_identify

Creates a hash of files to flush. This is implemented in `scr_flush.c`.

1. Check that all procs have all of their files for this dataset.
2. Add files to file list hash, including meta data in `scr_flush_identify_files`.

1. Read dataset hash from filemap, add to file list hash.
2. Loop over each file for dataset, if file is not `XOR` add it and its meta data to file list hash.

3. Add directories to file list hash (Section *0.1.4*).
4. Add containers to file list hash in (Section *0.1.5*).

2.32.4 scr_flush_identify_dirs

Specifies directories which must be created as part of flush, and identifies processes responsible for creating them. This is implemented in `scr_flush.c`.

1. Extract dataset hash from file list hash.
2. If we're preserving user directories:
    1. Allocate arrays to call DTCMP to rank directory names of all files.
    2. For each file, check that its user-specified path is under the prefix directory, insert dataset directory name in subdir array, and insert full parent directory in dir array.
    3. Check that all files from all procs are within a directory under the prefix directory.
    4. Call DTCMP with subdir array to rank user directory names for all files, and check that one common dataset directory contains all files.
    5. Broadcast dataset directory name to all procs.
    6. Record dataset directory in file list hash.
    7. Call DTCMP with the dir array to rank all directories across all procs.
    8. For each unique directory, we pick one process to later create that directory. This process records the directory name in the file list hash.
    9. Free arrays.
3. Otherwise (if we're not preserving user-defined directories):
    1. Get name of dataset from dataset hash.
    2. Append dataset name to prefix directory to define dataset directory.
    3. Record dataset directory in file list hash.
    4. Record dataset directory as destination path for each file in file list hash.

2.32.5 scr_flush_identify_containers

For each file to be flushed in file list hash, identify segments, containers, offsets, and length.s This is implemented in `scr_flush.c`.

1. Get our rank within the `scr_comm_node` communicator.
2. Get the container size.
3. Extract dataset hash from file list hash.
4. Define path within dataset directory to all container files.
5. Loop over each file in file list hash and compute total byte count.
6. Compute total bytes across all processes in run with allreduce on `scr_comm_world`.

7. Compute total bytes per node by reducing to node leader in `scr_comm_node`.
8. Compute offset for each node with a scan across node leaders in `scr_comm_node_across`.
9. Compute offset of processes within each node with scan within `scr_comm_node`.
10. Loop over each file and compute offset of each file.
11. Given the container size, the offset and length of each file, compute container file name, length, and offset for each segment and store details within file list hash.
12. Check that all procs identified their containers.

2.32.6 scr_flush_create_dirs

Given a file list hash, create dataset directory and any subdirectories to hold dataset. This is implemented in `scr_flush.c`.

1. Get file mode for creating directories.
2. Rank 0 creates the dataset directory:
    1. Read path from file list hash.
    2. Get subdirectory name of dataset within prefix directory.
    3. Extract dataset hash from file list hash, and get dataset id.
    4. Add dataset directory and id to index file, write index file to disk.
    5. Create dataset directory and its hidden `.scr` subdirectory.
3. Barrier across all procs.
4. Have each leader create its directory as designated in Section *0.1.4*.
5. Ensure that all directories were created.

2.32.7 scr_flush_data

This is implemented in `scr_flush_sync.c`. To flow control the number of processes writing, rank 0 writes its data first and then serves as a gate keeper. All processes wait until they receive a go ahead message from rank 0 before starting, and each sends a message back to rank 0 when finished. Rank 0 maintains a sliding window of active writers. Each process includes a flag indicating whether it failed or succeeded to copy its files. If rank 0 detects that a process fails, the go ahead message it sends to other writers indicates this failure, in which that writer immediate sends back a message without copying its files. This way time is not wasted by later writers if an earlier writer has already failed.

RANK 0

1. Flush files in list, writing data to containers if used (Section *0.1.8*).
2. Allocate arrays to manage a window of active writers.
3. Send “go ahead” message to first W writers.
4. Waitany for any writer to send completion notification, record flag indicating whether that writer was successful, and send “go ahead” message to next writer.
5. Loop until all writers have completed.
6. Execute allreduce to inform all procs whether flush was successful.

