SCR Documentation

Release 1.2.0

SCR

Apr 16, 2021

Contents

The Scalable Checkpoint / Restart (SCR) library enables MPI applications to utilize distributed storage on Linux clusters to attain high file I/O bandwidth for checkpointing, restarting, and writing large datasets. With SCR, jobs run more efficiently, recompute less work upon a failure, and reduce load on shared resources like the parallel file system. It provides the most benefit to large-scale jobs that write large datasets. Check out our video on how SCR works for more information.

SCR provides the following capabilities:

- scalable checkpoint, restart, and output bandwidth,
- asynchronous data transfers to the parallel file system,
- guidance for the optimal checkpoint frequency,
- automated tracking and restart from the most recent checkpoint,
- automated job relaunch within an allocation after hangs or failures.

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

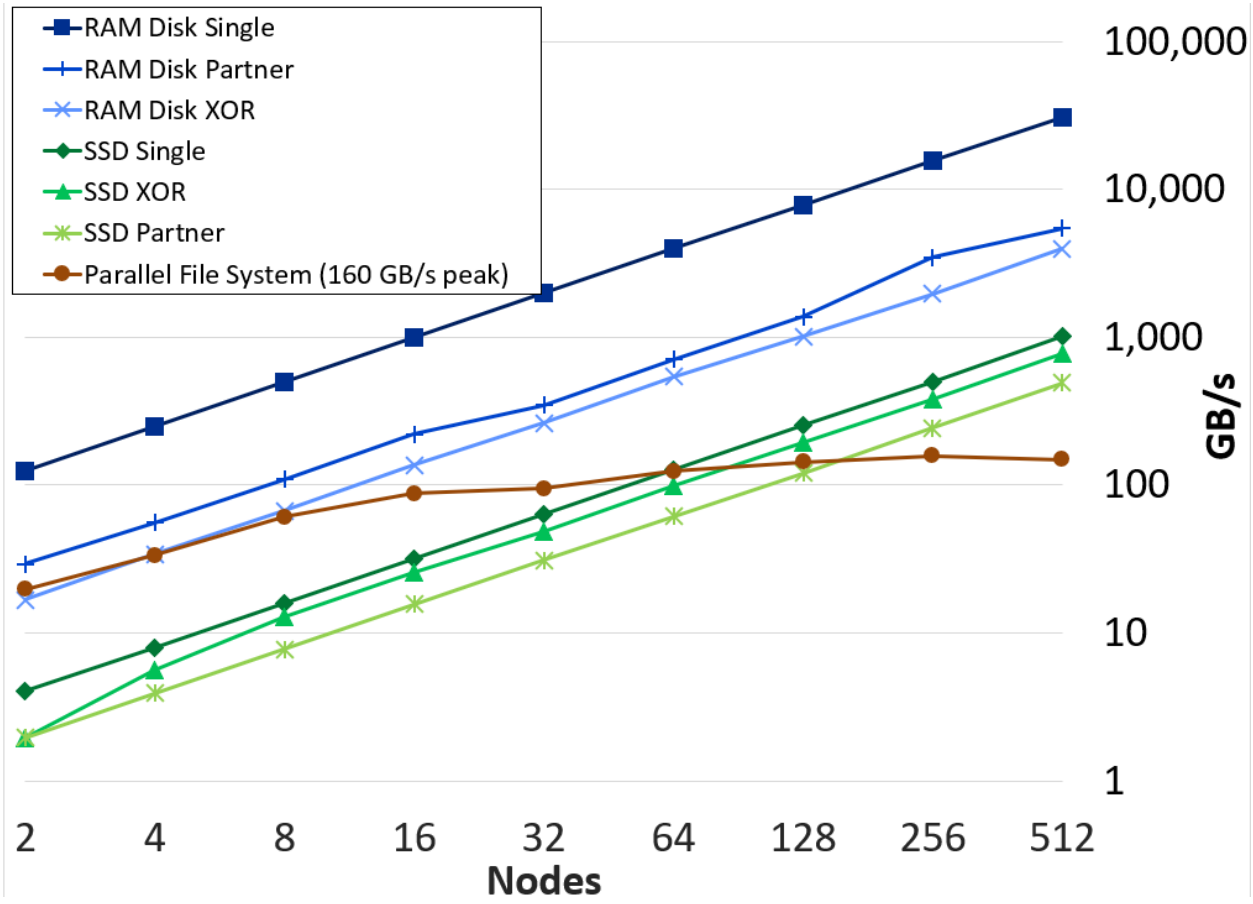

Fig. 1: Aggregate write bandwidth on Lassen

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

background. SCR supports general output datasets in addition to checkpoint datasets.

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

CHAPTER 1

Support and Additional Information

The main repository for SCR is located at:

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

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

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

- https://computing.llnl.gov/projects/scalable-checkpoint-restart-for-mpi
- https://insidehpc.com/2019/12/podcast-scr-scalable-checkpoint-restart-paves-the-way-for-exascale/
- https://www.youtube.com/watch?v=qt2VgIZaoNA
- https://youtu.be/_r6svl_eAns

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

https://computing.llnl.gov/projects/scalable-checkpoint-restart-for-mpi/contact

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

CHAPTER 2

Contents

2.1 Quick Start

In this quick start guide, we assume that you already have a basic understanding of SCR and how it works on HPC systems. We also assume you have access to a SLURM cluster with a few compute nodes and a working MPI environment. We walk through a bare bones example to quickly get you started with SCR. For more in-depth information, see subsequent sections in this user's guide.

2.1.1 Building SCR

SCR has a number of dependencies. To simplify the install process, one can use our `bootstrap.sh` script with CMake or use Spack. For full details on building SCR, please see Section *Build SCR*.

CMake

SCR requires CMake version 2.8 or higher. The SCR build uses the CMake FindMPI module to link with MPI. This module looks for the standard `mpicc` compiler wrapper, which must be in your `PATH`.

The quick version of building SCR with CMake is:

```
git clone git@github.com:llnl/scr.git
cd scr
git checkout v3.0rc1

./bootstrap.sh

mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=../install ..
make install
```

There are a number of CMake options to configure the build. For more details, see Section *CMake*.

Spack

If you use the Spack package manager, SCR and many of its dependencies have corresponding packages.

Before installing SCR with Spack, one should first configure `packages.yaml`. In particular, SCR depends on the system resource manager and MPI library, and one should define entries for those in `packages.yaml`. Examples for configuring common resource managers and MPI libraries are listed in Section *Spack*.

SCR can then be installed for SLURM systems with:

```
spack install scr@3.0rc1
```

This downloads, builds, and installs SCR and its dependencies.

2.1.2 Building the SCR `test_api` Example

In this quick start guide, we use the `test_api.c` program.

If you installed SCR with CMake, `test_api.c` was already compiled as part of the make install step above. You just need to change into the `examples` subdirectory within the current CMake `build` directory:

```
cd examples
```

Then skip to the next section to run `test_api.c`.

If you installed SCR with Spack, you will find example programs in the `<install>/share/scr/examples` directory, where `<install>` is the path in which SCR was installed. You can find Spack's SCR install directory using the following command:

```
spack location -i scr
```

Then build `test_api.c` by executing:

```
cd <install>/share/scr/examples
make test_api
```

Upon a successful build, you will have a `test_api` executable.

2.1.3 Running the SCR `test_api` Example

A quick test of your SCR installation can be done by running `test_api` in an interactive job allocation. The following assumes you are running on a SLURM-based system. If you are not using SLURM, then modify the node allocation and run commands as appropriate for your resource manager.

First, obtain compute nodes for testing. Here we allocate 4 nodes:

```
salloc -N 4
```

Once you have the compute nodes you can run `test_api`. Here we execute a 4-process run on 4 nodes:

```
srun -n 4 -N 4 ./test_api
```

This example program writes 6 checkpoints using SCR. Assuming all goes well, you should see output similar to the following

```
>>: srun -n 4 -N 4 ./test_api
Init: Min 0.033856 s    Max 0.033857 s  Avg 0.033856 s
No checkpoint to restart from
At least one rank (perhaps all) did not find its checkpoint
Completed checkpoint 1.
Completed checkpoint 2.
Completed checkpoint 3.
Completed checkpoint 4.
Completed checkpoint 5.
Completed checkpoint 6.
FileIO: Min   52.38 MB/s        Max   52.39 MB/s        Avg   52.39 MB/s       Agg
↪209.55 MB/s
```

If you do not see output similar to this, there may be a problem with your environment or your build of SCR. Please see the detailed sections of this user guide for more help or email us (see *Support and Additional Information*.)

One can use `test_api` to conduct more interesting tests. In the SCR source directory, the `testing` directory includes scripts to demonstrate different aspects of SCR. Depending on your shell preference, `TESTING.csh` or `TESTING.sh` are good for getting started. Each script contains a sequence of additional configurations and commands for running `test_api`. One can find those `TESTING` scripts in a clone of the repo, e.g.:

```
git clone git@github.com:llnl/scr.git
cd scr/testing
```

2.1.4 Adding SCR to Your Application

Here we provide an example of integrating the SCR API into an application to write checkpoints.

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

  /* Call SCR_Init after MPI_Init */
  SCR_Init();

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

    /* Ask SCR if a checkpoint should be saved (optional) */
    int need_ckpt;
    SCR_Need_checkpoint(&need_ckpt);
    if (need_ckpt)
      checkpoint(t);
  }

  /* Call SCR_Finalize before MPI_Finalize */
  SCR_Finalize();

  MPI_Finalize();

  return 0;
}

void checkpoint(int timestep) {
  /* Define a name for our checkpoint */
  char name[256];
  sprintf(name, "timestep.%d", timestep);
```

(continues on next page)

(continued from previous page)

```
  /* Tell SCR that we are starting a checkpoint phase */
  SCR_Start_output(name, SCR_FLAG_CHECKPOINT);

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* Define our checkpoint file name */
  char file[256];
  sprintf(file, "%s/rank_%d.ckpt", name, rank);

  /* Register our checkpoint file with SCR,
   * and obtain path we should use to open it */
  char scr_file[SCR_MAX_FILENAME];
  SCR_Route_file(file, scr_file);

  /* Each process will inform SCR whether it wrote
   * its checkpoint successfully */
  int valid = 1;

  /* Use path from SCR to open checkpoint file for writing */
  FILE* fs = fopen(scr_file, "w");
  if (fs != NULL) {
    int rc = fwrite(state, ..., fs);
    if (rc == 0)
      /* Failed to write, mark checkpoint as invalid */
      valid = 0;

    fclose(fs);
  } else {
    /* Failed to open file, mark checkpoint as invalid */
    valid = 0;
  }

