SCR v1.1-8 Developer's Manual

April 11, 2011

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344.

Contents

1 Introduction

This document describes implementation details of SCR version 1.1-8. It is intended to define essential concepts and to provide high-level explanation on how SCR is implemented so that one may better understand the source code.

2 Overview

This section covers basic concepts and terms used throughout the SCR documentation and source code. Due to legacy reasons, the meanings of some terms are overloaded in different contexts.

2.1 Integrating 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 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 */

  /* 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 illustrates the changes necessary to integrate SCR into such an application. 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 */

  /**** 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. 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 data. 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()`. When 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 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.

2.2 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. Within an allocation, a user may *run* a simulation one or more times. 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*. So in other words, one or more runs occur within an allocation, and one or more allocations occur within a job.

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.

2.3 Communicators, levels, and master processes

As shown in Figure 1, the SCR library creates and caches a number of MPI communicators in global variables. These communicators are created during `SCR_Init()` and they are freed during `SCR_Finalize()`. The variables are defined in `scr.c`.

First, 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.

Then, each process splits `scr_comm_world` into a communicator called `scr_comm_local`. This communicator contains all processes on the same node as the calling process. This is accomplished by splitting the communicator based on the hostname (or the IP address) of the calling process. Processes in `scr_comm_local` are ordered relative to their rank in `scr_comm_world`. Each process caches its rank and the size of `scr_comm_local` in `scr_my_rank_local` and `scr_ranks_local`, respectively. SCR makes frequent use of a process's rank in `scr_comm_local`. We refer to this value as a process's *level*.

Finally, each process splits `scr_comm_world` into a communicator called `scr_comm_level`. This communicator is guaranteed to contain at most one process from each node. To build this communicator, a process specifies its level (i.e., its rank in `scr_comm_local`) as the color value in `MPI_Comm_split()`. For a given level value, there is at most one process on each node belonging to that level, so after the split, `scr_comm_level` has at most one process from each node. Again, processes are ordered in `scr_comm_level` relative to their rank in `scr_comm_world`. And again, each process caches its rank and the size of `scr_comm_level` in `scr_my_rank_level` and `scr_ranks_level`, respectively.

Note that in a run that uses a different number of processes per node, the the size of `scr_comm_local` will be different on different processes. The same also holds for `scr_comm_level`.

SCR makes use of these different communicators in various ways. For global communication, SCR uses `scr_comm_world`, whereas it typically uses `scr_comm_local` to coordinate among processes on the same node. In particular, the current implementation is designed to work with node-local storage. Any time SCR needs one process to do something with each node-local storage device, such as create a directory, it selects the process on each node that has rank 0 in `scr_comm_local` (level = 0). In fact, this particluar process is used so often, we give it a special name. We call a process that has rank 0 in `scr_comm_local` the *master process* throughout the code. To communicate information between the master and the rest of the processes on the same node, the `scr_comm_local` communicator is often used.

Meanwhile, the `scr_comm_level` communicator is mostly used when assigning processes to a redundancy set. While one may implement redundancy schemes to handle all types of failures, in practice, the most common failures are single compute node failures. Thus, the current redundancy schemes are designed to ensure that two processes from the same compute node are not assigned to the same redundancy set. Since `scr_comm_level` ensures that it has at most one process from each node, both PARTNER and XOR select redundancy sets by picking subsets of the processes from `scr_comm_level`.

2.4 Control, cache, and prefix directories

SCR manages numerous files and directories to record its internal state and to cache checkpoint data. As shown in Figure 2, there are three fundamental types of directories: control, cache, and prefix directories. For a detailed

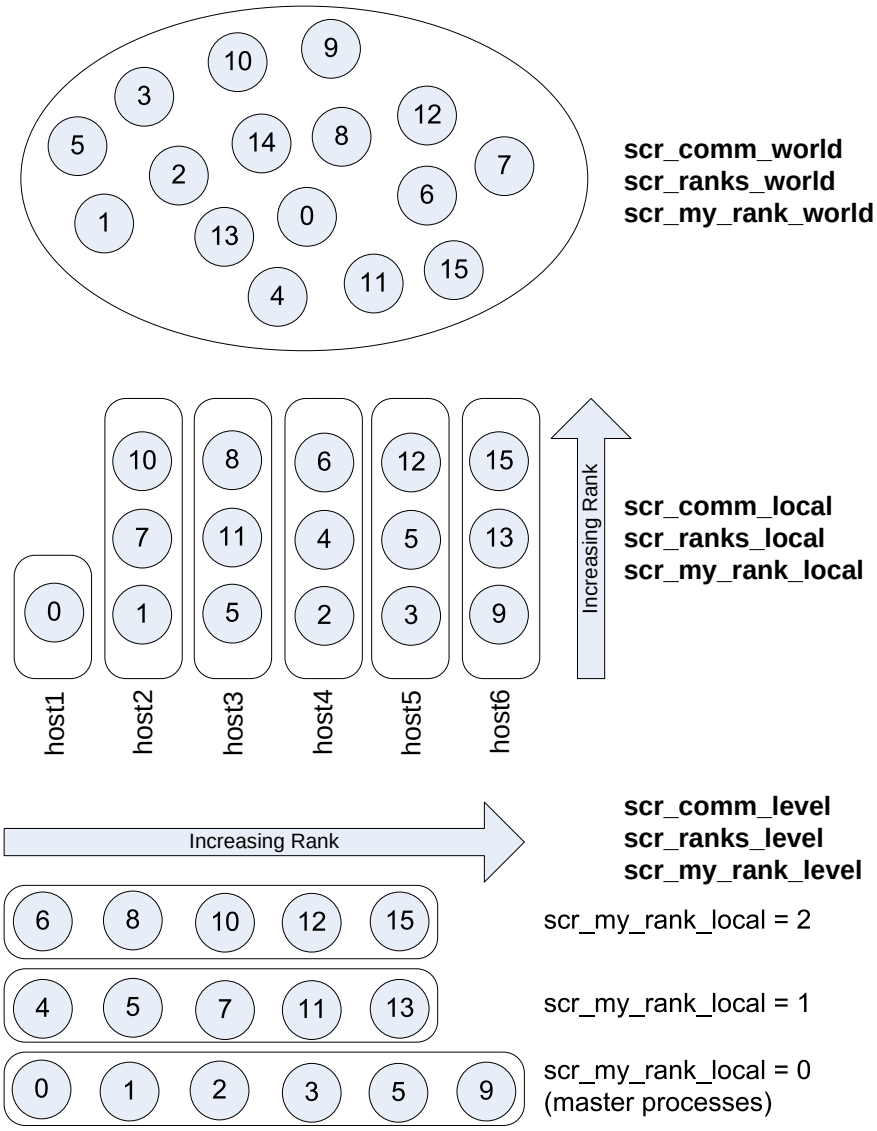


Figure 1: SCR Communicators

illustration of how these files and directories are arranged, see the example presented in Section 10.

The *control directory* is where SCR writes files to store its internal state. In the current implementation, this directory is expected to be stored in node-local storage. SCR writes numerous, small files in the control directory, and it may access these files frequently. It's 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 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>/<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 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 and associated meta data files in these subdirectories. The device serving the cache directory must be large enough to hold the data for one or more checkpoints. Multiple cache directories may be utilized in the same run, which enables SCR to use more than one storage device within a run. Cache directories are expected to be located in node-local storage.

Similar to the control directory, to construct the full path of a cache directory, SCR incorporates a 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>/<allocationid>`. A set of valid 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.

In the current implementation, the user must configure the maximum number of checkpoints SCR should keep in each cache base directory. It is up to the user to ensure that the capacity of the device associated with the base directory is large enough to hold the number of checkpoints the user specifies. Internally, SCR maintains a *cache descriptor* for each base directory to record the maximum number of checkpoints to keep in that base directory.

Each checkpoint is written to its own subdirectory within the cache directory, and this subdirectory is referred to as a *checkpoint directory*. SCR assigns a unique number to each checkpoint, called the *checkpoint id*. Further, SCR associates each checkpoint with a *checkpoint descriptor*. The checkpoint descriptor describes which cache directory should be used and how often that cache directory should be used to store checkpoints (Section 6). A single run employs a set of one or more checkpoint descriptors, and each descriptor is assigned a unique integer index counting up from 0. When starting a new checkpoint, SCR selects a checkpoint descriptor, and then it creates the checkpoint directory within the cache directory that is associated with the descriptor. To construct the path of this checkpoint directory, SCR incorporates the index of the selected checkpoint descriptor along with the checkpoint id. This path is then appended to the cache directory, such that the full path of the checkpoint directory is of the form: `<cachebase>/<username>/<allocationid>/<ckptdescindex>/<ckptid>`.

Finally, the *prefix directory* is a directory the user specifies in the parallel file system. SCR fetches, flushes, and drains checkpoint files to subdirectories within the prefix directory for permanent storage (Section 2.5). 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. SCR maintains an index file within the prefix directory (Section 9.1). The index file records status information for each checkpoint.

For each checkpoint SCR stores in the prefix directory, SCR creates and manages a *checkpoint directory*. The checkpoint directory holds all files associated with a particular checkpoint, including application checkpoint files, as well as, SCR redundancy data files and meta data files.