NON-RANK 0

1. Wait for go ahead message.
2. Flush files in list, writing data to containers if used (Section *0.1.8*).
3. Send completion message to rank 0 indicating whether copy succeeded.
4. Execute allreduce to inform all procs whether flush was successful.

2.32.8 scr_flush_files_list

Given a list of files, this function copies data file-by-file, and then it updates the hash that forms the rank2file map. It is implemented in `scr_flush_sync.c`.

1. Get path to summary file from file list hash.
2. Loop over each file in file list.

LOOP

1. Get file name.
2. Get basename of file (throw away directory portion).
3. Get hash for this file.
4. Get file meta data from hash.
5. Check for container segments (TODO: what if a process has no files?).

CONTAINERS

1. Add basename to rank2file map.
2. Flush file to its containers.
3. If successful, record file size, CRC32 if computed, and segment info in rank2file map.
4. Otherwise, record 0 for COMPLETE flag in rank2file map.
5. Delete file name and loop.

NON-CONTAINERS

1. Get directory to write file from PATH key in file hash.
2. Append basename to directory to get full path.
3. Compute relative path to file starting from dataset directory.
4. Add relative path to rank2file map.
5. Copy file data to destination.
6. If successful, copy file size and CRC32 if computed in rank2file map.
7. Otherwise, record 0 for COMPLETE flag in rank2file map.
8. Delete relative and full path names and loop.

END LOOP

2.32.9 scr_flush_complete

Writes out summary and rank2file map files. This is implemented in `scr_flush.c`.

1. Extract dataset hash from file list hash.
2. Get dataset directory path.
3. Write summary file (Section *0.1.10*).
4. Update index file to mark dataset as "current".
5. Broadcast signal from rank 0 to indicate whether flush succeeded.
6. Update flush file that dataset is now on parallel file system.

2.32.10 scr_flush_summary

Produces summary and rank2file map files in dataset directory on parallel file system. Data for the rank2file maps are gathered and written via a data-dependent tree, such that no process has to write more than 1MB to each file. This is implemented in `scr_flush.c`.

1. Get path to dataset directory and hidden `.scr` directory.
2. Given data to write to rank2file map file, pick a writer process so that each writer gets at most 1MB.
3. Call `scr_hash_exchange_direction` to fold data up tree.
4. Rank 0 creates summary file and writes dataset hash.
5. Define name of rank2file map files.
6. Funnel rank2file data up tree in recursive manner (Section *0.1.11*).
7. If process is a writer, write rank2file map data to file.
8. Free temporaries.
9. Check that all procs wrote all files successfully.

2.32.11 scr_flush_summary_map

Produces summary and rank2file map files in dataset directory on parallel file system. This is implemented in `scr_flush.c`.

1. Get path to dataset directory and hidden `.scr` directory.
2. If we received rank2file map in the previous step, create hash to specify its file name to include at next level in tree.
3. Given this hash, pick a writer process so that each writer gets at most 1MB.
4. Call `scr_hash_exchange_direction` to fold data up tree.
5. Define name of rank2file map files.
6. Funnel rank2file data up tree by calling `scr_flush_summary_map` recursively..
7. If process is a writer, write rank2file map data to file.
8. Free temporaries.

2.33 Scavenge

SCR commands should be executed after the final run of the application in a resource allocation to check that the most recent checkpoint is successfully copied to the parallel file system before exiting the allocation. This logic is encapsulated in the `scr_postrun` command.

2.33.1 scripts/common/scr_postrun.in

Checks whether there is a dataset in cache that must be copied to the parallel file system. If so, scavenge this dataset, rebuild any missing files if possible, and finally update SCR index file in prefix directory.