  /* Tell SCR that we have finished our checkpoint phase */
  SCR_Complete_output(valid);

  return;
}
```

Further sections in the user guide give more details and demonstrate how to perform restart with SCR. For a description of the API, see *SCR API*, and for more detailed instructions on integrating the API, see *Integrate SCR*.

It may also be instructive to examine the source of the `test_api.c` program and other programs in the examples directory.

2.1.5 Final Thoughts

This was a quick introduction to building and running with SCR. For more information, please look at the more detailed sections in the rest of this user guide.

2.2 Assumptions

A number of assumptions are made in the SCR implementation. If any of these assumptions do not hold for an application, the application may not be able to use certain features of SCR, or it might not be able to use SCR at all. If this is the case, or if you have any questions, please notify the SCR developers. The goal is to expand the implementation to support a large number of applications.

- The code must be an MPI application.
- For best performance, the code must read and write datasets as a file-per-process in a globally-coordinated fashion. There is support for reading and writing shared files, but one cannot utilize the fastest storage methods in that case.
- To use the scalable restart capability, a job must be restarted with the same number of processes it ran with when it wrote the checkpoint, and each process must only access the files it wrote during the checkpoint. Note that this may limit the effectiveness of the library for codes that can restart from a checkpoint with a different number of processes than were used to write the checkpoint. Such codes can often still benefit from the scalable checkpoint capability, but not the scalable restart – they must fall back to restarting from the parallel file system.
- It must be possible to store the dataset files from all processes in the same directory. In particular, all files belonging to a given dataset must have distinct names.
- Files cannot contain data that span multiple datasets. In particular, there is no support for appending data of the current dataset to a file containing data from a previous dataset. Each dataset must be self-contained.
- SCR maintains a set of meta data files that it stores in a subdirectory of the directory containing the application dataset files. The application must allow for these SCR meta data files to coexist with its own files.
- All files must reside under a top-level directory on the parallel file system called the "prefix" directory that is specified by the application. Under that prefix directory, the application may use subdirectory trees. See Section *Control, cache, and prefix directories* for details.
- Applications may configure SCR to cache datasets in RAM disk. One must ensure there is sufficient memory capacity to store the dataset files after accounting for the memory consumed by the application. The amount of storage needed depends on the number of cached datasets and the redundancy scheme that is applied. See Section *Scalable checkpoint* for details.
- Time limits should be imposed so that the SCR library has sufficient time to flush files from cache to the parallel file system before the resource allocation expires. See Section *Halt a job* for details.

2.3 Concepts

This section discusses concepts one should understand about the SCR library implementation including how it interacts with file systems. Terms defined here are used throughout the documentation.

2.3.1 Jobs, allocations, and runs

A large-scale simulation often must be restarted multiple times in order to run to completion. It may be interrupted due to a failure, or it may be interrupted due to time limits imposed by the resource scheduler. We use the term *allocation* to refer to an assigned set of compute resources that are available to the user for a period of time. A resource manager typically assigns an identifier label to each resource allocation, which we refer to as the *allocation id*. SCR embeds the allocation id in some directory and file names.

Within an allocation, a user may execute a simulation one or more times. We call each execution a *run*. For MPI applications, each run corresponds to a single invocation of `mpirun` or its equivalent.

Finally, multiple allocations may be required to complete a given simulation. We refer to this series of one or more allocations as a *job*.

To summarize, one or more runs occur within an allocation, and one or more allocations occur within a job.

2.3.2 Group, store, and redundancy descriptors

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

Each group is given a unique name. The library creates two groups by default: `NODE` and `WORLD`. The `NODE` group consists of all processes on the same compute node, and `WORLD` consists of all processes in the run. One can define additional groups via configuration files (see *Configure a job*).

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

The library creates one store descriptor by default: `/dev/shm`. The assumption is made that `/dev/shm` is mounted as a local file system on each compute node. On Linux clusters, `/dev/shm` is typically `tmpfs` (RAM disk), which implements a file system using main memory as the backing storage. Additional store descriptors can be defined in configuration files (see *Configure a job*).

Finally, SCR defines *redundancy descriptors* to associate a redundancy scheme with a class of storage devices and a group of processes that are likely to fail at the same time. It also tracks details about the particular redundancy scheme used, and the frequency with which it should be applied. Redundancy descriptors reference both store and group descriptors.

The library defines a default redundancy descriptor. It assumes that processes on the same node are likely to fail at the same time (compute node failure). It also assumes that datasets can be cached in `/dev/shm`, which is assumed to be storage local to each compute node. It applies an `XOR` redundancy scheme using a group size of 8. Additional redundancy descriptors may be defined in configuration files (see *Configure a job*).

2.3.3 Control, cache, and prefix directories

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

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

To construct the full path of the control directory, SCR incorporates a control base directory name along with the user name and allocation id associated with the resource allocation. This enables multiple users, or multiple jobs by the same user, to run at the same time without conflicting for the same control directory. A default control base directory is hard-coded into the SCR library at configure time, but this value may be overridden at runtime.

SCR can direct the application to write dataset files to subdirectories within a *cache directory*. SCR also stores its redundancy data in these subdirectories. The storage that hosts the cache directory must be large enough to hold the data for one or more datasets plus the associated redundancy data. Multiple cache directories may be utilized in the

same run, which enables SCR to use more than one class of storage within a run (e.g., RAM disk and SSD). Cache directories should ideally be located on scalable storage.

To construct the full path of a cache directory, SCR incorporates a cache base directory name with the user name and the allocation id associated with the resource allocation. It is valid for a cache directory to use the same base path as the control directory. A default cache base directory is hard-coded into the SCR library at configure time, but this value may be overridden at runtime.

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

SCR refers to each application checkpoint or output set as a *dataset*. SCR assigns a unique sequence number to each dataset called the *dataset id*. It assigns dataset ids starting from 1 and counts up with each successive dataset written by the application. Within a cache directory, a dataset is written to its own subdirectory called the *dataset directory*.

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

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

2.3.4 Example of SCR files and directories

To illustrate how files and directories are arranged in SCR, consider the example shown in Figure *Example SCR directories*. In this example, a user named `user1` runs a 3-task MPI job with one task per compute node. The base directory for the control directory is `/dev/shm`, the base directory for the cache directory is `/ssd`, and the prefix directory is `/pscratch/user1/simulation123`. The control and cache directories are storage devices local to the compute node.

The full path of the control directory is `/dev/shm/user1/scr.5132`. This is derived from the concatenation of the control base directory `/dev/shm`, the user name `user1`, and the allocation id `5132`. SCR keeps files to persist its internal state in the control directory, including a `cindex.scrinfo` file as shown.

Similarly, the cache directory is `/ssd/user1/scr.5132`, which is derived from the concatenation of the cache base directory `/ssd`, the user name `user1`, and the allocation id `5132`. Within the cache directory, SCR creates a subdirectory for each dataset. In this example, there are two datasets with ids `5` and `6`. The application dataset files and SCR redundancy files are stored within their corresponding dataset directory. On the node running MPI rank 0, there is one application dataset file `rank_0.ckpt`, and numerous SCR metadata files in a hidden `.scr` subdirectory.

The full path of the prefix directory is `/pscratch/user1/simulation123`. This is a path on the parallel file system that is specified by the user. It is unique to the particular simulation the user is running `simulation123`.

The prefix directory contains a hidden `.scr` directory where SCR writes its `index.scr` file to record info for each of the datasets (see *Manage datasets*). The SCR library writes other files to this hidden directory, including the `halt.scr` file (see *Halt a job*). Within the `.scr` directory, SCR also creates a directory for each dataset named `scr.dataset.<id>` where `<id>` is the dataset id. SCR stores metadata files that are specific to the dataset in these dataset directories including `summary.scr` and `rank2file` files along with redundancy files.

Application files for each dataset are written to their original path within the prefix directory as the application specified during its call to `SCR_Route_file`. In this example, the application stores all files for a particular dataset within its own subdirectory. There are directories named `ckpt.5` and `ckpt.6` corresponding to two datasets. The files from all processes for each dataset are contained within its respective `ckpt` directory. Application file names and directory

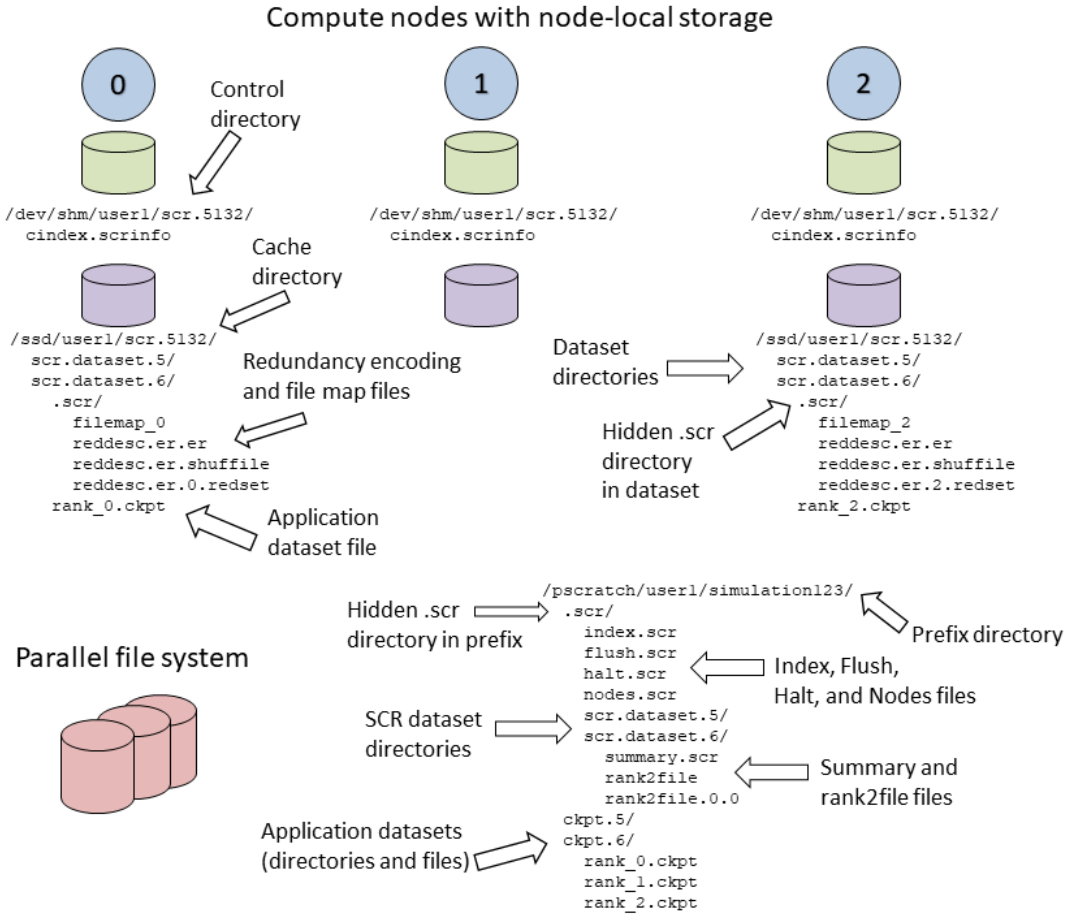

Fig. 1: Example SCR directories

paths can be arbitrary so long as all items are placed within the prefix directory and item names in each dataset are distinct from names in other datasets.

2.3.5 Scalable checkpoint

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

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

Users and system administrators can pass information about failure and storage groups to SCR in descriptors defined in configuration files (see *Group, store, and checkpoint descriptors*). Given this knowledge of failure and storage groups, the SCR library implements four redundancy schemes which trade off performance, storage space, and reliability:

- `Single` - each checkpoint file is written to storage accessible to the local process
- `Partner` - each checkpoint file is written to storage accessible to the local process, and a full copy of each file is written to storage accessible to a partner process from another failure group
- `XOR` - each checkpoint file is written to storage accessible to the local process, XOR parity data are computed from checkpoints of a set of processes from different failure groups, and the parity data are stored among the set.
- `RS` - each checkpoint file is written to storage accessible to the local process, and Reed-Solomon encoding data are computed from checkpoints of a set of processes from different failure groups, and the encoding data are stored among the set.

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

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

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

With `RS`, like `XOR`, SCR defines sets of processes where members within a set are selected from different failure groups. The processes within a set collectively compute Reed-Solomon encoding data which is stored in files along side the application checkpoint files. One may configure this scheme with the number of failures to tolerate within each set.

Computationally, `XOR` is more expensive than `Partner`, but it requires less storage space. Whereas `Partner` must store two full checkpoint files, `XOR` stores one full checkpoint file plus one XOR parity segment, where the segment size is roughly $1/(N-1)$ times the size of a checkpoint file for a set of size N. Similarly, `RS` is computationally more expensive than `XOR`, but it tolerates up to a configurable `k` number of failures per set. The `RS` encoding data scales as

$k/(N-k)$ times the size of a checkpoint file for a set of size N. Larger sets require less storage, but they also increase the probability that a given set will suffer multiple failures simultaneously. Larger sets may also increase the cost of recovering files in the event of a failure.

2.3.6 Scalable restart

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

Upon encountering a failure, SCR relies on the MPI library, the resource manager, or some other external service to kill the current run. After the run is killed, and if there are sufficient healthy nodes remaining, the same job can be restarted within the same allocation. In practice, such a restart typically amounts to issuing another `mpirun` in the job batch script.

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

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

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

2.3.7 Catastrophic failures

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

- **Multiple node failure which violates the redundancy scheme.** If multiple nodes fail in a pattern which violates the cache redundancy scheme, data are irretrievably lost.
- **Failure during a checkpoint.** Due to cache size limitations, some applications can only fit one checkpoint in cache at a time. For such cases, a failure may occur after the library has deleted the previous checkpoint but before the next checkpoint has completed. In this case, there is no valid checkpoint in cache to recover.
- **Failure of the node running the job batch script.** The logic at the end of the allocation to scavenge the latest checkpoint from cache to the parallel file system executes as part of the job batch script. If the node executing this script fails, the scavenge logic will not execute and the allocation will terminate without copying the latest checkpoint to the parallel file system.
- **Parallel file system outage.** If the application fails when writing output due to an outage of the parallel file system, the scavenge logic may also fail when it attempts to copy files to the parallel file system.

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

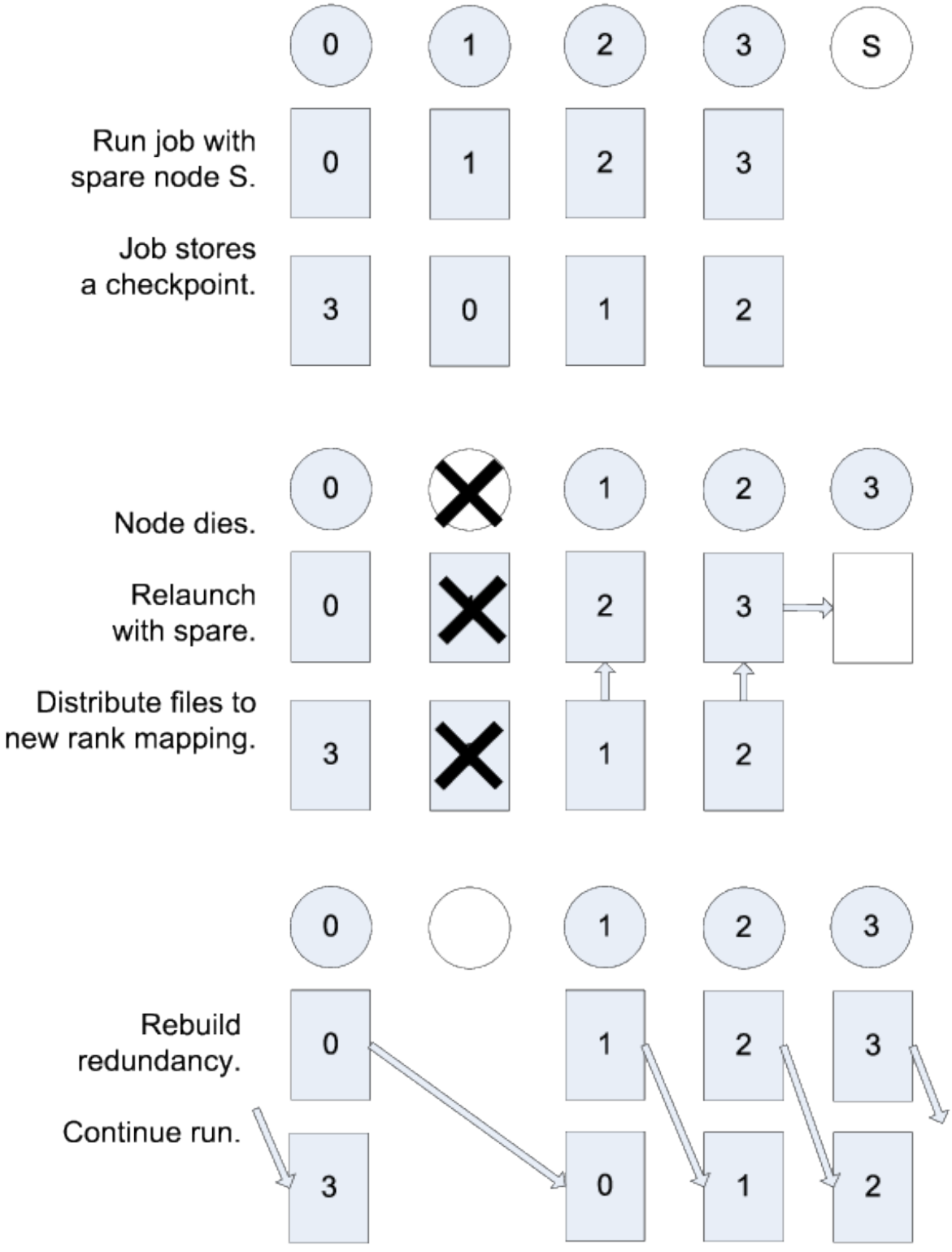

Fig. 2: Example restart after a failed node with `Partner`

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

2.3.8 Fetch, flush, and scavenge

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

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

To disable the fetch operation, set the `SCR_FETCH` parameter to 0. If an application disables the fetch feature, the application is responsible for reading its checkpoint set directly from the parallel file system upon a restart. In this case, the application should call `SCR_Current` to notify SCR which checkpoint it loaded. This enables SCR to set its internal state to maintain proper ordering in the checkpoint sequence.

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

2.4 Build SCR

2.4.1 Dependencies

SCR has several required dependencies. Others are optional, and if not available, corresponding SCR functionality is disabled.

Required:

- CMake, Version 2.8+
- Compilers (C, C++, and Fortran)
- MPI 3.0+
- pdsh (https://github.com/chaos/pdsh)
- DTCMP (https://github.com/llnl/dtcmp)

- LWGRP (https://github.com/llnl/lwgrp)
- AXL (https://github.com/ECP-VeloC/AXL)
- er (https://github.com/ECP-VeloC/er)
- KVTree (https://github.com/ECP-VeloC/KVTree)
- rankstr (https://github.com/ECP-VeloC/rankstr)
- redset (https://github.com/ECP-VeloC/redset)
- shuffile (https://github.com/ECP-VeloC/shuffile)
- spath (https://github.com/ECP-VeloC/spath)

Optional:

- libyogrt (for time remaining in a job allocation) (https://github.com/llnl/libyogrt)
- MySQL (for logging SCR activities)

To simplify the install process, one can use the SCR `bootstrap.sh` script with CMake or use Spack.

The CMake and Spack sections below assume that one is installing SCR on a system with existing compilers, a resource manager (like SLURM or LSF), and an MPI environment. These base software packages are typically preinstalled and configured for users by the support staff of HPC clusters.

For those who are installing SCR outside of an HPC cluster, are using Fedora, and have sudo access, the following steps install and activate most of the necessary base dependencies:

```
sudo dnf groupinstall "Development Tools"
sudo dnf install cmake gcc-c++ mpi mpi-devel environment-modules zlib-devel pdsh
[restart shell]
module load mpi
```

2.4.2 CMake

SCR requires CMake version 2.8 or higher. The SCR build uses the CMake FindMPI module to link with MPI. This module looks for the standard `mpicc` compiler wrapper, which must be in your `PATH`.

The quick version of building SCR with CMake is:

```
git clone git@github.com:llnl/scr.git
cd scr
git checkout v3.0rc1

./bootstrap.sh

mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=../install ..
make install
```

Some useful CMake command line options are:

- `-DCMAKE_INSTALL_PREFIX=[path]`: Place to install the SCR library
- `-DCMAKE_BUILD_TYPE=[Debug/Release]`: Build with debugging or optimizations
- `-DSCR_RESOURCE_MANAGER=[SLURM/APRUN/PMIX/LSF/NONE]`
- `-DSCR_ASYNC_API=[INTEL_CPPR/NONE]`

- `-DSCR_CNTL_BASE=[path]` : Path to SCR Control directory, defaults to `/dev/shm`
- `-DSCR_CACHE_BASE=[path]` : Path to SCR Cache directory, defaults to `/dev/shm`
- `-DSCR_CONFIG_FILE=[path]` : Path to SCR system configuration file, defaults to `/etc/scr/scr.conf`

For setting the default logging parameters:

- `-DSCR_LOG_ENABLE=[0/1]` : Whether to enable SCR logging of any type (1) or not (0), defaults to `0`
- `-DSCR_LOG_SYSLOG_ENABLE=[0/1]` : Whether to enable SCR logging via syslog (1) or not (0), defaults to `1`
- `-DSCR_LOG_SYSLOG_FACILITY=[facility]` : Facility for syslog messages (see man openlog), defaults to `LOG_LOCAL7`
- `-DSCR_LOG_SYSLOG_LEVEL=[level]` : Level for syslog messages (see man openlog), defaults to `LOG_INFO`
- `-DSCR_LOG_SYSLOG_PREFIX=[str]` : Prefix string to prepend to syslog messages, defaults to `SCR`
- `-DSCR_LOG_TXT_ENABLE=[0/1]` : Whether to enable SCR logging to a text file (1) or not (0), defaults to `1`

If one has installed SCR dependencies in different directories, there are CMake options to specify the path to each as needed, e.g.,:

- `-DBUILD_PDSH=[OFF/ON]`: CMake can automatically download and build the PDSH dependency
- `-DWITH_PDSH_PREFIX=[path to PDSH]`: Path to an existing PDSH installation (should not be used with `BUILD_PDSH`)
- `-DWITH_DTCMP_PREFIX=[path to DTCMP]`
- `-DWITH_AXL_PREFIX=[path to AXL]`
- `-DWITH_ER_PREFIX=[path to er]`
- `-DWITH_KVTREE_PREFIX=[path to KVTree]`
- `-DWITH_RANKSTR_PREFIX=[path to rankstr]`
- `-DWITH_REDSET_PREFIX=[path to redset]`
- `-DWITH_SHUFFILE_PREFIX=[path to shuffile]`
- `-DWITH_SPATH_PREFIX:PATH=[path to spath]`
- `-DWITH_YOGRT_PREFIX:PATH=[path to libyogrt]`
- `-DWITH_MYSQL_PREFIX=[path to MySQL]`

2.4.3 Spack

If you use the Spack package manager, SCR and many of its dependencies have corresponding packages.

Before installing SCR with Spack, one should first properly configure `packages.yaml`. In particular, SCR depends on the system resource manager and MPI library, and one should define entries for those in `packages.yaml`.

By default, Spack attempts to build all dependencies for SCR, including packages such as SLURM, MPI, and OpenSSL that are already installed on most HPC systems. It is recommended to use the system-installed software when possible. This ensures that the resulting SCR build actually works on the target system, and it can significantly reduce the build time.

Spack uses its `packages.yaml` file to locate external packages. Full information about `packages.yaml` can be found in the Spack documentation.

At minimum, it is important to register the system MPI library and the system resource manager. Other packages can be defined to accelerate the build. The following shows example entries for `packages.yaml`. One must modify these example entries to use the proper versions, module names, and paths for the target system:

```
packages:
  all:
    providers:
      mpi: [mvapich2,openmpi,spectrum-mpi]

  # example entry for MVAPICH2 MPI, accessed by a module named mvapich2
  mvapich2:
    buildable: false
    externals:
    - spec: mvapich2
      modules:
      - mvapich2

  # example entry for Open MPI
  openmpi:
    buildable: false
    externals:
    - spec: openmpi@4.1.0
      prefix: /opt/openmpi-4.1.0

  # example entry for IBM Spectrum MPI
  spectrum-mpi:
    buildable: false
    externals:
    - spec: spectrum-mpi
      prefix: /opt/ibm/spectrum_mpi

  # example entry for IBM LSF resource manager
  lsf:
    buildable: false
    externals:
    - spec: lsf@10.1
      prefix: /opt/ibm/spectrumcomputing/lsf/10.1

  # example entry for SLURM resource manager
  slurm:
    buildable: false
    externals:
    - spec: slurm@20
      prefix: /usr

  openssl:
    externals:
    - spec: openssl@1.0.2
      prefix: /usr

  libyogrt:
    externals:
    - spec: libyogrt scheduler=lsf
      prefix: /usr
    - spec: libyogrt scheduler=slurm
```

(continues on next page)

(continued from previous page)

```
      prefix: /usr
```

The *packages* key declares the following block as a set of package descriptions. The following descriptions tell Spack how to find items that already installed on the system.

- The *providers* key specifies that one of three different MPI versions are available, MVAPICH2, Open MPI, or IBM Spectrum MPI.
- `mvapich2`: declares that MVAPICH2 is available, and the location is defined in a *mvapich2* module file.
- `openmpi`: declares that Open MPI is installed in the system at the location specified by *prefix*, and the *buildable: false* line declares that Spack should always use that version of MPI rather than try to build its own. This description addresses the common situation where MPI is customized and optimized for the local system, and Spack should never try to compile a replacement.
- `spectrum-mpi`: declares that Spectrum MPI is available.
- `lsf`: declares that if LSF is needed (e.g. to use *scheduler=lsf*) the libraries can be found at the specified *prefix*.
- `slurm`: declares that if SLURM is needed (e.g. to use *scheduler=slurm*) the libraries can be found at the specified *prefix*.
- `openssl`: declares that *openssl* version 1.0.2 is installed on the system and that Spack should use that if it satisfies the dependencies required by any spack-installed packages, but if a different version is requested, Spack should install its own version.
- `libyogrt`: declares that libyogrt is installed, but Spack may decide to build its own version. If *scheduler=slurm* or *scheduler=lsf* is selected, use the version installed under /usr, otherwise build from scratch using the selected scheduler.

After configuring `packages.yaml`, one can install SCR.

For SLURM systems, SCR can be installed with:

```
spack install scr@3.0rc1 resource_manager=SLURM
```

For LSF, systems, SCR can be installed with:

```
spack install scr@3.0rc1 resource_manager=LSF
```

The SCR Spack package provides other variants that may be useful. To see the full list, type:

```
spack info scr
```

2.5 SCR API

SCR is designed to support MPI applications that write application-level checkpoint and output datasets. All processes must access data in a globally-coordinated fashion, and in fact, many SCR calls are implicit collectives over all processes in `MPI_COMM_WORLD`. In a given dataset, each process may write zero or more files, but the current implementation is most efficient when all processes write the same amount of data. Multiple processes can access a given file, though significant performance is gained by applications that use file-per-process access patterns, where each file is only accessed by a single "owner" process.

Parallel file systems allow any process in an MPI job to read/write any byte of a file at any time. However, most applications do not require this full generality. SCR supplies API calls that enable the application to specify limits on its data access in both time and space. Start and complete calls indicate when an application needs to write or read its data. Files in the dataset cannot be accessed outside of these markers. Additionally, for best performance, each

MPI process may only access files written by itself or another process having the same MPI rank in a previous run. In this mode, an MPI process cannot access files written by a process having a different MPI rank. SCR can provide substantial improvements in I/O performance by enabling an application to specify its limits on when and where it needs to access its data.

The API is designed to be simple, scalable, and portable. It consists of a small number of function calls to wrap existing application I/O logic. Unless otherwise stated, SCR functions are collective, meaning all processes must call the function synchronously. The underlying implementation may or may not be synchronous, but to be portable, an application must treat a collective call as though it is synchronous. This constraint enables the SCR implementation to utilize the full resources of the job in a collective manner to optimize performance at critical points such as computing redundancy data.

In the sections below, we show the function prototypes for C and Fortran. Applications written in C should include `scr.h`, and Fortran applications should include `scrf.h`. Unless otherwise noted, all functions return `SCR_SUCCESS` if successful.

2.5.1 Startup and Shutdown API

SCR_Init

```
int SCR_Init(void);
```

```
SCR_INIT(IERROR)
  INTEGER IERROR
```

Initialize the SCR library. This function must be called after `MPI_Init`. A process should only call `SCR_Init` once during its execution. It is not valid to call any SCR function before calling `SCR_Init`, except for `SCR_Config`.

On new runs, if `SCR_FETCH` is enabled, `SCR_Init` reads the index file to identify a checkpoint to load. On runs restarted within an allocation, `SCR_Init` inspects and attempts to rebuild any cached datasets.

SCR_Finalize

```
int SCR_Finalize(void);
```

```
SCR_FINALIZE(IERROR)
  INTEGER IERROR
```

Shut down the SCR library. This function must be called before `MPI_Finalize`. A process should only call `SCR_Finalize` once during its execution.

If `SCR_FLUSH` is enabled, `SCR_Finalize` flushes any datasets to the prefix directory if necessary. It updates the halt file to indicate that `SCR_Finalize` has been called. This halt condition prevents the job from restarting (see *Halt a job*).

SCR_Config

```
const char* SCR_Config(const char* config);
const char* SCR_Configf(const char* format, ...);
```

```
SCR_CONFIG(CONFIG, VAL, IERROR)
  CHARACTER*(*) CONFIG , VAL
  INTEGER IERROR
```

Configure the SCR library. Most of the SCR configuration parameters listed in *Configure a job* can be set at run time using `SCR_Config`. The syntax to set parameters matches the syntax used in SCR configuration files. The application can make multiple calls to `SCR_Config`. All calls to `SCR_Config` must come before the application calls `SCR_Init`. This function is collective, and all processes must provide identical values for `config`.

To set a parameter, one specifies a parameter name and its value in the form of a `key=value` string as the `config` argument. For example, passing the string `SCR_FLUSH=10` sets `SCR_FLUSH` to the value of 10. If one sets the same parameter with multiple calls to `SCR_Config`, SCR applies the most recent value. When setting a parameter, for C applications, `SCR_Config` always returns `NULL`. For Fortran applications, `IERROR` is always set to `SCR_SUCCESS`.

To query a value, one specifies just the parameter name as the string in `config`. For example, one can specify the string `SCR_FLUSH` to query its current value. When querying a value, for C applications, the call allocates and returns a pointer to a string holding the value of the parameter. The caller is responsible for calling `free` to release the returned string. If the parameter has not been set, `NULL` is returned. For Fortran applications, the value is returned as a string in the `VAL` argument.

To unset a value, one specifies the parameter name with an empty value in the form of a `key=` string as the `config` argument. For example, to unset the value assigned to `SCR_FLUSH`, specify the string `SCR_FLUSH=`. Unsetting a parameter removes any value that was assigned by a prior call to `SCR_Config`, but it does not unset the parameter value that may have been set through other means, like an environment variable or in a configuration file (see *Configure a job*). When unsetting a value, for C applications, `SCR_Config` always returns `NULL`. For Fortran applications, `IERROR` is always set to `SCR_SUCCESS`.

Multi-item SCR configuration parameters like `CKPT` can be set using a sequence of `key=value` pairs that are separated by spaces. For example, to define a `CKPT` redundancy descriptor, one can pass a string such as `CKPT=0 TYPE=XOR SET_SIZE=16`.

To query a subvalue, one must specify the top level `key=value` pair followed by the name of the key being queried. For instance, to get the type of the redundancy scheme of redundancy descriptor `0`, one can specify the string `CKPT=0 TYPE`.

For C applications, `SCR_Configf` provides a formatted string variant of `SCR_Config`. The caller can use printf-style formatting patterns to define the string, as in `SCR_Configf("SCR_FLUSH=%d", 10)`. This call otherwise behaves the same as `SCR_Config`.

2.5.2 File Routing API

When files are under control of SCR, they may be written to or exist on different levels of the storage hierarchy at different points in time. For example, a checkpoint might be written first to the RAM disk of a compute node and then later transferred to the parallel file system by SCR. In order for an application to discover where a file should be written to or read from, one calls the `SCR_Route_file` routine.

The precise behavior of `SCR_Route_file` varies depending on the current state of SCR. Depending on the calling context, sections below extend the definition as described in this section. This section describes general information about `SCR_Route_file` that applies in all contexts.

SCR_Route_file