Note that the term "checkpoint directory" is overloaded. In some cases, we use this term to refer to a directory in cache that contains checkpoint files, 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.

2.5 Fetches, flushes, and drains

SCR manages the transfer of checkpoint files between the prefix directory on the parallel file system and the cache directories in node-local storage. We use the term *fetch* to refer to the action of copying a set of checkpoint files from the parallel file system to cache. When transferring data in the other direction, there are two terms used:

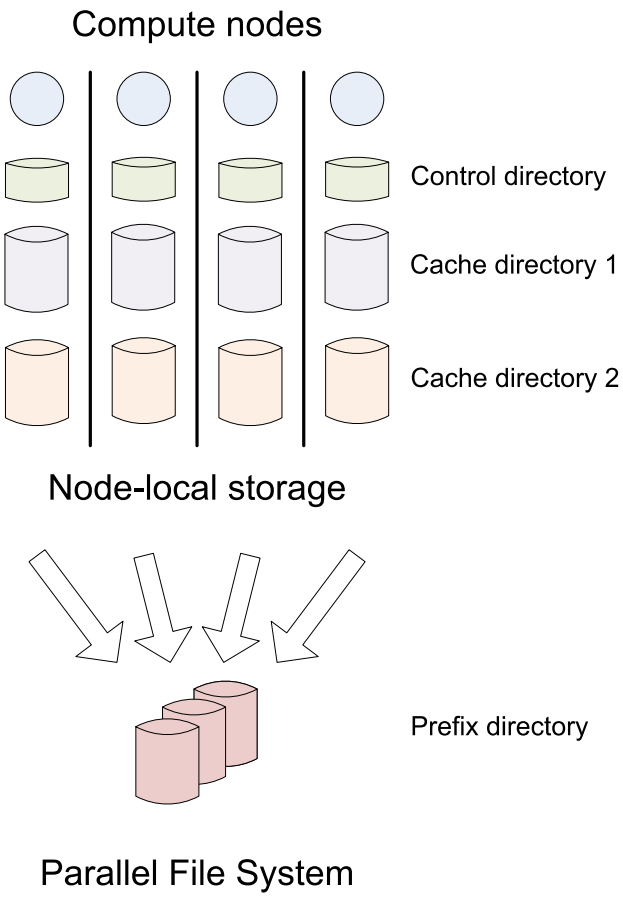


Figure 2: SCR Directories

flush and *drain*. Under normal circumstances, the library directly copies files from cache to the parallel file system. We refer to this transfer as a flush. However, sometimes a run is killed before the library can complete (or even start) 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 resource allocation expires. When a checkpoint is copied to the parallel file system via these commands (rather than through the library), we refer to this transfer as a drain.

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) is used to copy the checkpoint 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.

2.6 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 bash shell syntax or in PERL acquire parameter values through the `scr_param.pm` PERL module (Section 11.1.2). 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 initializing the `scr_param` interface in an MPI job, as in the case of the library, 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.

3 Hash

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
          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.

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`.

3.2.1 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

3.2.2 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);
}
```

3.2.3 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. In such cases, it is necessary to first unset a key before setting the new value, when attempting to change the 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(hash, "TIMESTEP", 21);
```

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

```
TIMESTEP
  20
  21
```

3.2.4 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);
```

3.2.5 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 buf.

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

Unpack hash from buffer into given hash object.

```
scr_hash_unpack(buf, hash);
```

3.2.6 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 will create the file if it does not exist, and it overwrites any existing file. To read a hash from a file (unpacks 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's 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 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.

3.2.7 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. It does not clear `hash` before unpacking the received hash, so it effectively merges the received hash into `hash`.

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.

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
```

3.4 Binary format

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

3.4.1 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 ELEMENT	Sequence of packed elements of length Count.

Format of a PACKED ELEMENT:

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

3.4.2 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 HASH	Packed hash data (see Section 3.4.1).
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.

4 Filemap

4.1 Overview

The `scr_filemap` data structure maintains the mapping between files, process ranks, and checkpoints. In a given checkpoint, each process may write zero or more files. SCR uses the filemap to record which rank writes which file in which checkpoint. 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 checkpoint files are stored in cache. The library uses this information to determine which checkpoints and which ranks it has data for. The library also uses this data to know which files to remove when deleting a checkpoint from cache, and it uses this data to know which files to copy when flushing a checkpoint to the parallel file system.

SCR internally numbers each checkpoint with a unique *checkpoint id*. It assigns checkpoint ids starting at 0 and counts up with each successive checkpoint. Functions that return checkpoint ids return -1 if there is no valid checkpoint contained in the filemap that matches a particular query.

The `scr_filemap` makes heavy use of the `scr_hash` data structure (Section 3). The `scr_hash` is utilized in the `scr_filemap` API and its implementation.

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 checkpoint ids 10 and 11.

```
CKPT
  10
    RANK
      20
  11
    RANK
      20
      28
RANK
  20
    CKPT
      10
        EXPECT
          2
        FILE
          /<path_to_foo_20.ckpt.10>/foo_20.ckpt.10
          /<path_to_foo_20.ckpt.10.xor>/foo_20.ckpt.10.xor
        CKPTDESC
          <checkpoint descriptor hash>
      11
        EXPECT
          1
        FILE
          /<path_to_foo_20.ckpt.11>/foo_20.ckpt.11
        CKPTDESC
          <checkpoint descriptor hash>
  28
    CKPT
      11
        EXPECT
          1
        FILE
          /<path_to_foo_28.ckpt.11>/foo_28.ckpt.11
        PARTNER
          atlas56
        CKPTDESC
          <checkpoint descriptor hash>
```

Note the full path to each file is recorded, along with the expected number of files and the "checkpoint descriptor" hash. In addition, there may be arbitrary tags such as the `PARTNER` element.

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, checkpoint ids are listed in the hash of a `CKPT` element. Finally, each checkpoint id contains elements for the expected number of files (under `EXPECT`), the file names (under `FILE`), and the checkpoint descriptor hash (under `CKPTDESC`, see Section 6) describing the redundancy scheme applied to the checkpoint files.

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

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`.

4.3.1 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)
```

4.3.2 Adding and removing data

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

```
scr_filemap_add_file(map, ckpt, rank, filename)
```

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

```
scr_filemap_remove_file(map, ckpt, rank, filename)
```

Remove all info corresponding to a given checkpoint id.

```
scr_filemap_remove_checkpoint(map, ckpt)
```

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 checkpoint number.

```
scr_filemap_remove_rank_by_checkpoint(map, ckpt, 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)
```

4.3.3 Query functions

Get the number of checkpoints in a filemap.

```
int num_ckpts = scr_filemap_num_checkpoints(map)
```

Get the most recent checkpoint (highest checkpoint id).

```
int ckpt = scr_filemap_latest_checkpoint(map)
```

Get the oldest checkpoint (lowest checkpoint id).

```
int ckpt = scr_filemap_oldest_checkpoint(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 checkpoint.

```
int num_ranks = scr_filemap_num_ranks_by_checkpoint(map, ckpt)
```

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 checkpoint id. Returns 1 if true, 0 if false.

```
scr_filemap_have_rank_by_checkpoint(map, ckpt, rank)
```

For a given rank in a given checkpoint, 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 checkpoint. 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 checkpoint.

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

```
int num_files = scr_filemap_num_files(map, ckpt, rank)
```

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

```
scr_filemap_set_expected_files(map, ckpt, rank, num_expected_files)
```

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

```
int num_expected_files = scr_filemap_num_expected_files(map, ckpt, rank)
```

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

```
scr_filemap_unset_expected_files(map, ckpt, rank)
```

4.3.4 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 checkpoint ids (ordered oldest to most recent).

```
scr_filemap_list_checkpoints(map, &num_ckpts, &ckpts)
```

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

```
scr_filemap_list_ranks(map, &num_ranks, &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 checkpoint. The list is in arbitrary order.

```
scr_filemap_list_files(map, ckpt, rank, &num_files, &files)
```

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

```
int num_ckpts;
int* ckpts;
scr_filemap_list_checkpoints(map, &num_ckpts, &ckpts)
...
if (ckpts != NULL)
  free(ckpts)
```

4.3.5 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 checkpoint ids contained in a filemap.

```
struct scr_hash_elem* elem = scr_filemap_first_checkpoint(map)
```

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

```
struct scr_hash_elem* elem = scr_filemap_first_rank_by_checkpoint(map, ckpt)
```

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

```
struct scr_hash_elem* elem = scr_filemap_first_file(map, ckpt, rank)
```

4.3.6 Checkpoint descriptors

A checkpoint descriptor is a data structure that describes the location and redundancy scheme that is applied to a set of checkpoint files in cache (Section 6). In addition to knowing what checkpoint files are in cache, it's also useful to know what redundancy scheme is applied to that checkpoint data. To do this, a checkpoint descriptor can be associated with a given checkpoint and rank in the filemap.

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

```
scr_filemap_set_desc(map, ckpt, rank, desc)
```

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

```
scr_filemap_get_desc(map, ckpt, rank, desc)
```

Unset a checkpoint descriptor.

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

4.3.7 Tags

One may also associate arbitrary key/value string pairs for a given checkpoint 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 checkpoint.