1. Interprets `$SCR_ENABLE`, bails with success if set to 0.
2. Interprets `$SCR_DEBUG`, enables verbosity if set > 0.
3. Invokes `scr_prefix` to determine prefix directory on parallel file system (but this value is overridden via "-p" option when called from `scr_srun`).
4. Interprets `$SCR_NODELIST` to determine set of nodes job is using, invokes `scr_env --nodes` if not set.
5. Invokes `scr_list_down_nodes` to determine which nodes are down.
6. Invokes `scr_glob_hosts` to subtract down nodes from node list to determine which nodes are still up, bails with error if there are no up nodes left.
7. Invokes `scr_list_dir control` to get the control directory.
8. Invokes "`scr_flush_file -dir $pardir -latest`" providing prefix directory to determine id of most recent dataset.
9. If this command fails, there is no dataset to scavenge, so bail out with error.
10. Invokes "`scr_inspect -up $UPNODES -from $cntldir`" to get list of datasets in cache.
11. Invokes "`scr_flush_file -dir $pardir -needflush $id`" providing prefix directory and dataset id to determine whether this dataset needs to be copied.
12. If this command fails, the dataset has already been flushed, so bail out with success.
13. Invokes "`scr_flush_file -dir $pardir -subdir $id`" to get name for dataset directory.
14. Creates dataset directory on parallel file system, and creates hidden `.scr` directory.
15. Invokes `scr_scavenge` providing control directory, dataset id to be copied, dataset directory, and set of known down nodes, which copies dataset files from cache to the PFS.
16. Invokes `scr_index` providing dataset directory, which checks whether all files are accounted for, attempts to rebuild missing files if it can, and records the new directory and status in the SCR index file.
17. If dataset was copied and indexed successfully, marks the dataset as current in the index file.

2.33.2 scripts/TLCC/scr_scavenge.in

Executes within job batch script to manage scavenge of files from cache to parallel file system. Uses `scr_hostlist.pm` and `scr_param.pm`.

1. Uses `scr_param.pm` to read `SCR_FILE_BUF_SIZE` (sets size of buffer when writing to file system).
2. Uses `scr_param.pm` to read `SCR_CRC_ON_FLUSH` (flag indicating whether to compute CRC32 on file during scavenge).

3. Uses `scr_param.pm` to read `SCR_USE_CONTAINERS` (flag indicating whether to combine application files into container files).
4. Invokes "`scr_env -jobid`" to get jobid.
5. Invokes "`scr_env -nodes`" to get the current nodeset, can override with "`-jobset`" on command line.
6. Logs start of scavenge operation, if logging is enabled.

START ROUND 1

1. Invokes `pdsh` of `scr_copy` providing control directory, dataset id, dataset directory, buffer size, CRC32 flag, partner flag, container flag, and list of known down nodes.
2. Directs `stdout` to one file, directs `stderr` to another.
3. Scan `stdout` file to build list of partner nodes and list of nodes where copy command failed.
4. Scan `stderr` file for a few well-known error strings indicating pdsh failed.
5. Build list of all failed nodes and list of nodes that were partners to those failed nodes, if any.
6. If there were any nodes that failed in ROUND 1, enter ROUND 2.

END ROUND 1, START ROUND 2

1. Build list of updated failed nodes, includes nodes known to be failed before ROUND 1, plus any nodes detected as failed in ROUND 1.
2. Invokes `pdsh` of `scr_copy` on partner nodes of failed nodes (if we found a partner for each failed node) or on all non-failed nodes otherwise, provided the updated list of failed nodes.

END ROUND 2

1. Logs end of scavenge, if logging is enabled.

2.33.3 `src/scr_copy.c`

Serial process that runs on a compute node and copies files for specified dataset to parallel file system.

1. Read control directory, dataset id, destination dataset directory, etc from command line.
2. Read master filemap and each filemap it lists.
3. If specified dataset id does not exist, we can't copy it so bail out with error.
4. Loop over each rank we have for this dataset.

RANK LOOP

1. Get flush descriptor from filemap. Record partner if set and whether we should preserve user directories or use containers.
2. If partner flag is set, print node name of partner and loop.
3. Otherwise, check whether we have all files for this rank, and if not loop to next rank.
4. Otherwise, we'll actually start to copy files.
5. Allocate a rank filemap object and set expected number of files.
6. Copy dataset hash into rank filemap.
7. Record whether we're preserving user directories or using containers in the rank filemap.
8. Loop over each file for this rank in this dataset.

FILE LOOP

1. Get file name.
2. Check that we can read the file, if not record an error state.
3. Get file meta data from filemap.
4. Check whether file is application or SCR file.
5. If user file, set destination directory. If preserving directories, get user-specified directory from meta data and call mkdir.
6. Create destination file name.
7. Copy file from cache to destination, optionally compute CRC32 during copy.
8. Compute relative path to destination file from dataset directory.
9. Add relative path to file to rank filemap.
10. If CRC32 was enabled and also was set on original file, check its value, or if it was not already set, set it in file meta data.
11. Record file meta data in rank filemap.
12. Free temporaries.