```
int SCR_Route_file(const char* name, char* file);
```

```
SCR_ROUTE_FILE(NAME, FILE, IERROR)
  CHARACTER*(*) NAME, FILE
  INTEGER IERROR
```

A process calls `SCR_Route_file` to obtain the full path and file name it must use to access a file. A call to `SCR_Route_file` is local to the calling process; it is not a collective call.

The name of the file that the process intends to access must be passed in the `name` argument. This must be a relative or absolute path that specifies the location of the file on the parallel file system. If given a relative path, SCR prepends the current working directory at the time `SCR_Route_file` is called. This path must resolve to a location under the prefix directory.

A pointer to a character buffer of at least `SCR_MAX_FILENAME` bytes must be passed in `file`. When a call to `SCR_Route_file` returns, the full path and file name to access the file named in `name` is written to the buffer pointed to by `file`. The process must use the character string returned in `file` to access the file.

If `SCR_Route_file` is called outside of output and restart phases, i.e., outside of a Start/Complete pair, the string in `name` is copied verbatim into the output buffer `file`.

In the current implementation, SCR only changes the directory portion of `name` when storing files in cache. It extracts the base name of the file by removing any directory components in `name`. Then it prepends a cache directory to the base file name and returns the full path and file name in `file`.

2.5.3 Checkpoint/Output API

Here we describe the SCR API functions that are used for writing checkpoint and output datasets. In addition to checkpoints, it may be useful for an application to write its pure output (non-checkpoint) datasets through SCR to utilize asynchronous transfers to the parallel file system. This lets the application return to computation while the SCR library migrates the dataset to the parallel file system in the background.

Using a combination of bit flags, a dataset can be designated as a checkpoint, output, or both. The checkpoint property means that the dataset can be used to restart the application. The output property means that the dataset must be written to the prefix directory.

If a user specifies that a dataset is a checkpoint only, then SCR may delete an older checkpoint to store a more recent checkpoint without having first copied the older checkpoint to the prefix directory. SCR may thus discard some checkpoints from cache without persisting them to the parallel file system. In cases where one can write checkpoints to cache much faster than one can write checkpoints to the parallel file system, discarding defensive checkpoints in this way allows the application to checkpoint more frequently, which in turn can significantly improve run time efficiency.

If a user specifies that a dataset is for output only, the dataset will first be cached and protected with its corresponding redundancy scheme. Then the dataset will be copied to the prefix directory. When the transfer to the prefix directory is complete, the cached copy of the output dataset is deleted.

If the user specifies that the dataset is both a checkpoint and output, then SCR uses a hybrid approach. The dataset is copied to the prefix directory as output, but it is also kept in cache according to the policy set in the configuration for checkpoints. For example, if the user configures SCR to keep three checkpoints in cache, then the dataset will be preserved in cache until it is replaced by a newer checkpoint after three more checkpoint phases.

SCR_Need_checkpoint

```
int SCR_Need_checkpoint(int* flag);
```

```
SCR_NEED_CHECKPOINT(FLAG, IERROR)
  INTEGER FLAG, IERROR
```

Since the failure frequency and the cost of checkpointing vary across platforms, `SCR_Need_checkpoint` provides a portable way for an application to determine whether a checkpoint should be taken. This function is passed a pointer to an integer in `flag`. Upon returning from `SCR_Need_checkpoint`, `flag` is set to the value `1` if a checkpoint should be taken, and it is set to `0` otherwise. The call returns the same value in `flag` on all processes.

SCR_Start_output

```
int SCR_Start_output(char* name, int flags);
```

```
SCR_START_OUTPUT(NAME, FLAGS, IERROR)
  CHARACTER*(*) NAME
  INTEGER FLAGS, IERROR
```

Inform SCR that a new output phase is about to start. A process must call this function before it creates any files belonging to the dataset. `SCR_Start_output` must be called by all processes, including processes that do not write files as part of the dataset.

The caller can provide a name for the dataset in `name`. It is recommended to define names that are user-friendly, since an end user may need to read and type these names at times. The name value must be less than `SCR_MAX_FILENAME` characters. All processes must provide identical values in `name`. In C, the application may pass `NULL` for name in which case SCR generates a default name for the dataset based on its internal dataset id.

The dataset can be output, a checkpoint, or both. The caller specifies these properties using `SCR_FLAG_OUTPUT` and `SCR_FLAG_CHECKPOINT` bit flags. Additionally, a `SCR_FLAG_NONE` flag is defined for initializing variables. In C, these values can be combined with the `|` bitwise OR operator. In Fortran, these values can be added together using the + sum operator. Note that with Fortran, the values should be used at most once in the addition. All processes must provide identical values in `flags`.

This function should be called as soon as possible when initiating a dataset output. It is used internally within SCR for timing the cost of output operations. The SCR implementation uses this call as the starting point to time the cost of the checkpoint in order to optimize the checkpoint frequency via `SCR_Need_checkpoint`.

Each call to `SCR_Start_output` must be followed by a corresponding call to `SCR_Complete_output`.

In the current implementation, `SCR_Start_output` holds all processes at an `MPI_Barrier` to ensure that all processes are ready to start the output before it deletes cached files from a previous checkpoint.

SCR_Route_file

```
int SCR_Route_file(const char* name, char* file);
```

```
SCR_ROUTE_FILE(NAME, FILE, IERROR)
  CHARACTER*(*) NAME, FILE
  INTEGER IERROR
```

A process must call `SCR_Route_file` for each file it writes as part of the output dataset. It is valid for a process to call `SCR_Route_file` multiple times for the same file.

When called within an output phase, between `SCR_Start_output` and `SCR_Complete_output`, `SCR_Route_file` registers the file as part of the output dataset.

A process does not need to create any directories listed in the string returned in `file`. The SCR implementation creates any necessary directories before it returns from `SCR_Route_file`. After returning from `SCR_Route_file`, the process may create and open the target file for writing.

SCR_Complete_output

```
int SCR_Complete_output(int valid);
```

```
SCR_COMPLETE_OUTPUT(VALID, IERROR)
  INTEGER VALID, IERROR
```

Inform SCR that all files for the current dataset output are complete (i.e., done writing and closed) and whether they are valid (i.e., written without error). A process must close all files in the dataset before calling `SCR_Complete_output`, and it may no longer access its dataset files upon calling `SCR_Complete_output`. `SCR_Complete_output` must be called by all processes, including processes that did not write any files as part of the output.

The parameter `valid` should be set to `1` if either the calling process wrote all of its files successfully or it wrote no files during the output phase. Otherwise, the process should call `SCR_Complete_output` with `valid` set to `0`. SCR determines whether all processes wrote their output files successfully. `SCR_Complete_output` only returns `SCR_SUCCESS` if all processes called with `valid` set to `1`, meaning that all processes succeeded in their output. The call returns the same value on all processes.

Each call to `SCR_Complete_output` must be preceded by a corresponding call to `SCR_Start_output`. The SCR implementation uses this call as the stopping point to time the cost of the checkpoint that started with the preceding call to `SCR_Start_output`.

In the current implementation, SCR applies the redundancy scheme during `SCR_Complete_output`. The dataset is then flushed to the prefix directory if needed.

2.5.4 Restart API

Here we describe the SCR API functions used for restarting applications.

SCR_Have_restart

```
int SCR_Have_restart(int* flag, char* name);
```

```
SCR_HAVE_RESTART(FLAG, NAME, IERROR)
  INTEGER FLAG
  CHARACTER*(*) NAME
  INTEGER IERROR
```

This function indicates whether SCR has a checkpoint available for the application to read. This function is passed a pointer to an integer in `flag`. Upon returning from `SCR_Have_restart`, `flag` is set to the value `1` if a checkpoint is available, and it is set to `0` otherwise. The call returns the same value in `flag` on all processes.

A pointer to a character buffer of at least `SCR_MAX_FILENAME` bytes can be passed in `name`. If there is a checkpoint, and if that checkpoint was assigned a name when it was created, `SCR_Have_restart` returns the name of that checkpoint in `name`. The value returned in `name` is the same string that was passed to `SCR_Start_output` when the checkpoint was created. The same value is returned in `name` on all processes. In C, one may optionally pass `NULL` to this function to avoid returning the name.

SCR_Start_restart

```
int SCR_Start_restart(char* name);
```

```
SCR_START_RESTART(NAME, IERROR)
  CHARACTER*(*) NAME
  INTEGER IERROR
```

This function informs SCR that a restart operation is about to start. A process must call this function before it opens any files belonging to the restart. SCR_Start_restart must be called by all processes, including processes that do not read files as part of the restart.

SCR returns the name of the loaded checkpoint in name. A pointer to a character buffer of at least SCR_MAX_FILENAME bytes can be passed in name. The value returned in name is the same string that was passed to SCR_Start_output when the checkpoint was created. The same value is returned in name on all processes. In C, one may optionally pass NULL to this function to avoid returning the name.

One may only call SCR_Start_restart when SCR_Have_restart indicates that there is a checkpoint to read. SCR_Start_restart returns the same value in name as the preceding call to SCR_Have_restart.

Each call to SCR_Start_restart must be followed by a corresponding call to SCR_Complete_restart.

SCR_Route_file

```
int SCR_Route_file(const char* name, char* file);
```

```
SCR_ROUTE_FILE(NAME, FILE, IERROR)
  CHARACTER*(*) NAME, FILE
  INTEGER IERROR
```

A process must call SCR_Route_file for each file it reads during restart. It is valid for a process to call SCR_Route_file multiple times for the same file.

When called within a restart phase, between SCR_Start_restart and SCR_Complete_restart, SCR checks whether the file exists and is readable. In this mode, SCR_Route_file returns an error code if the file does not exist or is not readable.

It is recommended to provide the relative or absolute path to the file under the prefix directory in name. However, for backwards compatibility, the caller may provide only a file name in name, even if prepending the current working directory to the file name does not resolve to the correct path to the file on the parallel file system. Using just the file name, SCR internally looks up the full path to the file using SCR metadata for the currently loaded checkpoint. This usage is deprecated, and it may be not be supported in future releases.

SCR_Complete_restart

```
int SCR_Complete_restart(int valid);
```

```
SCR_COMPLETE_RESTART(VALID, IERROR)
  INTEGER VALID, IERROR
```

This call informs SCR that the process has finished reading its checkpoint files. A process must close all restart files before calling SCR_Complete_restart, and it may no longer access its restart files upon calling SCR_Complete_restart. SCR_Complete_restart must be called by all processes, including processes that did not read any files as part of the restart.

The parameter valid should be set to 1 if either the calling process read all of its files successfully or it read no files as part of the restart. Otherwise, the process should call SCR_Complete_restart with valid set to 0. SCR determines whether all processes read their checkpoint files successfully based on the values supplied in the valid parameter. SCR_Complete_restart only returns SCR_SUCCESS if all processes called with valid set to 1, meaning that all processes succeeded in their restart. The call returns the same value on all processes.

If the restart failed on any process, SCR loads the next most recent checkpoint, and the application can call SCR_Have_restart to determine whether a new checkpoint is available. An application can loop until it either successfully restarts from a checkpoint or it exhausts all known checkpoints.

Each call to SCR_Complete_restart must be preceded by a corresponding call to SCR_Start_restart.

2.5.5 General API

SCR_Get_version