```
scr_filemap_set_tag(map, ckpt, rank, tag, value)
```

To retrieve the value associated with a tag.

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

To unset a tag value.

```
scr_filemap_unset_tag(map, ckpt, rank, tag)
```

4.3.8 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)
```

5 Meta data

5.1 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), the rank which wrote the file, and the checkpoint id that the file belongs to. It also tracks info regarding the integrity of the file, such as whether the application marked the file as valid or invalid, the expected file size, and its CRC32 checksum value, if computed.

The `scr_meta` data structure makes heavy use of the `scr_hash` data structure (Section 3). 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.

5.2 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
RANK
  0
```

The `FILE` field records the file name this meta data associates with. This 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 `RANK` field records the MPI rank of the process which wrote the file.

In this case, "`rank_0.ckpt`" was created during checkpoint id 6, and it was written by rank 0 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`.

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.

5.3 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`.

5.3.1 Allocating, freeing, merging, and clearing 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)
```

5.3.2 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_rank(meta, rank)
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)
```

If a field was already set to a value before making this call, the new value overwrites any existing value.
The following call can be used as a shortcut to set multiple fields at once.

```
scr_meta_set(meta, filename, filetype, filesize, checkpoint_id, rank, ranks, complete)
```

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_rank(meta, rank)
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)
```

Note before using the value returned by these get functions, one must first check that the specified field was actually set in the provided meta data object. The get functions return `SCR_SUCCESS` if the the field was set.

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_rank(meta, rank)
scr_meta_check_checkpoint(meta, ckpt)
scr_meta_check_filesize(meta, filesize)
scr_meta_check_filetype(meta, filetype)
scr_meta_check_filename(meta, filename)
```

Like 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)
```

5.3.3 Meta data files

The meta data is often stored in a file having the same file name as the file it is associated with, but with an added "`.scr`" extension. The following functions are used to access these type of meta data files. In these functions, the caller should provide the name of the original file (not the name of its meta data file).

To read meta data for a specified file.

```
scr_meta_read(file, meta)
```

This read does not clear any values already stored in the specified meta data object. Any values associated with file are thus merged into the provided meta data object. To write out meta data for a specified file.

```
scr_meta_write(file, meta)
```

To delete meta data file associated with a specified file.

```
scr_meta_unlink(file)
```

To get the actual name of the meta data file.

```
scr_meta_name(metadata_filename, file)
```

6 Checkpoint descriptors

6.1 Overview

A checkpoint descriptor is a data structure that describes how checkpoint data 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 checkpoint 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. 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 checkpoint descriptor structures in a global variable named `scr_ckptdescs`. It records the number of descriptors in this list in a variable named `scr_nckptdescs`. It builds this list during `SCR_Init()` using a series of checkpoint descriptor hashes defined in a third variable named `scr_ckptdesc_hash`. The hashes in this variable are constructed while processing SCR parameters.

6.2 Checkpoint descriptor struct

Here is the definition for the C struct.

```
struct scr_ckptdesc {
  int      enabled;
  int      index;
  int      interval;
  char*    base;
  char*    directory;
  int      copy_type;
  int      hop_distance;
  int      set_size;
  MPI_Comm comm;
  int      groups;
  int      group_id;
  int      ranks;
  int      my_rank;
  int      lhs_rank;
  int      lhs_rank_world;
  char     lhs_hostname[256];
  int      rhs_rank;
  int      rhs_rank_world;
  char     rhs_hostname[256];
};
```

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

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 checkpoint subdirectory is created. This path consists of the cache directory followed by the checkpoint descriptor index directory, such that one must only append the checkpoint id to compute the full path of the corresponding checkpoint 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_LOCAL`, `SCR_COPY_PARTNER`, or `SCR_COPY_XOR`. The `hop_distance` and `set_size` fields record parameters relevant to the redundancy scheme.

The remaining fields describe the group of processes that make up the redundancy set for a particular process. For a given checkpoint 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.

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.

6.3 Example checkpoint descriptor hash

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

6.4 Common functions

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

6.4.1 Initializing and freeing checkpoint descriptors

Initialize a checkpoint descriptor structure (clear its fields).

```
struct scr_ckptdesc desc;
scr_ckptdesc_init(&desc)
```

Free memory associated with a checkpoint descriptor.

```
scr_ckptdesc_free(&desc)
```

6.4.2 Checkpoint descriptor array

Allocate and fill in `scr_ckptdescs` array using checkpoint descriptor hashes provided in `scr_ckptdesc_hash`.

```
scr_ckptdesc_create_list()
```

Free the list of checkpoint descriptors.

```
scr_ckptdesc_free_list()
```

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

```
struct scr_ckptdesc* desc = scr_ckptdesc_get(ckpt, nckpts, ckpts)
```

6.4.3 Converting between structs and hashes

Convert a checkpoint descriptor struct to its equivalent hash.

```
scr_ckptdesc_store_to_hash(desc, hash)
```

This function clears any entries in the specified hash before setting fields according to the struct.
Given a checkpoint descriptor hash, build and fill in the fields for its equivalent checkpoint descriptor struct.

```
scr_ckptdesc_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.

6.4.4 Interacting with filemaps

Checkpoint descriptor hashes are cached in filemaps. There are functions to set, get, and unset a checkpoint descriptor hash in a filemap for a given checkpoint id and rank id (Section 4.3.6). There are additional functions to extract info from a checkpoint descriptor hash that is stored in a filemap.

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

```
char* basedir = scr_ckptdesc_base_from_filemap(map, ckpt, rank)
```

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

```
char* dir = scr_ckptdesc_dir_from_filemap(map, ckpt, rank)
```

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