END FILE LOOP

1. Write rank filemap to dataset directory.
2. Delete rank filemap object.

END RANK LOOP

1. Free path to dataset directory and hidden `.scr` directory.
2. Print and exit with code indicating success or error.

2.33.4 src/scr_index.c

Given a dataset directory as command line argument, checks whether dataset is indexed and adds to index if not. Attempts to rebuild missing files if needed.

1. If "`-add`" option is specified, call `index_add_dir` (Section *0.1.5*) to add directory to index file.
2. If "`-remove`" option is specified, call `index_remove_dir` to delete dataset directory from index file. Does not delete associated files, only the reference to the directory from the index file.
3. If "`-current`" option is specified, call `index_current_dir` to mark specified dataset directory as current. When a dataset is marked as current, SCR attempts to restart the job from that dataset and works backwards if it fails.
4. If "`-list`" option is specified, call `index_list` to list contents of index file.

2.33.5 index_add_dir

Adds specified dataset directory to index file, if it doesn't already exist. Rebuilds files if possible, and writes summary file if needed.

1. Read index file.
2. Lookup dataset directory in index file, if it's already indexed, bail out with success.
3. Otherwise, concatenate dataset subdirectory name with prefix directory to get full path to the dataset directory.

4. Attempt to read summary file from dataset directory. Call `scr_summary_build` (Section *0.1.6*) if it does not exist.
5. Read dataset id from summary file, if this fails exit with error.
6. Read completeness flag from summary file.
7. Write entry to index hash for this dataset, including directory name, dataset id, complete flag, and flush timestamp.
8. Write hash out as new index file.

2.33.6 scr_summary_build

Scans all files in dataset directory, attempts to rebuild files, and writes summary file.

1. If we can read the summary file, bail out with success.
2. Call `scr_scan_files` (Section *0.1.7*) to read meta data for all files in directory. This records all data in a scan hash.
3. Call `scr_inspect_scan` (Section *0.1.9*) to examine whether all files in scan hash are complete, and record results in scan hash.
4. If files are missing, call `scr_rebuild_scan` (Section *0.1.10*) to attempt to rebuild files. After the rebuild, we delete the scan hash, rescan, and re-inspect to produce an updated scan hash.
5. Delete extraneous entries from scan hash to form our summary file hash (Section *Summary file*).
6. Write out summary file.

2.33.7 scr_scan_files

Reads all filemap and meta data files in directory to build a hash listing all files in dataset directory.

1. Build string to hidden `.scr` subdirectory in dataset directory.
2. Build regular expression to identify `XOR` files.
3. Open hidden directory.

BEGIN LOOP

1. Call `readdir` to get next directory item.
2. Get item name.
3. If item does not end with "`.scrfilemap`", loop.
4. Otherwise, create full path to file name.
5. Call `scr_scan_file` to read file into scan hash.
6. Free full path and loop to next item.

END LOOP

2.33.8 scr_scan_file

1. Create new rank filemap object.
2. Read filemap.

3. For each dataset id in filemap...
4. Get dataset id.
5. Get scan hash for this dataset.
6. Lookup rank2file map in scan hash, or create one if it doesn't exist.
7. For each rank in this dataset...
8. Get rank id.
9. Read dataset hash from filemap and record in scan hash.
10. Get rank hash from rank2file hash for the current rank.
11. Set number of expected files.
12. For each file for this rank and dataset...
13. Get file name.
14. Build full path to file.
15. Get meta data for file from rank filemap.
16. Read number of ranks, file name, file size, and complete flag for file.
17. Check that file exists.
18. Check that file size matches.
19. Check that number of ranks we expect matches number from meta data, use this value to set the expected number of ranks if it's not already set.
20. If any check fails, skip to next file.
21. Otherwise, add entry for this file in scan hash.
22. If meta data is for an `XOR` file, add an `XOR` entry in scan hash.

2.33.9 scr_inspect_scan

Checks that each rank has an entry in the scan hash, and checks that each rank has an entry for each of its expected number of files.