```
char* SCR_Get_version(void);
```

```
SCR_GET_VERSION(VERSION, IERROR)
  CHARACTER*(*) VERSION
  INTEGER IERROR
```

This function returns a string that indicates the version number of SCR that is currently in use. The caller must not free the returned version string.

SCR_Should_exit

```
int SCR_Should_exit(int* flag);
```

```
SCR_SHOULD_EXIT(FLAG, IERROR)
  INTEGER FLAG, IERROR
```

SCR_Should_exit provides a portable way for an application to determine whether it should halt its execution. This function is passed a pointer to an integer in flag. Upon returning from SCR_Should_exit, flag is set to the value 1 if the application should stop, and it is set to 0 otherwise. The call returns the same value in flag on all processes. It is recommended to call this function after each checkpoint.

It is critical for a job to stop early enough to leave time to copy datasets from cache to the prefix directory before the allocation expires. One can configure how early the job should exit within its allocation by setting the SCR_HALT_SECONDS parameter.

This call also enables a running application to react to external commands. For instance, if the application has been instructed to halt using the scr_halt command, then SCR_Should_exit relays that information.

2.5.6 Dataset Management API

SCR provides functions to manage existing datasets. These functions take a name argument, which corresponds to the same name the application assigned to the dataset when it called SCR_Start_output.

SCR_Current

```
int SCR_Current(const char* name);
```

```
SCR_CURRENT(NAME, IERROR)
  CHARACTER*(*) NAME
  INTEGER VALID, IERROR
```

It is recommended for an application to restart using the SCR Restart API. However, it is not required to do so. If an application restarts without using the SCR Restart API, it can call SCR_Current to notify SCR about which checkpoint it loaded. The application should pass the name of the checkpoint it restarted from in the name argument.

This enables SCR to initialize its internal state to properly order any new datasets that the application creates after it restarts.

An application should not call `SCR_Current` if it restarts using the SCR Restart API.

SCR_Delete

```
int SCR_Delete(const char* name);
```

```
SCR_DELETE(NAME, IERROR)
  CHARACTER*(*) NAME
  INTEGER VALID, IERROR
```

Instruct SCR to delete a dataset. The application provides the name of the dataset to be deleted in the `name` argument. SCR deletes all application files and its own internal metadata associated with that dataset from both the prefix directory and cache. SCR also deletes any directories that become empty as a result of deleting the dataset files up to the SCR prefix directory.

SCR_Drop

```
int SCR_Drop(const char* name);
```

```
SCR_DROP(NAME, IERROR)
  CHARACTER*(*) NAME
  INTEGER VALID, IERROR
```

Instruct SCR to drop an entry for a dataset from the SCR index file. SCR removes the entry for that dataset, but it does not delete any data files. A common use for this function is to remove entries for datasets that an application or user has deleted outside of SCR. For instance, if an application deletes a dataset without calling `SCR_Delete`, it can call `SCR_Drop` to maintain a consistent view of available datasets in the SCR index file.

2.5.7 Space/time semantics

SCR imposes the following semantics which enable an application to limit where and when it accesses its data:

- For best performance, 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. Shared access to files is permitted, though that may reduce performance and functionality.
- During a checkpoint/output phase, a process may only access files of the dataset between calls to `SCR_Start_output` and `SCR_Complete_output`. Once a process calls `SCR_Complete_output` it may no longer access any file it registered as part of that dataset through a call to `SCR_Route_file`.
- During a restart, a process may only access files from the currently loaded checkpoint, and it must access those files between calls to `SCR_Start_restart` and `SCR_Complete_restart`. Once a process calls `SCR_Complete_restart` it may no longer access its restart files. SCR selects which checkpoint is considered to be the "most recent".

These semantics enable SCR to cache 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 files as needed, such that it can manage this cache.

2.5.8 SCR API state transitions

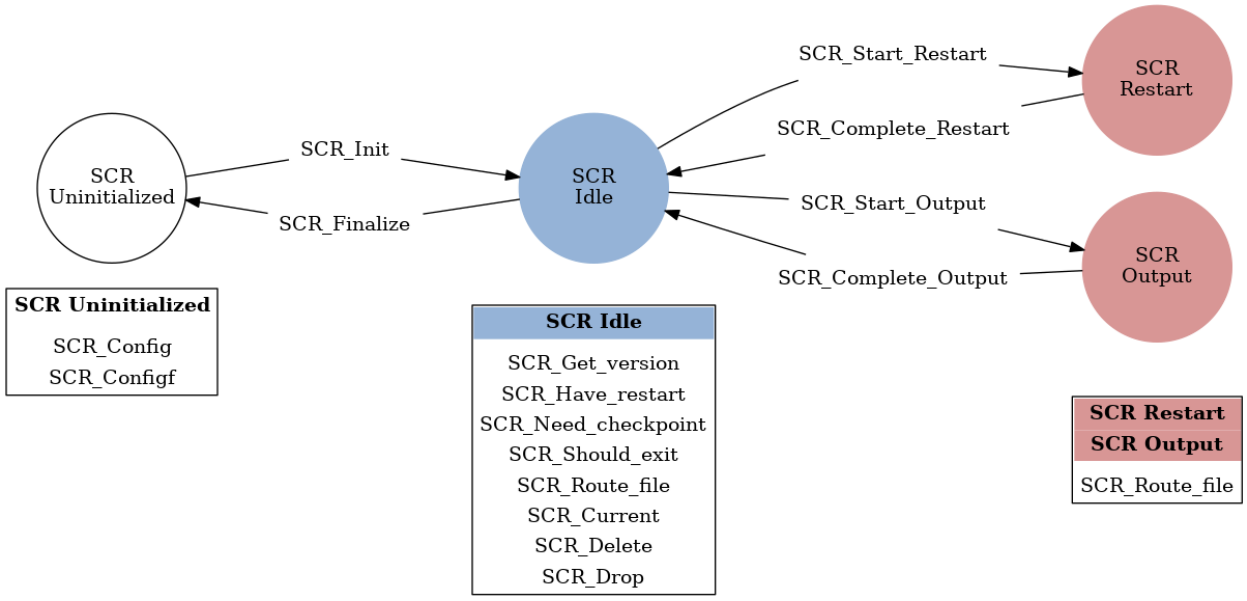

Fig. 3: SCR API State Transition Diagram

Figure *SCR API State Transition Diagram* illustrates the internal states in SCR and which API calls can be used from within each state. The application must call `SCR_Init` before it may call any other SCR function, except for `SCR_Config`, and it may not call SCR functions after calling `SCR_Finalize`. Some calls transition SCR from one state to another as shown by the edges between states. Other calls are only valid when in certain states as shown in the boxes. For example, `SCR_Have_restart` is only valid within the Idle state. All SCR functions are implicitly collective across `MPI_COMM_WORLD`, except for `SCR_Route_file` and `SCR_Get_version`.

2.6 Integrate SCR

This section provides details on how to integrate the SCR API into an application. There are three steps to consider: Init/Finalize, Checkpoint, and Restart. It is recommended to restart using the SCR Restart API, but it is not required. Sections below describe each case.

2.6.1 Using the SCR API

Before adding calls to the SCR library, consider that an application has existing checkpointing code that looks like the following

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

  /* initialize our state from checkpoint file */
  state = restart();

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

(continues on next page)

(continued from previous page)

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

  MPI_Finalize();
  return 0;
}

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

  /* define checkpoint directory for the timestep */
  char checkpoint_dir[256];
  sprintf(checkpoint_dir, "timestep.%d", timestep);

  /* 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 = fopen(checkpoint_file, "w");
  if (fs != NULL) {
    fwrite(checkpoint_data, ..., fs);
    fclose(fs);
  }

  /* wait for all files to be closed */
  MPI_Barrier(MPI_COMM_WORLD);

  /* rank 0 updates the pointer to the latest checkpoint */
  FILE* fs = fopen("latest", "w");
  if (fs != NULL) {
    fwrite(checkpoint_dir, ..., fs);
    fclose(fs);
  }
}

void* restart() {
  /* rank 0 broadcasts directory name to read from,
   * and then each process reads its state from a file */

  /* get rank of this process */
  int rank;
```

(continues on next page)

(continued from previous page)

```
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* rank 0 reads and broadcasts checkpoint directory name */
  char checkpoint_dir[256];
  if (rank == 0) {
    FILE* fs = fopen("latest", "r");
    if (fs != NULL) {
      fread(checkpoint_dir, ..., fs);
      fclose(fs);
    }
  }
  MPI_Bcast(checkpoint_dir, sizeof(checkpoint_dir), MPI_CHAR, ...);

  /* 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, reads, and closes its file */
  FILE* fs = fopen(checkpoint_file, "r");
  if (fs != NULL) {
    fread(state, ..., fs);
    fclose(fs);
  }

  return state;
}
```

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

Init/Finalize

You must add calls to `SCR_Init` and `SCR_Finalize` in order to start up and shut down the library. The SCR library uses MPI internally, and all calls to SCR must be from within a well defined MPI environment, i.e., between `MPI_Init` and `MPI_Finalize`. It is recommended to call `SCR_Init` close to `MPI_Init` and to call `SCR_Finalize` just before `MPI_Finalize`. For example, modify the source to look something like this

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

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

  state = restart();

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

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

(continues on next page)

(continued from previous page)

```
    /**** change #3 ****/
    int should_exit;
    SCR_Should_exit(&should_exit);
    if (should_exit)
      break;
  }

  /**** change #4 ****/
  SCR_Finalize();

  MPI_Finalize();
  return 0;
}
```

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. Internally, SCR duplicates `MPI_COMM_WORLD` during `SCR_Init`, so MPI messages from the SCR library do not mix with messages sent by the application.

One may configure SCR with calls to `SCR_Config`. Any calls to `SCR_Config` must come before `SCR_Init`. It is common to configure SCR depending on command line options the user passes to the application, so it is typical to place `SCR_Init` after application command line processing.

As shown in change #4, one should shut down the SCR library by calling `SCR_Finalize()`. This must be done before calling `MPI_Finalize()`. Some applications contain multiple calls to `MPI_Finalize`. In such cases, be sure to account for each call.

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. For this, the application should call `SCR_Need_checkpoint` at each opportunity that it could checkpoint, e.g., at the end of each time step, and then initiate a checkpoint when SCR advises it to do so. An application may ignore the output of `SCR_Need_checkpoint`, and it does not have to call the function at all. The intent of `SCR_Need_checkpoint` is to provide a portable way for an application to determine when to checkpoint across platforms with different reliability characteristics and different file system speeds.

Also note how the application can call `SCR_Should_exit` to determine whether it is time to stop as shown in change #3. This is important so that an application stops with sufficient time remaining to copy datasets from cache to the parallel file system before the allocation expires. It is recommended to call this function after completing a checkpoint.

Checkpoint

To actually write a checkpoint, there are three steps. First, the application must call `SCR_Start_output` with the `SCR_FLAG_CHECKPOINT` flag to define the start boundary of a new checkpoint. It must do this before it creates any file belonging to the new checkpoint. Then, the application must call `SCR_Route_file` for each file that it will write in order to register the file with SCR and to determine the full path and file name to open each file. Finally, it must call `SCR_Complete_output` to define the end boundary of the checkpoint.