```
scr_ckptdesc_create_from_filemap(map, ckpt, rank, desc)
```

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

7 Redundancy schemes

7.1 LOCAL

TODO

7.2 PARTNER

TODO

7.3 XOR

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

The `XOR` redundancy scheme is designed such that the checkpoint files for any single member of a set can be reconstructed using the checkpoint and `XOR` files from all remaining members. Thus, all checkpoint files can be recovered, even if the checkpoint 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 checkpoint descriptor. Each process also caches information abouts its left and right neighbor processes in the checkpoint descriptor.

Since compute node failures are a common failure type, SCR ensures that it does not select two processes from the same node to be in the same `XOR` set. It accomplishes this by selecting subsets of processes from `scr_comm_level`. The `SCR_SET_SIZE` parameter determines the maximum number of processes to include in a set. The `SCR_HOP_DISTANCE` parameter specifies the number of ranks to the left and right a process should select its left and right neighbors from `scr_comm_level`. The selection algorithm is implemented in `scr_ckptdesc_create_from_hash()` and `scr_set_partners()` in `scr.c`. Some example `XOR` sets are shown in Figure 3.

7.3.1 XOR algorithm

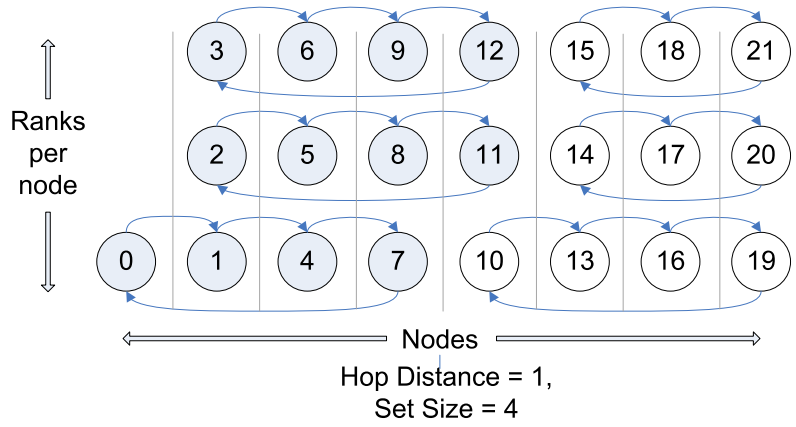
The `XOR` redundancy scheme applies the algorithm described in [1] (which is based on RAID5 [2]). Assuming that each process writes one file and that the files on all processes are the same size, this algorithm is illustrated in Figure 4. Given N processes in the set, each file is logically partitioned 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 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 5, 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.

In practice, to read from this logical file, we first open each real 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 $N-1$. At this point, we can implement the reduce-scatter algorithm to compute the data for the `XOR` chunk on each process. 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.



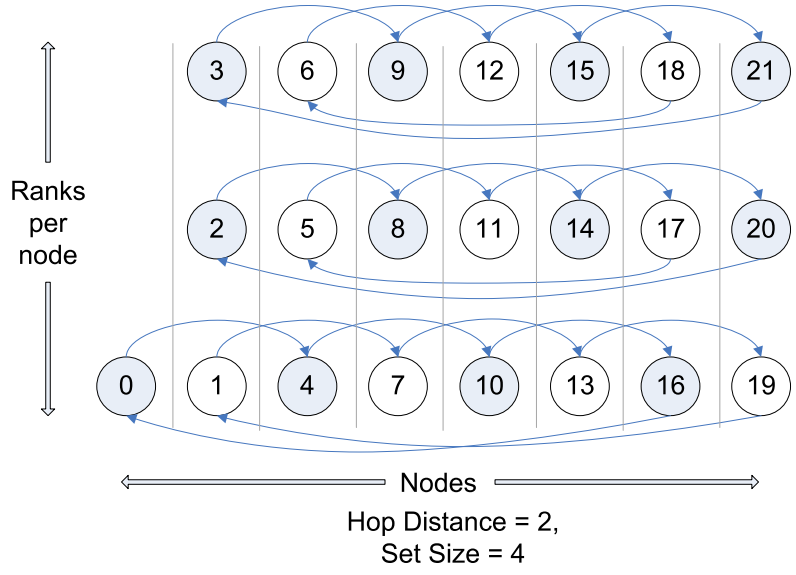


Figure 3: XOR set selection

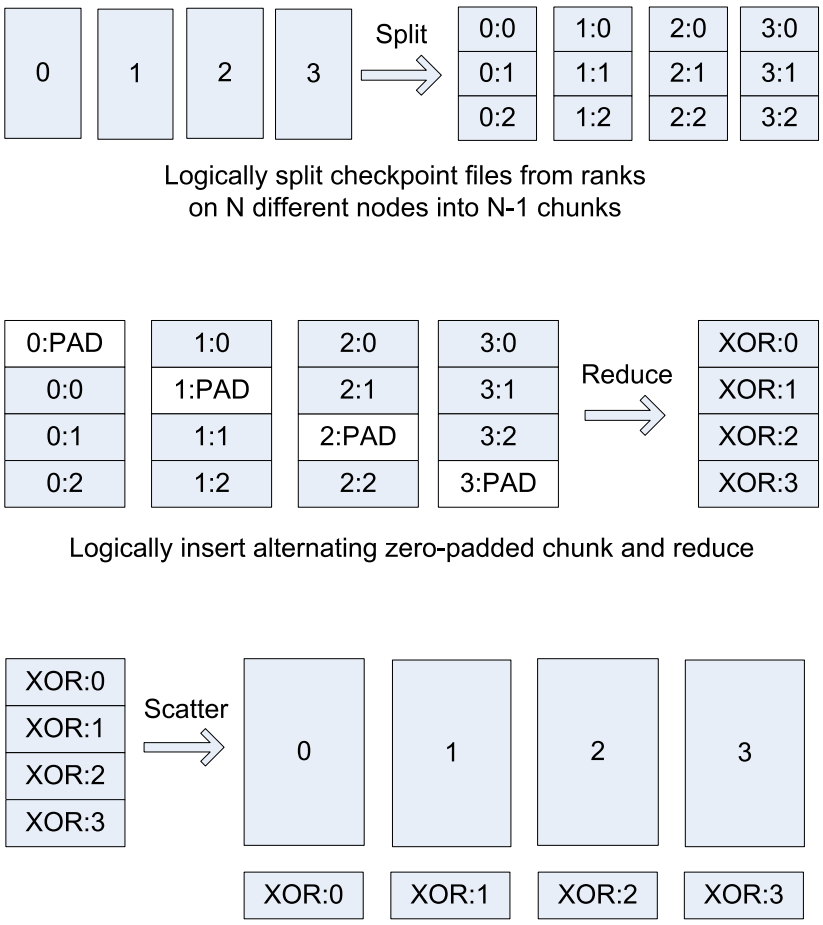


Figure 4: XOR reduce-scatter

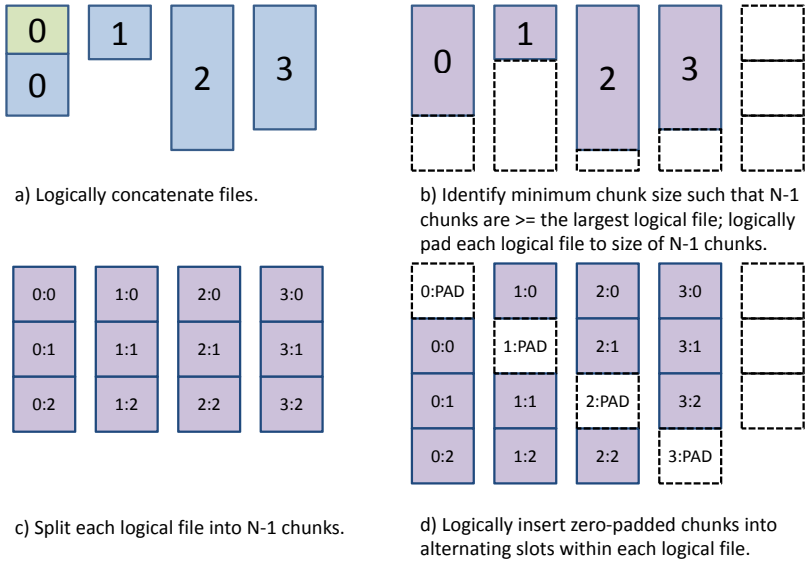


Figure 5: Extension to multiple files

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.

4. Operates on the data in small pieces, so that our 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 6.

7.3.2 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 checkpoint 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 checkpoint 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:

```
CKPT
  6
RANKS
  4
GROUP
  RANKS
```

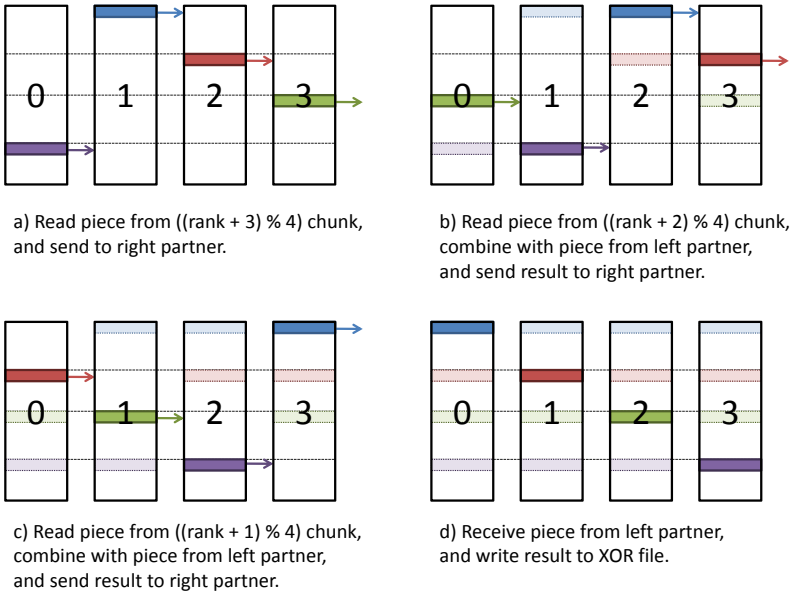


Figure 6: XOR reduce-scatter implementation

```
    4
  RANK
    0
      0
    1
      1
    2
      2
    3
      3
CHUNK
  174766
CURRENT
  RANK
    0
  FILES
    1
  FILE
    0
      FILE
        rank_0.ckpt
      TYPE
        FULL
      COMPLETE
        1
      SIZE
        524294
      CKPT
```

```
      6
    RANKS
      4
    RANK
      0
PARTNER
  RANK
    3
  FILES
    1
  FILE
    0
      FILE
        rank_3.ckpt
      TYPE
        FULL
      COMPLETE
        1
      SIZE
        524297
      CKPT
        6
      RANKS
        4
      RANK
        3
```

The topmost `CKPT` field records the checkpoint id 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`.

7.3.3 XOR rebuild

SCR provides two different methods to rebuild checkpoint files using the `XOR` scheme. If a run is restarted and a checkpoint is stored in cache, then SCR rebuilds checkpoint files during `SCR_Init()`. On the other hand, at the end of an allocation, SCR can rebuild files after draining a checkpoint from cache.

During `SCR_Init()` in a restarted run, SCR can use MPI to rebuild checkpoint 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 checkpoint 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 7.3.1. Namely, the implementation attempts to

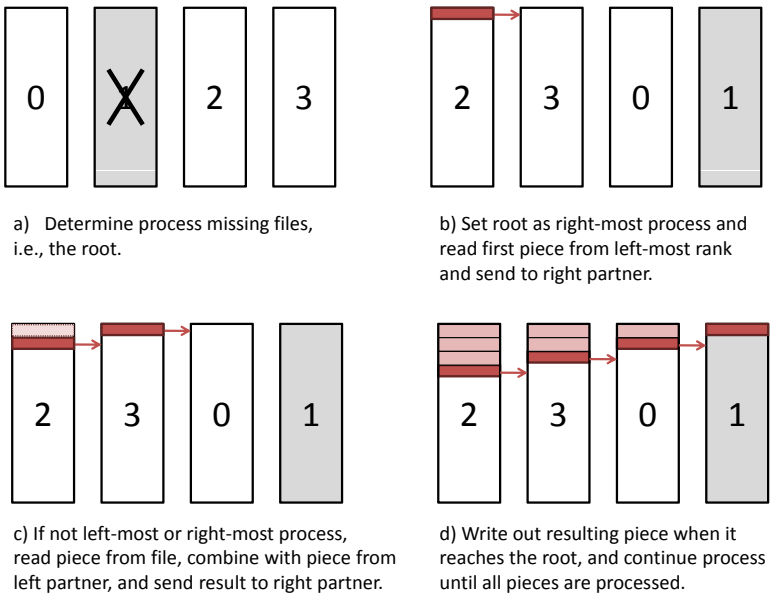


Figure 7: Pipelined XOR reduction to root

distribute work evenly among all processes, minimize network contention, and minimize file accesses. This algorithm is implemented in `scr_rebuild_xor()` in `scr.c`. An example is illustrated in Figure 7.

TODO: discuss drain rebuild

8 Drain

At the end of an allocation, a set of 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 drain this checkpoint before the allocation ends. In this section, we detail key concepts referenced as part of the drain operations. Detailed program flow for these operations is provided in Section 11.10.

8.1 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 checkpoint id from the cache to a specified checkpoint directory on the parallel file system. It is implemented in `scr_copy.c` whose program flow is described in Section 11.10.3. The `scr_copy` command copies all application files and SCR redundancy data files, as well as, their corresponding meta data files. In addition, it writes a special filemap file for each rank to the checkpoint directory. The name of this filemap file is of the format: `<rank>.scrfilemap`. An example hash for such a filemap file is shown below:

```
CKPT
  6
    RANK
      2
RANK
  2
```

```
CKPT
  6
    FILE
      3_of_4_in_0.xor
      rank_2.ckpt
    EXPECT
      2
```

It lists the files owned by a rank for a particular checkpoint. In this case, it shows that rank 2 wrote two files (EXPECT=2) as part of checkpoint 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 4. The main differences are that files are listed using relative paths instead of absolute paths and there are no checkpoint descriptors. The paths are relative so that the checkpoint directory on the parallel file system may be moved or renamed. Checkpoint descriptors are cache-specific, so these entries are excluded.

8.2 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 checkpoint to the SCR index file (Section 9.1). When invoking the `scr_index` command, the full path to the prefix directory and the name of the checkpoint 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 11.10.4.

The `scr_index` command first acquires a listing of all items contained in the checkpoint 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 checkpoint 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*. After reading all filemap files, the scan hash looks like the following.

```
CKPT
  <checkpoint_id>
    RANK
      <rank1>
        FILES
          <num_expected_files_for_rank1>
      <rank2>
        FILES
          <num_expected_files_for_rank2>
      ...
```

The `scr_index` command then iterates over all file names for a second time to read all meta data files (those files that end with the "`.scr`" extension). It verifies the information specified in each meta data file against the original file the meta data refers to (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 9.2). 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 second iteration, the scan hash looks like the following example:

```
CKPT
  <checkpoint_id>
    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>
      ...
  <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>
      ...
  ...
```

8.3 Inspecting files

After merging data from all filemap and meta data files in the checkpoint directory, the `scr_index` command inspects the scan hash to identify any missing files. For each checkpoint, it determines the number of ranks associated with the checkpoint, 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.

```
CKPT
  <checkpoint_id>
    INVALID
    MISSING
      <rank1>
      <rank2>
      ...
```

8.4 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.

```
CKPT
  <checkpoint_id>
    UNRECOVERABLE
    BUILD
```

```
      <cmd_to_rebuild_files_for_set1>
      <cmd_to_rebuild_files_for_set2>
      ...
```

8.5 Scan hash

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

```
CKPT
  <checkpoint_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 checkpoint 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.

9 Files

This section covers the contents of many of the files used in SCR. See Section 10 for an example of where these files are written.

9.1 Index file

The index file records information about each of the checkpoints 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
DIR
  scr.2011-02-28_12:37:36.1145655.18
    CKPT
      18
  scr.2011-02-28_12:37:14.1145655.18
    CKPT
      18
  scr.2011-02-28_12:37:04.1145655.12
    CKPT
      12
  scr.2011-02-28_12:37:04.1145655.10
    CKPT
      10
  scr.2011-02-28_12:36:50.1145655.6
    CKPT
      6
CKPT
  18
    DIR
      scr.2011-02-28_12:37:36.1145655.18
        COMPLETE
          1
        FLUSHED
          2011-02-28T12:37:36
      scr.2011-02-28_12:37:14.1145655.18
        FAILED
          2011-02-28T12:37:36
        FETCHED
          2011-02-28T12:37:36
        COMPLETE
          1
        FLUSHED
          2011-02-28T12:37:14
  12
    DIR
      scr.2011-02-28_12:37:04.1145655.12
        FETCHED
          2011-02-28T12:37:36
          2011-02-28T12:37:14
        COMPLETE
          1
        FLUSHED
          2011-02-28T12:37:04
```

```
10
  DIR
    scr.2011-02-28_12:37:04.1145655.10
      COMPLETE
        1
      FLUSHED
        2011-02-28T12:37:04
6
  DIR
    scr.2011-02-28_12:36:50.1145655.6
      FETCHED
        2011-02-28T12:37:04
      COMPLETE
        1
      FLUSHED
        2011-02-28T12:36:50
```

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 `DIR` hash is a simple index which maps a directory name to a checkpoint id.

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

When restarting a job, SCR starts with the checkpoint directory pointed to by the `scr.current` symlink, if one exists. 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.

9.2 Summary file

The summary file tracks which files were written by which ranks during a particular checkpoint. It is stored in the checkpoint directory within the prefix 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
  5
CKPT
  18
    COMPLETE
      1
    RANKS
      4
    RANK
      0
        FILE
          rank_0.ckpt
            SIZE
              524294
            CRC
              0x6697d4ef
      2
        FILE
```

```
      rank_2.ckpt
        SIZE
          524296
        CRC
          0xb6a62246
    3
      FILE
        rank_3.ckpt
          SIZE
            524297
          CRC
            0x213c897a
    1
      FILE
        rank_1.ckpt
          SIZE
            524295
          CRC
            0x28eeb9e
```

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.

The details of the files for this checkpoint are then contained within the CKPT hash, which lists the checkpoint id along with the details of the checkpoint. A COMPLETE flag concisely denotes whether all files for this checkpoint are thought to be valid. The number of ranks used to write this checkpoint is recorded under the RANKS field. Then, there is an entry for each rank under the RANK hash indexed by rank id. Each file that a rank wrote as part of the checkpoint is indexed by file name under the FILE hash. Here the file name specifies the relative path to the file starting from the location of the summary file. 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.

When fetching a checkpoint upon a restart, rank 0 reads the summary file and scatters its contents to the other ranks. From this information, each rank constructs the meta data for each of its checkpoint files.

9.3 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 4.2. 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 drain, 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)
}
```

9.4 Flush file

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