1. For each dataset in scan hash, get dataset id and pointer to its hash entries.
2. Lookup rank2file hash under `RANK2FILE` key.
3. Lookup hash for `RANKS` key, and check that we have exactly one entry.
4. Read number of ranks for this dataset.
5. Sort entries for ranks in scan hash by rank id.
6. Set expected rank to 0, and iterate over each rank in loop.

BEGIN LOOP

1. Get rank id and hash entries for current rank.
2. If rank id is invalid or out of order compared to expected rank, throw an error and mark dataset as invalid.
3. While current rank id is higher than expected rank id, mark expected rank id as missing and increment expected rank id.
4. Get `FILES` hash for this rank, and check that we have exactly one entry.

5. Read number of expected files for this rank.
6. Get hash of file names for this rank recorded in `scr_scan_files`.
7. For each file, if it is marked as incomplete, mark rank as missing.
8. If number of file entries for this rank is less than expected number of files, mark rank as missing.
9. If number of file entries for this rank is more than expected number of files, mark dataset as invalid.
10. Increment expected rank id.

END LOOP

1. While expected rank id is less than the number of ranks for this dataset, mark expected rank id as missing and increment expected rank id.
2. If expected rank id is more than the number of ranks for this dataset, mark dataset as invalid.
3. Return `SCR_SUCCESS` if and only if we have all files for each dataset.

2.33.10 scr_rebuild_scan

Identifies whether any files are missing and forks and execs processes to rebuild missing files if possible.

1. Iterate over each dataset id recorded in scan hash.
2. Get dataset id and its hash entries.
3. Look for flag indicating that dataset is invalid. We assume the dataset is bad beyond repair if we find such a flag.
4. Check whether there are any ranks listed as missing files for this dataset, if not, go to next dataset.
5. Otherwise, iterate over entries for each `XOR` set.

BEGIN LOOP

1. Get `XOR` set id and number of members for this set.
2. Iterate over entries for each member in the set. If we are missing an entry for the member, or if we have its entry but its associated rank is listed as one of the missing ranks, mark this member as missing.
3. If we are missing files for more than one member of the set, mark the dataset as being unrecoverable. In this case, we won't attempt to rebuild any files.
4. Otherwise, if we are missing any files for the set, build the string that we'll use later to fork and exec a process to rebuild the missing files.

END LOOP

1. If dataset is recoverable, call `scr_fork_rebuilds` to fork and exec processes to rebuild missing files. This forks a process for each missing file where each invokes `scr_rebuild_xor` utility, implemented in `scr_rebuild_xor.c`. If any of these rebuild processes fail, then consider the rebuild as failed.
2. Return `SCR_SUCCESS` if and only if, for each dataset id in the scan hash, the dataset is not explicitly marked as bad, and all files already existed or we were able to rebuild all missing files.

2.34 Testing SCR on a New Systems

To verify that SCR works a new system, there are number of steps and failure modes to check. Usually, these tests require close coordination with a system administrator. It often takes 2-3 months to ensure that everything can work.

2.34.1 Restart in Place

1. Configure a job allocation so that it does not automatically terminate due to a node failure. Most resource managers can support this, but they are often not configured that way by default, because most centers don't care to continue through those kinds of failures. The only way to know for sure is to get a job allocation, and have a system admin actually kill a node, e.g., by powering it off. For this, create a job script that sleeps for some time and then prints some messages after sleeping. Start the job, then kill a node during the sleep phase, and then verify that the print commands after the sleep still execute.
2. Ensure that a running MPI job actually exits and returns control to the job script. If a node failure causes MPI to hang so that control never comes back to the job script, that's not going to work. To do this, you can launch an MPI job that does nothing but runs in a while loop or sleeps for some time. Then have an admin kill a node during that sleep phase, and verify that some commands in the job script that come after the mpirun still execute. Again, simply printing a message is often enough.