If a process does not write any files during a checkpoint, it must still call `SCR_Start_output` and `SCR_Complete_output` as these functions are collective over all processes. All files registered through a call to `SCR_Route_file` between a given `SCR_Start_output` and `SCR_Complete_output` pair are considered to be part of the same checkpoint file set. Some example SCR checkpoint code looks like the following

```
void checkpoint(int timestep) {
  /* each process saves its state to a file */

  /* define checkpoint directory for the timestep */
  char checkpoint_dir[256];
  sprintf(checkpoint_dir, "timestep.%d", timestep);

  /**** change #5 ****/
  SCR_Start_output(checkpoint_dir, SCR_FLAG_CHECKPOINT);

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

  /**** change #6 ****/
  /*
      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 #7 ****/
  char scr_file[SCR_MAX_FILENAME];
  SCR_Route_file(checkpoint_file, scr_file);

  /**** change #8 ****/
  /* each rank opens, writes, and closes its file */
  int valid = 1;
  FILE* fs = fopen(scr_file, "w");
  if (fs != NULL) {
    int write_rc = fwrite(checkpoint_data, ..., fs);
    if (write_rc == 0) {
      /* failed to write file, mark checkpoint as invalid */
      valid = 0;
    }
    fclose(fs);
  } else {
    /* failed to open file, mark checkpoint as invalid */
    valid = 0;
  }

  /**** change #9 ****/
  /*
      // wait for all files to be closed
      MPI_Barrier(MPI_COMM_WORLD);

      // rank 0 updates the pointer to the latest checkpoint
      FILE* fs = fopen("latest", "w");
      if (fs != NULL) {
```

(continues on next page)

(continued from previous page)

```
        fwrite(checkpoint_dir, ..., fs);
        fclose(fs);
      }
  */

  /**** change #10 ****/
  SCR_Complete_output(valid);
}
```

As shown in change #5, the application must inform SCR when it is starting a new checkpoint by calling `SCR_Start_output()` with the `SCR_FLAG_CHECKPOINT`. The application should provide a name for the checkpoint, and all processes must provide the same name and the same flags values.

The application must inform SCR when it has completed the checkpoint with a corresponding call to `SCR_Complete_output()` as shown in change #10. When calling `SCR_Complete_output()`, each process sets the `valid` flag to indicate whether it wrote all of its checkpoint files successfully. Note how a `valid` variable has been added to track any errors while writing the checkpoint.

SCR manages checkpoint directories, so the `mkdir` operation is removed in change #6. Additionally, the application can rely on SCR to track the latest checkpoint, so the logic to track the latest checkpoint is removed in change #9.

Between the call to `SCR_Start_output()` and `SCR_Complete_output()`, the application must register each of its checkpoint files by calling `SCR_Route_file()` as shown in change #7. As input, the process may provide either an absolute or relative path to its checkpoint file. If given a relative path, SCR internally prepends the current working directory to the path when `SCR_Route_file()` is called. In either case, the fully resolved path must be located somewhere within the prefix directory. If SCR copies the file to the parallel file system, it writes the file to this path. When storing the file in cache, SCR "routes" the file by replacing any leading directory on the file name with a path that points to a cache directory. SCR returns this routed path as output.

As shown in change #8, the application must use the exact string returned by `SCR_Route_file()` to open its checkpoint file.

Restart with SCR

To use SCR for restart, the application can call `SCR_Have_restart` to determine whether SCR has a previous checkpoint loaded. If there is a checkpoint available, the application can call `SCR_Start_restart` to tell SCR that it is initiating a restart operation.

The application must call `SCR_Route_file` to determine the full path and file name to each of its files that it will read during the restart. The calling process can specify either an absolute or relative path in its input file name. If given a relative path, SCR internally prepends the current working directory at the point when `SCR_Route_file()` is called. The fully resolved path must be located somewhere within the prefix directory and it must correspond to a file associated with the particular checkpoint name that SCR returned in `SCR_Start_restart`.

After the application reads its checkpoint files, it must call `SCR_Complete_restart` to indicate that it has completed reading its checkpoint files. If any process fails to read its checkpoint files, `SCR_Complete_restart` returns something other than `SCR_SUCCESS` on all processes and SCR prepares the next most recent checkpoint if one is available. The application can try again with another call to `SCR_Have_restart`.

For backwards compatibility, the application can provide just a file name in `SCR_Route_file` during restart, even if the combination of the current working directory and the provided file name do not specify the correct path on the parallel file system. This usage is deprecated, and it may be not be supported in future releases. Instead it is recommended that one construct the full path to the checkpoint file using information from the checkpoint name returned by `SCR_Start_restart`.

Some example SCR restart code may look like the following

```
void* restart() {
  /* each process reads its state from a file */

  /**** change #12 ****/
  int restarted = 0;
  while (! restarted) {

    /**** change #13 ****/
    int have_restart = 0;
    char checkpoint_dir[SCR_MAX_FILENAME];
    SCR_Have_restart(&have_restart, checkpoint_dir);
    if (! have_restart) {
      /* no checkpoint available from which to restart */
      break;
    }

    /**** change #14 ****/
    SCR_Start_restart(checkpoint_dir);

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

    /**** change #15 ****/
    /*
        // rank 0 reads and broadcasts checkpoint directory name
        char checkpoint_dir[256];
        if (rank == 0) {
          FILE* fs = fopen("latest", "r");
          if (fs != NULL) {
            fread(checkpoint_dir, ..., fs);
            fclose(fs);
          }
        }
        MPI_Bcast(checkpoint_dir, sizeof(checkpoint_dir), MPI_CHAR, ...);
    */

    /**** change #16 ****/
    /* build file name of checkpoint file for this rank */
    char checkpoint_file[256];
    sprintf(checkpoint_file, "%s/rank_%d.ckpt",
      checkpoint_dir, rank
    );

    /**** change #17 ****/
    char scr_file[SCR_MAX_FILENAME];
    SCR_Route_file(checkpoint_file, scr_file);

    /**** change #18 ****/
    /* each rank opens, reads, and closes its file */
    int valid = 1;
    FILE* fs = fopen(scr_file, "r");
    if (fs != NULL) {
      int read_rc = fread(state, ..., fs);
      if (read_rc == 0) {
        /* failed to read file, mark restart as invalid */
        valid = 0;
```

(continues on next page)

(continued from previous page)

```
      }
      fclose(fs);
    } else {
      /* failed to open file, mark restart as invalid */
      valid = 0;
    }

    /**** change #19 ****/
    int rc = SCR_Complete_restart(valid);

    /**** change #20 ****/
    restarted = (rc == SCR_SUCCESS);
  }

  if (restarted) {
    return state;
  } else {
    return new_run_state;
  }
}
```

With SCR, the application can attempt to restart from its most recent checkpoint, and if that fails, SCR loads the next most recent checkpoint. This process continues until the application successfully restarts or exhausts all available checkpoints. To enable this, we create a loop around the restart process, as shown in change #12.

For each attempt, the application must first call `SCR_Have_restart()` to determine whether SCR has a checkpoint available as shown in change #13. If there is a checkpoint, the application calls `SCR_Start_restart()` as shown in change #14 to inform SCR that it is beginning its restart. The application logic to identify the latest checkpoint is removed in change #15, since SCR manages which checkpoint to load. The application should use the checkpoint name returned in `SCR_Start_restart()` to construct the input path to its checkpoint file as shown in change #16. The application obtains the path to its checkpoint file by calling `SCR_Route_file()` in change #17. It uses this path to open the file for reading in change #18. After the process reads each of its checkpoint files, it informs SCR that it has completed reading its data with a call to `SCR_Complete_restart()` in change #19.

When calling `SCR_Complete_restart()`, each process sets the `valid` flag to indicate whether it read all of its checkpoint files successfully. Note how a `valid` variable has been added to track whether the process successfully reads its checkpoint.

As shown in change #20, SCR returns `SCR_SUCCESS` from `SCR_Complete_restart()` if all processes succeeded. If the return code is something other than `SCR_SUCCESS`, then at least one process failed to restart. In that case, SCR loads the next most recent checkpoint if one is available, and the application can call `SCR_Have_restart()` to iterate through the process again.

It is not required for an application to loop on failed restarts, but SCR allows for that. SCR never loads a checkpoint that is known to be incomplete or one that is explicitly marked as invalid, though it is still possible the application will encounter an error while reading those files on restart. If an application fails to restart from a checkpoint, SCR marks that checkpoint as invalid so that it will not attempt to load that checkpoint again in future runs.

It is possible to use the SCR Restart API even if the application must restart from a global file system. For such applications, one should set `SCR_GLOBAL_RESTART=1`. Under this mode, SCR flushes any cached checkpoint to the prefix directory during `SCR_Init`, and it configures its restart operation to use cache bypass mode so that `SCR_Route_file` directs the application to read its files directly from the parallel file system.

Restart without SCR

If the application does not use SCR for restart, it should not make calls to `SCR_Have_restart`, `SCR_Start_restart`, `SCR_Route_file`, or `SCR_Complete_restart` during the restart. Instead, it should access files directly from the parallel file system.

When not using SCR for restart, one should set `SCR_FLUSH_ON_RESTART=1`, which causes SCR to flush any cached checkpoint to the file system during `SCR_Init`. Additionally, one should set `SCR_FETCH=0` to disable SCR from loading a checkpoint during `SCR_Init`. The application can then read its checkpoint from the parallel file system after calling `SCR_Init`.

If the application reads a checkpoint that it previously wrote through SCR, it should call `SCR_Current` after `SCR_Init` to notify SCR which checkpoint that it restarted from. This lets SCR configure its internal state to properly track the ordering of new datasets that the application writes.

If restarting without SCR and if `SCR_Current` is not called, the value of the `SCR_FLUSH` counter will not be preserved between restarts. The counter will be reset to its upper limit with each restart. Thus each restart may introduce some offset in a sequence of periodic SCR flushes.

2.6.2 Building with the SCR library

To compile and link with the SCR library, add the flags shown below to your compile and link lines. The value of the variable `SCR_INSTALL_DIR` should be the path to the installation directory for SCR.

Compile Flags	`-I$(SCR_INSTALL_DIR)/include`
C Dynamic Link Flags	`-L$(SCR_INSTALL_DIR)/lib64 -lscr -Wl,-rpath, $(SCR_INSTALL_DIR)/lib64`
C Static Link Flags	`-L$(SCR_INSTALL_DIR)/lib64 -lscr`
Fortran Dynamic Link Flags	`-L$(SCR_INSTALL_DIR)/lib64 -lscrf -Wl,-rpath, $(SCR_INSTALL_DIR)/lib64`
Fortran Static Link Flags	`-L$(SCR_INSTALL_DIR)/lib64 -lscrf`

Note: On some platforms the default library installation path will be `/lib` instead of `/lib64`.

If Spack was used to build SCR, the `SCR_INSTALL_DIR` can be found with:

```
spack location -i scr
```

2.7 Configure a job

The default SCR configuration suffices for many Linux clusters. However, significant performance improvement or additional functionality may be gained via custom configuration.