```
CKPT
  18
    LOCATION
      PFS
      CACHE
  17
    LOCATION
      CACHE
```

Each checkpoint is indexed by checkpoint id under the `CKPT` hash. Then, under the `LOCATION` hash, different flags are set to indicate where that checkpoint is stored. The `PFS` flag indicates that a copy of this checkpoint is stored on the parallel file system, while the `CACHE` flag indicates that the checkpoint is stored in cache. The same checkpoint may be stored in multiple locations at the same time. At the end of a run, the flush and drain 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.

9.5 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 control 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 checkpoint, 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.

9.6 Nodes file

The nodes file is kept in the control 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.

9.7 Transfer file

When using the asynchronous flush, the library creates a checkpoint 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/checkpoint.1/rank_0.ckpt
    DESTINATION
      /p/lscratchb/user1/simulation123/scr.2010-06-02_17:35:13.1001186.1/rank_0.ckpt
    SIZE
      524294
    WRITTEN
      524294
  /tmp/user1/scr.1001186/index.0/checkpoint.1/rank_0.ckpt.scr
    DESTINATION
      /p/lscratchb/user1/simulation123/scr.2010-06-02_17:35:13.1001186.1/rank_0.ckpt.scr
    SIZE
      124
    WRITTEN
      124
PERCENT
  0.000000
BW
  52428800.000000
COMMAND
  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 checkpoint directory and updating the index file in the prefix directory.

10 Example of SCR files and directories

To illustrate how various files and directories are arranged in SCR, consider the example shown in Figure 8. In this example, someone whose user name on the system is "`user1`" is running 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`.

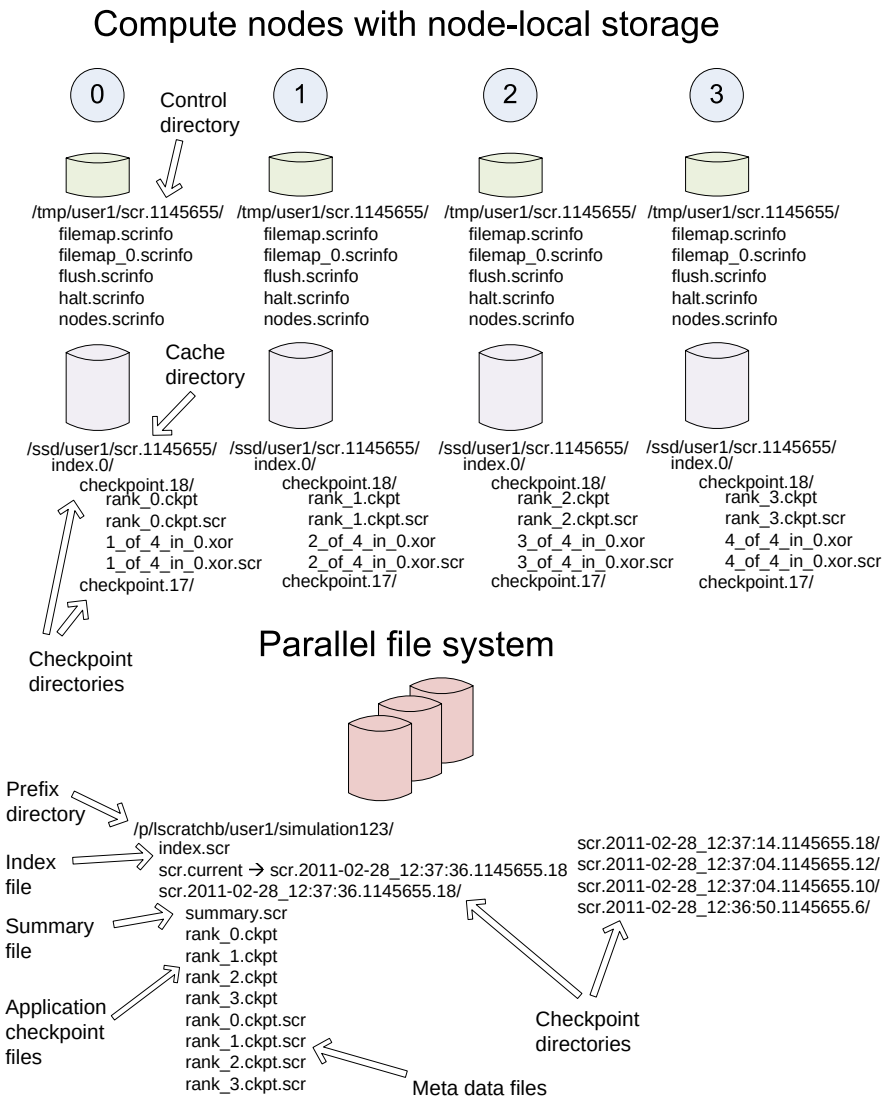


Figure 8: Example of SCR Directories

Note that 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 9.3), the flush file (Section 9.4), the halt file (Section 9.5), the nodes file (Section 9.6), and the transfer file (Section 9.7).

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 checkpoint. The checkpoint directory name incorporates the index of the checkpoint descriptor associated with the checkpoint (in this case, index `0`), along with the checkpoint id (`17` or `18`). In this example, there are two checkpoints currently stored in cache, and both are associated with the checkpoint descriptor having index `0` (`index.0/checkpoint.17` and `index.0/checkpoint.18`). The application checkpoint files, SCR redundancy data files, and meta data files (Section 5) are stored within their corresponding checkpoint directory. In this example, on the node where MPI rank 0 is running, there is one application checkpoint file (`rank_0.ckpt`) and one XOR redundancy data file (`1_of_4_in_0.xor`), along with a meta data file for each that has the same name but an added "`.scr`" extension.

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`). This directory contains a subdirectory for each checkpoint along with the index file (`index.scr`) that SCR uses to record info for each of the checkpoints (Section 9.1). SCR also maintains the "`scr.current`" symlink to point to the checkpoint currently marked as the most recent. The user may adjust this symlink to force a job to restart from a different checkpoint.

While the user provides the full path for the prefix directory, SCR defines the name of each checkpoint directory. In order to avoid creating conflicting checkpoint directory names, SCR specifies the timestamp at which the checkpoint was written to the prefix directory, the allocation id of the resource allocation in which it was created, and the checkpoint id associated with the checkpoint. In this example, there are multiple checkpoints stored on the parallel file system (corresponding to checkpoint ids 6, 10, 12, and 18) all written at different times during the same resource allocation (`1145655`).

Note that it is possible for the same checkpoint id to be written multiple times. For example there are two copies of checkpoint id 18 (`scr.2011-02-28_12:37:36.1145655.18` and `scr.2011-02-28_12:37:14.1145655.18`). This may occur when a job is restarted from an earlier checkpoint, perhaps because the first copy of checkpoint id 18 was corrupted or incomplete.

Within each checkpoint directory, SCR stores application checkpoint files and meta data files associated with the checkpoint. SCR may also store redundancy data files and filemap files here. Additionally, SCR creates a summary file (`summary.scr`) in which it records information needed to fetch files from the directory in order to restart the job from the checkpoint (Section 9.2). A checkpoint directory may not contain a complete and valid set of checkpoint data. This may happen if SCR failed to copy all checkpoint data from cache to the parallel file system, or it can happen if the data have been corrupted while being stored on the parallel file system. This condition may be detected while fetching the checkpoint, and it is then noted in the index file. This way, SCR will know not to attempt to fetch this checkpoint again in later runs, and it will instead attempt to fetch the next most recent checkpoint.

11 Program flow

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

11.1 Perl modules

11.1.1 `scripts/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:

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

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

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);
```

11.1.2 scripts/scr_param.pm

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).

The full path to a user configuration file is provided when an instance of the object is initialized. If no path is provided, it's assumed that a user config file does not exist.

Some parameters cannot be set by a user, and for these parameters any settings in #1 or #2 is ignored.

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

Allocate a new `scr_param` object and optionally specify the path to a user configuration file.

```
my $param = new scr_param("/path/to/user/conf.file");
```

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);
```

11.2 Utilities

11.2.1 scripts/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.

```
numhosts= 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.

```
thirdhost= 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.

```
diff= 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.

```
intersection= 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.

11.2.2 src/scr_flush_file.c

Utility to access info in SCR flush file.

Read the flush file and return the latest checkpoint id.

```
latest_ckpt= scr_flush_file --dir /tmp/user1/scr.856 --latest
```

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.