3. Detect which node has failed. Is there a way to ask the system software about failed nodes? If not, or if the system software is too slow to report it, can we run commands to go inspect things ourselves? Our existing scripts do various tests, like ping each node that is thought to be up, and then if a node responds to a ping, we run additional tests on each node. There are failure modes where the node will respond to ping, but the SSD or the GPU may have died, so you also need to run tests on those devices to ensure that the node and all of its critical components are healthy. 4. Verify that you can steer the next job launch to only run on healthy nodes. There are two things here. First is to avoid the bad nodes in the next MPI job. For example, in SLURM one can use the -w option to specify the target node set, and with mpirun, there is often a hostfile that one can create. For CORAL, jobs are launched with jsrun, and that command has its own style of hostfile that we need to generate. The second thing is to verify that the system supports a second MPI job to run in a broken allocation after a first MPI job was terminated unexpectedly. 5. Finally verify the complete cycle work with a veloc checkpoint.

a. Allocate 5 nodes

b. Write a single job script that launches an MPI job on to the first 4 nodes, saves a veloc checkpoint, then spins in a loop

c. Have an admin kill one of those 4 nodes

d. Verify that the MPI job exits, the scripting detects the down node and builds the command to avoid it in the next launch

e. Launch a new MPI job on the remaining 4 healthy nodes, verify that job rebuilds and restarts from latest veloc checkpoint

2.34.2 Node Health Checks

1. Ask resource manager if it knows of any down nodes if there is a way to do that. This helps us immediately exclude nodes the system already knows to be down. SLURM can do this, though it may take time to figure it out (5-10 minutes).
2. Attempt to ping all nodes that the resource manager thinks are up or simply all nodes if resource manager can't tell us anything. Exclude any nodes that fail to respond. Some systems are not configured to allow normal users to ping the compute nodes, so this step may be skipped.
3. Try to run a test on each node that responds to the *ping*, using *pdsh* if possible, but *aprun* or other system launchers if *pdsh* doesn't work. The tests one needs to run can vary. One might simply run an *echo* command to print something. On a system with GPUs, you might want to run a test on the GPUs to verify they haven't gone out to lunch. On a with SSDs, the SSDs would fail in various ways. This required the following checks: a. run *df* against the SSD mount point and check that the total space looks right (a firmware bug caused some drives to lose track of their capacity) b. ensure that the mount point was listed as writable. Sometimes the bits would turn off (another firmware bug) c. try to touch and delete a file and verify that works. Some drives looked ok by the first two checks, but you couldn't actually create a file (a still different firmware bug)

4. Additionally, the *SCR_EXCLUDE_NODES* environment variable allows a user to list nodes they want to avoid. This is a catch-all so that a user could make progress through a weekend in case they found a problematic node that our other checks didn't pick up. That came into existence when a user stumbled into a node with a bad CPU one weekend that would generate segfaults. We didn't have any checks for it, nor did the system, so the jobs kept getting scheduled to the bad node, even though it was clear to the user that the node was bogus.
5. Finally, the SCR scripting was written so that once a node is ruled out for any of the above, we always keep it out for the remainder of the job allocation. That way we'd avoid retrying problematic nodes that would come and go.

This logic is often system specific and is stored in the *scr_list_down_nodes* script.

2.35 Bamboo Test Suite

SCR's bamboo testing suite does a few things:

1. Test that SCR will build with CMake
2. Trigger SCR's built-in testing with *make test* (uses ctest)
3. Test that SCR will build with Spack
4. Trigger more advanced test by running the *testing/TEST* script

This document records some of the scripts for the bamboo test suite. Within bamboo, the test plans have the ability to specify different machines which on which a particular step can be run. In this way, SCR can be tested for each platform.

2.35.1 Bamboo Test Plan Overview

1. Clone SCR from the repository and check out the particular branch which is being tested. This functionality is built-in to bamboo and does not have an associated script.
2. **Build and Make**:

```
#!/bin/bash

. /etc/profile
. /etc/bashrc

mkdir build install
cd build
cmake -DCMAKE_INSTALL_PREFIX=../install ../SCR
make
make install
```

3. **Test parallel**:

```
cd build
ctest --verbose -T Test -R parallel*
mkdir Testing/Tests
cp `grep -rl '.xml$' Testing/*` Testing/*/* Testing/Tests
```