```
need_flush= scr_flush_file --dir /tmp/user1/scr.856 --id 6
```

The command above opens the flush file, and 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.

11.2.3 scripts/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. Interprets $SCR_CONF_FILE to look for configuration file when initializing scr_param object.
2. Invokes "scr_env --nodes" to get the current nodeset, if not specified on command line.
3. Invokes "scr_env --down" to ask resource manager whether any nodes are known to be down.
4. Invokes ping to each node thought to be up.
5. Uses scr_param.pm to read SCR_EXCLUDE_NODES, user may explicitly exclude nodes this way.
6. Adds any nodes explicitly listed on command line as being down.

7. Invokes `scr_cntl_dir` to get list of base directories for control directory.
8. Uses `scr_param.pm` to read `CNTLDIR` hash from config file to get expected capacity corresponding to each base directory.
9. Invokes `scr_cntl_dir` to get list of base directories for cache directory.
10. Uses `scr_param.pm` to read `CACHEDIR` hash from config file to get expected capacity corresponding to each base directory.
11. Invokes `pdsh` to run `scr_check_node` on each node that hasn't yet failed a test.
12. Optionally print list of down nodes to stdout.
13. Optionally log each down node with reason via `scr_log_event` if logging is enabled.
14. Exit with appropriate code to indicate whether any nodes are down.

11.2.4 scripts/scr_cntl_dir.in

Returns full path to control or cache directory. Uses `scr_param.pm`, and interprets `$SCR_CONF_FILE` to look for user configuration file when initializing `scr_param` object. Thus, this command should be executed in an environment where this variable is set to the same value as in 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 `CACHEDESC` hash from config file to get info on checkpoint descriptors.
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.

11.3 Launching (and relaunching) a run

11.3.1 scripts/scr_srun.in

Prepares a resource allocation for SCR, launches a run (and continues to relaunch the run in the case of a failure), drains and rebuilds files for most recent checkpoint if needed, updates SCR index file in prefix directory to account for this last checkpoint.

1. Interprets `$SCR_ENABLE`, bails with success if set to 0.
2. Interprets `$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. Interprets `$SCR_PREFIX` to get prefix directory on parallel file system. sets and exports `$SCR_PREFIX` to `pwd` if not set.
7. Invokes `scr_glob_hosts` to check that this command is running on a node in the nodeset, bails with error if not.
8. Invokes `scr_cntl_dir` to get control directory.

9. Issues a NOP `srun` command on all nodes to force each node to run SLURM prologue to delete old files from cache.
10. Invokes `scr_prerun` to prepare nodes for SCR run.
11. 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. Count the number of nodes that the application needs.
3. Count the number of nodes that are still left in the resource allocation.
4. If number of nodes left is smaller than number needed, break loop.
5. Check that the node where the `scr_srun` command is running is still good, if not, break loop.
6. Build list of nodes to be excluded from run.
7. Optionally log start of run.
8. Invokes `srun` including node where the `scr_srun` command is running and excluding down nodes.
9. Invokes `scr_list_down_nodes` to get list of down nodes.
10. Optionally log end of run (and down nodes and reason those nodes are down).
11. If number of attempted runs is >= than number of allowed retries, break loop.
12. Invokes `scr_retries_halt` and breaks loop if halt condition is detected.
13. Invokes "`sleep 60`" to give nodes in allocation a chance to cleanup.
14. 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.
15. 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 drain the most recent checkpoint.

11.3.2 `scripts/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.

11.3.3 scripts/scr_prerun.in

Currently does nothing, just serves as a place holder.

11.3.4 src/scr_retries_halt.c

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

11.4 SCR_Init

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

1. If not enabled, if so bail out with error.
2. Create `scr_comm_world` by duplicating `MPI_COMM_WORLD`.
3. Get hostname, store in `scr_my_hostname`.
4. Get memory page size, store in `scr_page_size`.
5. Initialize parameters – rank 0 reads any config files and broadcasts info to other ranks.
6. Check whether we are still enabled (a config file may disable us), and bail out with error if not.
7. Check that `scr_username` and `scr_jobid` are defined, which are used for logging purposes.
8. Create `scr_comm_local` and `scr_comm_level`.
9. Setup our list of checkpoint descriptors in `scr_ckptdescs`.
10. Check that we have a valid descriptor that we can use for each checkpoint.
11. Log the start of this run, if logging is enabled.
12. Create the control directory.
13. 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. Check that prefix directory is specified if we need to fetch or flush.
4. Master process reads and distributes info from filemaps on this node (Section 11.4.1).
5. Execute rebuild loop to get the most recent checkpoint from cache (Section 11.4.2).

6. If we still don't have a checkpoint (rebuild loop failed), clear the cache (delete all files) and execute fetch loop to try to read a checkpoint from the parallel file system (Section 11.4.7).

7. If the fetch loop also 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.

11.4.1 scr_scatter_filemaps

During a restart, the master process on each node reads in all of the filemap data and distributes this data to the other processes on the node, if any. After this distribution phase, a process is responsible for each file it has filemap data for, and each file in cache will be the responsibility of some process. We use this approach to handle cases where the number of tasks running on the node in the current run is different from the number of tasks that ran on the same node in the prior run. This function is implemented in `scr.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 on the node to master process.

4. If the filemap has data for a rank on our node, master prepares hash to send corresponding data to that rank.

5. Master evenly distributes the remainder of the filemap data to all processes on the node.

6. Distribute filemap data via `scr_hash_exchange()`.

7. Master writes new master filemap file.

8. Each process writes new filemap file.

11.4.2 Rebuild loop

This section describes the loop used to distribute and rebuild checkpoint files in cache. In short, SCR attempts to restart from the most recent checkpoint, which corresponds to the checkpoint having the highest checkpoint id. SCR enters a loop in which it finds the highest checkpoint id among all processes in the run and attempts to rebuild files for that checkpoint. If the rebuild succeeds, it exits the loop and uses that checkpoint. Otherwise, it deletes that checkpoint id from all processes and continues the loop trying the next highest checkpoint id. This functionality is implemented within `SCR_Init()`.

1. Start timer.

LOOP

1. Inspect and delete any bad files from cache.

2. Identify most recent checkpoint id across all processes (max of highest checkpoint id).

3. If there is no checkpoint id specified on any process, break loop.

4. Otherwise, log which checkpoint we are attempting to rebuild.

5. Distribute checkpoint descriptors for this checkpoint and store in temporary checkpoint descriptor object. This informs each process about the cache device and the redundancy scheme to use for this checkpoint.

6. If we fail to distribute the checkpoint descriptors to all processes, delete this checkpoint from cache and loop.
7. Create directory in cache for this checkpoint according to checkpoint descriptor.
8. Distribute files to the ranks that wrote them (Section 11.4.3). The owner ranks may now be on different nodes.
9. Rebuild any missing files for this checkpoint using redundancy scheme specified in checkpoint descriptor (Section 11.4.4).
10. If the rebuild fails, delete this checkpoint from cache and loop.
11. Otherwise, the rebuild succeeded. Set `scr_checkpoint_id` to the checkpoint id of this checkpoint, so that we continue counting up from this number when assigning ids to later checkpoints.
12. Update our flush file to note that this checkpoint is in cache. If any process has marked this checkpoint as flushed, mark it as flushed on all processes. Remove FLUSHING flag from flush file.
13. Free the temporary checkpoint descriptor for this checkpoint.
14. Log whether or not the rebuild succeeded.

EXIT LOOP

1. If we successfully rebuilt a checkpoint, delete all other checkpoints from cache.
2. Stop timer and log whether we were able to rebuild a checkpoint from cache.
3. Flush this checkpoint to the parallel file system if we need to.

11.4.3 scr_distribute_files

This section describes the algorithm used to distribute files for a specified checkpoint. Essentially this transfers files from their current location to the node-local storage device on 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 just 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 checkpoint descriptor, and a checkpoint id as input. This implementation is in `scr_distribute_files()` in `scr.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 checkpoint.
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 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. Specify 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 checkpoint.
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.
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.

EXIT LOOP EXCHANGE

1. Free list of files that we sent in this round.
2. Remove rank we sent files to from our filemap (no need to delete its files as they were deleted during the exchange loop).

3. Write out our updated filemap to record that the files for this rank are now gone.

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.

11.4.4 scr_rebuild_files

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

1. Attempt to rebuild files according to the redundancy scheme specified in the checkpoint descriptor. Currently, only `XOR` can actually rebuild files (Section 11.4.5).
2. If the rebuild failed, return with an error.
3. Otherwise, check that all processes have all of their files for this checkpoint.
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.

11.4.5 scr_attempt_rebuild_xor

Before we attempt to rebuild files using the `XOR` redundancy scheme, we first check whether it's even possible. If we detect that two or more processes from the same `XOR` set are missing files, we will not be able to recover all files, so there's no point to rebuild any of them. We execute this check in `scr_attempt_rebuild_xor()` in `scr.c`. The caller specifies a filemap, a checkpoint descriptor, and a checkpoint id as input.

1. Check whether we have our checkpoint 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_rebuild_xor()` to rebuild files (Section 11.4.6).
5. Check that the rebuild succeeded on all tasks, return error if not, otherwise return success.

11.4.6 scr_rebuild_xor

If there's a chance to rebuild all files, we invoke `scr_rebuild_xor()` within each set that is missing files. The caller specifies a filemap, a checkpoint descriptor, and a checkpoint 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.c`

ALL

1. Allocate empty hash to hold the header of our `XOR` file.

NON-ROOT

1. Get name of our `XOR` file.
2. Open `XOR` file.
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 checkpoint 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 checkpoint.
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 checkpoint 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 7. For a full description of the redundancy scheme, see Section 7.3.1.

4. Close our `XOR` file.
5. Close each of our checkpoint files.

ROOT

1. For each of our checkpoint files and our `XOR` file, write out our meta data file.
2. 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.

11.4.7 Fetch loop

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 (or by an "`scr.current`" symlink). If SCR fails to fetch this checkpoint, it then works backwords 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_Init()`.

1. Start timer.
2. Build full path to checkpoint directory pointed to by "`scr.current`" symlink.
3. Rank 0 reads index file from prefix directory.

LOOP

1. Rank 0 selects a target directory name. Start with directory pointed to by `scr.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 but before the current directory.
2. Use data in index file to lookup checkpoint id corresponding to directory name.
3. Attempt to fetch checkpoint from selected directory.
4. If fetch fails, rank 0 deletes "`scr.current`" symlink and mark the attempted directory as failed in the index file.
5. If fetch succeeds, rank 0 updates "`scr.current`" symlink to point to this directory, break loop.

EXIT LOOP

1. Stop timer and print statistics.

11.5 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 phase time to the total compute phase time is below a maximum threshold.
6. Rank 0 broadcasts the decision to all other tasks.

11.6 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 our internal `scr_checkpoint_id`.
3. Get checkpoint descriptor for this checkpoint id.
4. Start timer for checkpoint phase, and log start of checkpoint.
5. Get a list of all checkpoints in cache.
6. Determine how many checkpoints are currently stored in the base cache directory specified by the checkpoint descriptor.
7. Delete oldest checkpoints from this base 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 (asynchronously), wait for it to complete then delete it.
8. Free the list of checkpoints.
9. Record the checkpoint descriptor in the filemap.
10. Create the directory in cache.

11.7 SCR_Route_file

Given a name of a checkpoint 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. Direct path to appropriate cache directory based on current checkpoint id.
4. If called from within a Start/Complete pair, add file name to filemap.
5. Otherwise, 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.
6. Return success.

11.8 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. Write out meta data for each checkpoint file registered in filemap for this checkpoint id.
5. Apply redundancy scheme specified in checkpoint descriptor (Section 11.8.1 or Section 11.8.2).
6. Stop timer measuring length of checkpoint, and log cost of checkpoint.
7. If checkpoint was successful, update our flush file, check whether we need to halt, and check whether we need to flush.
8. If checkpoint was not successful, delete it from cache.
9. Check whether any ongoing asynchronous flush has completed.

BARRIER

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

11.8.1 scr_copy_partner

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

1. Read list of files for this rank for the specified checkpoint.
2. Inform our right-hand partner how many files we'll send to him.
3. Record number of files we expect to receive from our left-hand partner in our filemap.
4. Remember the node name where our left-hand partner is running (used during drain).

5. Record the checkpoint descriptor hash for our left-hand partner. Each process needs to be able to recover its own checkpoint descriptor hash after a failure, so we make a copy in our partner's the filemap.
6. Write out our filemap (includes expected number of files and checkpoint descriptor for left-hand partner).
7. Get checkpoint directory we'll copy partner's files to.
8. 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()`.

EXIT LOOP

1. Free the list of file names.

11.8.2 scr_copy_xor

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

1. Allocate a buffers to send and receive data.
2. Count the number of files this process wrote during the specified checkpoint id. Allocate space to record a file descriptor, the file name, and the size of each file.
3. Record the checkpoint descriptor hash for our left-hand partner in our filemap. Each process needs to be able to recover its own checkpoint descriptor hash after a failure, so each process sends a copy to his right-hand partner.
4. Allocate a hash to hold the header of our `XOR` redundancy data file.
5. Initialize the header by calling `scr_copy_xor_header_set_ranks()`. This function records the MPI ranks that are in our `XOR` set.
6. Record the checkpoint id in our header.
7. Open each of our files, get the size of each file, and read the meta data for each file.
8. Create a hash and record our rank, the number of files we have, and the meta data for each file.
9. Send this hash to our right-hand partner, and receive equivalent hash from left-hand partner.
10. Record our hash along with the hash from our left-hand partner in our `XOR` header hash. This way, the meta data for each checkpoint file is recorded in the headers of two different `XOR` files.
11. Determine chunk size for RAID algorithm (Section 7.3.1) and record this size in the `XOR` header.
12. Determine full path name for `XOR` file.
13. Record `XOR` file name in our filemap and update the filemap on disk.
14. Open the `XOR` file for writing.

15. Write header to file and delete header hash.
16. Execute RAID algorithm and write data to `XOR` file (Section 7.3.1).
17. Close our `XOR` file (with `fsync()`) and close each of our checkpoint files.
18. Free off scratch space memory and MPI buffers.
19. Write out meta data file for `XOR` file.
20. If `SCR_CRC_ON_COPY` is specified, compute CRC32 checksum of `XOR` file.

11.9 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.

11.10 Drain

SCR includes a set of commands which are executed after the final run of the application in a resource allocation to check whether the most recently cached checkpoint was successfully copied to the parallel file system before exiting the resource allocation. This logic is encapsulated in the `scr_postrun` command.

11.10.1 scripts/scr_postrun.in

Checks whether there is a checkpoint in cache that must be copied to the parallel file system. If so, drain this checkpoint, and then rebuild any missing files if possible, 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. Interprets `$SCR_PREFIX` to determine prefix directory on parallel file system (but this value is overridden via "-p" option when called from `scr_srun`).
4. Interprets `$SCR_NODELIST` to determine set of nodes job is using.
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_cntl_dir` to get the control directory.
8. Invokes "`scr_flush_file --dir $cntldir --latest`" providing control directory to determine id of most recent checkpoint.
9. If this command fails, there is no checkpoint to flush, so bail out with error.
10. Invokes "`scr_flush_file --dir $cntldir --needflush $id`" providing control directory and checkpoint id to determine whether this checkpoint needs to be flushed
11. If this command fails, the checkpoint has already been flushed, so bail out with success.
12. Creates checkpoint directory on parallel file system for latest checkpoint.
13. Invokes `scr_flush` providing control directory, checkpoint id to be flushed, checkpoint directory, and set of known down nodes, which flushes checkpoint files from cache to the PFS.
14. Invokes `scr_index` providing checkpoint directory, which checks whether all files are accounted for, attempts to rebuild missing files if it can, and records the checkpoint directory and status in the SCR index file.
15. If checkpoint was flushed and indexed successfully, the "`scr.current`" symlink is updated to point to this checkpoint.

11.10.2 `scripts/scr_flush.in`

Manages drain of checkpoint files from cache to parallel file system. Uses `scr_hostlist.pm`. Uses `scr_param.pm`, and interprets `$SCR_CONF_FILE` to look for configuration file when initializing `scr_param` object.

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 drain).
3. Invokes "`scr_env --jobid`" to get jobid.
4. Invokes "`scr_env --nodes`" to get the current nodeset, can override with "`--jobset`" on command line.
5. Logs start of drain operation, if logging is enabled.

START ROUND 1

1. Invokes `pdsh` of `scr_copy` providing control directory, checkpoint id to be drained, checkpoint directory, buffer size, CRC32 flag, partner 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 drain, if logging is enabled.

11.10.3 src/scr_copy.c

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

1. Read control directory, checkpoint id, destination checkpoint directory, etc from command line.
2. Read "`flush.scrinfo`" file from control directory.
3. If specified checkpoint id does not exist, we can't flush it so bail out with error.
4. If specified checkpoint id has already been flushed, we're done so bail out with success.
5. Read "`filemap.scrinfo`" file (master filemap) and then read each filemap file to get list of all files on the node.
6. For each file belonging to the specified checkpoint, determine whether it is a partner copy, and if so, print out the node name it is the partner for.
7. If the partner flag is set on the command line (meaning the original node failed to copy the file), this node needs to copy the file instead since it is serving as the partner to the failed node.
8. Check that we have each file for the specified checkpoint (executes a number of validity checks on each file).
9. Copy each file to the checkpoint directory (optionally compute CRC32 during copy).
10. Copy each meta data file to the checkpoint directory (after updating CRC32 value, if it was computed).
11. Write filemap for the rank to the checkpoint directory (Section 8).

11.10.4 src/scr_index.c

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

1. If "`--add`" option is specified, verify that prefix directory and name of checkpoint directory are specified, and then call `index_add_dir` (Section 11.10.5) to add directory to index file.
2. If "`--list`" option is specified, call `index_list` to list contents of index file.

11.10.5 index_add_dir

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

1. Attempt to lookup named checkpoint directory in index file, if it's already indexed, bail out with success.
2. Concatenate checkpoint directory name with prefix directory to get path to checkpoint directory.
3. Attempt to read summary file from checkpoint directory. Call `scr_summary_build` (Section 11.10.6) if it does not exist.
4. Read checkpoint id from summary file, if this fails exit with error.
5. Read completeness flag from summary file.
6. Write entry to index hash for this checkpoint, including directory name, checkpoint id, complete flag, and flush timestamp.
7. Write hash out as new index file.

11.10.6 scr_summary_build

Scans all files in checkpoint 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 11.10.7) to read meta data for all files in directory. This records all data in a scan hash.
3. Call `scr_inspect_scan` (Section 11.10.8) 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 11.10.9) to attempt to rebuild files. After the rebuild, we delete the scan hash, rescan, and reinspect to produce an updated scan hash.
5. Delete extraneous entries from scan hash to form our summary file hash (Section 9.2).
6. Write out summary file.

11.10.7 scr_scan_files

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

1. Call `scr_read_dir` to get listing of all directories and files in checkpoint directory.
2. Iterate over list of files, and read in each filemap file (those ending with "`.scrfilemap`" extension). Extract number of expected files and record in scan hash.
3. Iterate over list of files again to read in meta data files.

BEGIN LOOP

1. Skip to next file if file name does not end with with "`.scr`" extension.
2. Skip to next file if file is the summary file (want meta data files).
3. Read meta data file.
4. Read checkpoint id, rank id, number of ranks, file name, and file size from meta data.
5. Use these values to set current checkpoint id and number of ranks if we haven't already set them.
6. Build full path to file named in meta data.
7. Check that the full path matches file named in meta data.
8. Check that file is recorded as being complete in meta data.
9. Check that file exists.
10. Check that the file size matches value recorded in meta data.
11. Check that the checkpoint id and number of ranks match values in meta data.
12. If any check fails, skip to next file.
13. Otherwise, add entry for this file in scan hash.
14. If meta data is for an `XOR` file, add an `XOR` entry in scan hash.

END LOOP

11.10.8 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 checkpoint id in scan hash, get checkpoint id and pointer to hash entries for this checkpoint.
2. Lookup hash for `RANKS` key, and check that we have exactly one entry.
3. Read number of ranks for this checkpoint.
4. Sort entries for ranks in scan hash by rank id.
5. Set expected rank id 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, throw an error and mark checkpoint 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 checkpoint as invalid.
10. Increment expected rank id.

END LOOP

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

11.10.9 scr_rebuild_scan

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

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

BEGIN LOOP

1. Get set id and number of members for this `XOR` 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 checkpoint 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 checkpoint is recoverable, fork and exec processes to rebuild missing files (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 checkpoint id in the scan hash, the checkpoint is not explicitly marked as bad, and all files already existed or we were able to rebuild all missing files.

References

[1] 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.

[2] 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.

[3] M. Jette, "Simple Linux Utility for Resource Management (SLURM)." https://computing.llnl.gov/linux/slurm.

[4] N. H. Vaidya, "A case for two-level recovery schemes," *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 656–666, 1998.