4. **Test serial**:

```
cd build
ctest --verbose -T Test -R serial*
```

5. Bamboo has a built-in CTest test parser. This is configured with the test file path pattern: ***/Testing/*/*.xml*
6. Clone spack from its repository.
7. **Install SCR**:

```
#!/bin/bash -l
. /etc/profile
. /etc/bashrc

cd spack
sed -i "s#/.spack#/.spack-${SYS_TYPE}#" lib/spack/spack/__init__.py
. share/spack/setup-env.sh
#spack compiler find

spack install --keep-stage scr@develop resource_manager=SLURM
#spack install --run-tests scr@develop resource_manager=NONE

spack cd -i scr
cd share/scr/examples
export MPICC=/usr/tce/packages/mvapich2/mvapich2-2.2-gcc-4.9.3/bin/mpicc
export MPICXX=/usr/tce/packages/mvapich2/mvapich2-2.2-gcc-4.9.3/bin/mpicxx
export MPIF77=/usr/tce/packages/mvapich2/mvapich2-2.2-gcc-4.9.3/bin/mpif77
export MPIF90=/usr/tce/packages/mvapich2/mvapich2-2.2-gcc-4.9.3/bin/mpif90
make
```

8. **Run SCR/Testing/TEST python script**:

```
#!/bin/bash -l
# This script takes 1 variable, the script you want to run.
# This variable, $1, comes from the bamboo command line.
# Here, $1 = TEST

. spack/share/spack/setup-env.sh

# setup environment for script to be run

spack cd scr@develop
export SCR_PKG=`pwd`

spack cd -i scr@develop
export SCR_INSTALL=`pwd`

cp $SCR_PKG/testing/$1 $SCR_INSTALL/bin/$1

cd $SCR_INSTALL/bin
export SCR_LAUNCH_DIR=`pwd`

# submit job
jobID=$(sbatch --export=ALL -ppbatch -n4 -N4 -t5 -J SCR-TESTS -o bamboo_test_$1.
↪out $1 | tr -dc [:digit:])

# watch and wait until job has completed
# this no longer works because bamboo has its own timeout
#jobInfo=$(mdiag -j ${jobID})
#jobStatus=$(echo $jobInfo | awk '{print $13}')

#while [ $jobStatus != "CG" ] && [ $jobStatus != "CD" ]
#do
```

(continues on next page)

(continued from previous page)

```
#    jobInfo=$(mdiag -j ${jobID})
#    jobStatus=$(echo $jobInfo | awk '{print $13}')
#done

# watch and wait until job has completed
jobStatus=$(checkjob ${jobID} | grep State | awk '{print $2}')

count=1
while [ "$jobStatus" != "Completed" ]; do
jobStatus=$(checkjob ${jobID} | grep State | awk '{print $2}')
if ([ "$jobStatus" = "Idle" ] || [ "$jobStatus" = "Resources" ]) && [ $((count %␣
↪60)) -eq 0 ]; then
echo "Job $jobID waiting for resources"
count=1
fi
((count++))
sleep 1
done

checkjob ${jobID}

# print results of script
if [ -e bamboo_test_$1.out ]; then
cat bamboo_test_$1.out
else
echo "File bamboo_test_$1.out does not exist"
exit 1
fi

# determine if script was successful
result=$(cat bamboo_test_$1.out | tail -n5 | grep -o PASSED)

# post test cleanup
rm -rf .scr/ ckpt.*

if [ "$result" != "PASSED" ]; then
exit 1
fi

exit 0
```

2.36 GitLab CI Testing

SCR's GitLab testing suite does a few things:

1. Test that SCR will build with CMake
2. Trigger SCR's built-in testing with *make test* (uses ctest)
3. Test that SCR will build with Spack
4. Trigger more advanced test by running the *testing/TEST* script

Bibliography

[Vaidya] "A Case for Two-Level Recovery Schemes", Nitin H. Vaidya, IEEE Transactions on Computers, 1998, http://doi.ieeecomputersociety.org/10.1109/12.689645.

[Ross] W. Gropp, R. Ross, and N. Miller, "Providing Efficient I/O Redundancy in MPI Environments," in Lecture Notes in Computer Science, 3241:7786, September 2004. 11th European PVM/MPI Users Group Meeting, 2004.

[RAID5] D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in Proc. of 1988 ACM SIGMOD Conf. on Management of Data, 1988.

[Ross] W. Gropp, R. Ross, and N. Miller, "Providing Efficient I/O Redundancy in MPI Environments," in Lecture Notes in Computer Science, 3241:7786, September 2004. 11th European PVM/MPI Users Group Meeting, 2004.

[RAID5] D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in Proc. of 1988 ACM SIGMOD Conf. on Management of Data, 1988.