2.7.1 Setting parameters

SCR searches the following locations in the following order for a parameter value, taking the first value it finds.

- Environment variables,
- User configuration file,

- Values set with `SCR_Config`,
- System configuration file,
- Compile-time constants.

To find a user configuration file, SCR looks for a file named `.scrconf` in the prefix directory (note the leading dot). Alternatively, one may specify the name and location of the user configuration file by setting the `SCR_CONF_FILE` environment variable at run time, e.g.,:

```
export SCR_CONF_FILE=~/myscr.conf
```

The location of the system configuration file is hard-coded into SCR at build time. This defaults to `/etc/scr/scr.conf`. One may set this using the `SCR_CONFIG_FILE` option with cmake, e.g.,:

```
cmake -DSCR_CONFIG_FILE=/path/to/scr.conf ...
```

To set an SCR parameter in a configuration file, list the parameter name followed by its value separated by an '=' sign. Blank lines are ignored, and any characters following the '#' comment character are ignored. For example, a configuration file may contain something like the following:

```
>>: cat ~/myscr.conf
# set the halt seconds to one hour
SCR_HALT_SECONDS=3600

# set SCR to flush every 20 checkpoints
SCR_FLUSH=20
```

One can include environment variable expressions in SCR configuration parameters. SCR interpolates the value of the environment variable at run time before setting the parameter. This is especially useful for some parameters like storage paths, which may only be defined within the job environment, e.g.,:

```
# set cache directory based on user and jobid
SCR_CACHE_BASE=/dev/shm/$USER/scr.$SLURM_JOBID
```

2.7.2 Group, store, and checkpoint descriptors

SCR must have information about process groups, storage devices, and redundancy schemes. The defaults provide reasonable settings for Linux clusters, but one can define custom settings via group, store, and checkpoint descriptors in configuration files.

SCR must know which processes are likely to fail at the same time (failure groups) and which processes access a common storage device (storage groups). By default, SCR creates a group of all processes in the job called `WORLD` and another group of all processes on the same compute node called `NODE`. If more groups are needed, they can be defined in configuration files with entries like the following:

```
GROUPS=host1  POWER=psu1  SWITCH=0
GROUPS=host2  POWER=psu1  SWITCH=1
GROUPS=host3  POWER=psu2  SWITCH=0
GROUPS=host4  POWER=psu2  SWITCH=1
```

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

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

In the above example, there are four compute nodes: host1, host2, host3, and host4. There are two groups defined: `POWER` and `SWITCH`. Nodes host1 and host2 belong to the same `POWER` group, as do nodes host3 and host4. For the `SWITCH` group, nodes host1 and host3 belong to the same group, as do nodes host2 and host4.

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

```
STORE=/dev/shm      GROUP=NODE   COUNT=1
STORE=/ssd          GROUP=NODE   COUNT=3  FLUSH=PTHREAD
STORE=/dev/persist  GROUP=NODE   COUNT=1  ENABLED=1  MKDIR=0
STORE=/p/lscratcha  GROUP=WORLD
```

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

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

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

Finally, SCR must be configured with redundancy schemes. By default, SCR uses cache bypass mode and writes all datasets to the parallel file system. To specify something different, one must set various SCR configuration parameters to define checkpoint and output descriptors. Example descriptors in a configuration file look like the following:

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

# enable datasets to be stored in cache
SCR_CACHE_BYPASS=0

# the following instructs SCR to run with three checkpoint configurations:
# - save every 8th checkpoint to /ssd using the PARTNER scheme
# - save every 4th checkpoint (not divisible by 8) and any output dataset
#   to /ssd using RS a set size of 8
# - save all other checkpoints (not divisible by 4 or 8) to /dev/shm using XOR with
#   a set size of 16
CKPT=0 INTERVAL=1 GROUP=NODE   STORE=/dev/shm TYPE=XOR     SET_SIZE=16
CKPT=1 INTERVAL=4 GROUP=NODE   STORE=/ssd     TYPE=RS      SET_SIZE=8  SET_FAILURES=3␣
↪OUTPUT=1
CKPT=2 INTERVAL=8 GROUP=SWITCH STORE=/ssd     TYPE=PARTNER BYPASS=1
```

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

To store datasets in cache, one must set SCR_CACHE_BYPASS=0 to disable bypass mode, which is enabled by default. Otherwise all datasets will be written directly to the parallel file system.

Checkpoint descriptor entries are identified by a leading CKPT key. The values of the CKPT keys must be numbered sequentially starting from 0. The INTERVAL key specifies how often a descriptor is to be applied. For each checkpoint, SCR selects the descriptor having the largest interval value that evenly divides the internal SCR checkpoint iteration number. It is necessary that one descriptor has an interval of 1. This key is optional, and it defaults to 1 if not specified. The GROUP key lists the failure group, i.e., the name of the group of processes likely to fail. This key is optional, and it defaults to the value of the SCR_GROUP parameter if not specified. The STORE key specifies the directory in which to cache the checkpoint. This key is optional, and it defaults to the value of the SCR_CACHE_BASE parameter if not specified. The TYPE key identifies the redundancy scheme to be applied. This key is optional, and it defaults to the value of the SCR_COPY_TYPE parameter if not specified. The BYPASS key indicates whether to bypass cache and access data files directly on the parallel file system (1) or whether to store them in cache (0). In either case, redundancy is applied to internal SCR metadata using the specified descriptor settings. This key is optional, and it defaults to the value of the SCR_CACHE_BYPASS parameter if not specified.

Other keys may exist depending on the selected redundancy scheme. For XOR and RS schemes, the SET_SIZE key specifies the minimum number of processes to include in each redundancy set. This defaults to the value of SCR_SET_SIZE if not specified. For RS, the SET_FAILURES key specifies the maximum number of failures to tolerate within each redundancy set. If not specified, this defaults to the value of SCR_SET_FAILURES.

One checkpoint descriptor can be marked with the OUTPUT key. This indicates that the descriptor should be selected to store datasets that the application flags with SCR_FLAG_OUTPUT. The OUTPUT key is optional, and it defaults to 0. If there is no descriptor with the OUTPUT key defined and if the dataset is also a checkpoint, SCR will choose the checkpoint descriptor according to the normal policy. Otherwise, if there is no descriptor with the OUTPUT key defined and if the dataset is not a checkpoint, SCR will use the checkpoint descriptor having interval of 1.

If one does not explicitly define a checkpoint descriptor, the default SCR descriptor can be defined in pseudocode as:

```
CKPT=0 INTERVAL=1 GROUP=$SCR_GROUP STORE=$SCR_CACHE_BASE TYPE=$SCR_COPY_TYPE SET_SIZE=
↪$SCR_SET_SIZE BYPASS=$SCR_CACHE_BYPASS
```

If those parameters are not set otherwise, this defaults to the following:

```
CKPT=0 INTERVAL=1 GROUP=NODE STORE=/dev/shm TYPE=XOR SET_SIZE=8 BYPASS=1
```

2.7.3 SCR parameters

The table in this section specifies the full set of SCR configuration parameters.

Table 1: SCR parameters

Name	Default	Description
SCR_DEBUG	0	Set to 1 or 2 for increasing verbosity levels of debug messages.
SCR_CHECKPOINT_INTERVAL	0	Set to positive number of times SCR_Need_checkpoint should be called before returning 1. This provides a simple way to set a periodic checkpoint frequency within an application.
SCR_CHECKPOINT_SECONDS	0	Set to positive number of seconds to specify minimum time between consecutive checkpoints as guided by SCR_Need_checkpoint.

Continued on next page

Table 1 – continued from previous page

Name	Default	Description
SCR_CHECKPOINT_OVERHEAD	0.0	Set to positive floating-point value to specify maximum percent overhead allowed for checkpointing operations as guided by SCR_Need_checkpoint.
SCR_CNTL_BASE	/dev/shm	Specify the default base directory SCR should use to store its runtime control metadata. The control directory should be in fast, node-local storage like RAM disk.
SCR_HALT_EXIT	0	Whether SCR should call exit() when it detects an active halt condition. When enabled, SCR can exit the job during SCR_Init and SCR_Complete_output after each successful checkpoint. Set to 1 to enable.
SCR_HALT_SECONDS	0	Set to a positive integer to instruct SCR to halt the job if the remaining time in the current job allocation is less than the specified number of seconds.
SCR_GROUP	NODE	Specify name of default failure group.
SCR_COPY_TYPE	XOR	Set to one of: SINGLE, PARTNER, XOR, RS, or FILE.
SCR_CACHE_BASE	/dev/shm	Specify the default base directory SCR should use to cache datasets.
SCR_CACHE_SIZE	1	Set to a non-negative integer to specify the maximum number of checkpoints SCR should keep in cache. SCR will delete the oldest checkpoint from cache before saving another in order to keep the total count below this limit.
SCR_CACHE_BYPASS	1	Specify bypass mode. When enabled, data files are directly read from and written to the parallel file system, bypassing the cache. Even in bypass mode, internal SCR metadata corresponding to the dataset is stored in cache. Set to 0 to direct SCR to store datasets in cache.
SCR_CACHE_PURGE	0	Whether to delete all datasets from cache during SCR_Init. Enabling this setting may be useful for test and development while integrating SCR in an application.
SCR_SET_SIZE	8	Specify the minimum number of processes to include in an redundancy set. Increasing this value may decrease the amount of storage required to cache the dataset. However, higher values may have an increased likelihood of encountering a catastrophic error. Higher values may also require more time to reconstruct lost files from redundancy data.
SCR_SET_FAILURES	2	Specify the number of failures to tolerate in each set while using the RS scheme. Increasing this value enables one to tolerate more failures per set, but it increases redundancy storage and encoding costs.
SCR_PREFIX	$PWD	Specify the prefix directory on the parallel file system where checkpoints should be read from and written to.
SCR_PREFIX_SIZE	0	Specify number of checkpoints to keep in the prefix directory. SCR deletes older checkpoints as new checkpoints are flushed to maintain a sliding window of the specified size. Set to 0 to keep all checkpoints. Checkpoints marked with SCR_FLAG_OUTPUT are not deleted.
SCR_PREFIX_PURGE	0	Set to 1 to delete all datasets from the prefix directory (both checkpoint and output) during SCR_Init.
SCR_CURRENT	N/A	Name of checkpoint to mark as current and attempt to fetch in a new run during SCR_Init.
SCR_DISTRIBUTE	1	Set to 0 to disable cache rebuild during SCR_Init.
SCR_FETCH	1	Set to 0 to disable SCR from fetching files from the parallel file system during SCR_Init.
SCR_FETCH_WIDTH	256	Specify the number of processes that may read simultaneously from the parallel file system.

Continued on next page

Table 1 – continued from previous page

Name	Default	Description
SCR_FLUSH	10	Specify the number of checkpoints between periodic flushes to the parallel file system. Set to 0 to disable periodic flushes.
SCR_FLUSH_ASYNC	0	Set to 1 to enable asynchronous flush methods (if supported).
SCR_FLUSH_TYPE	SYNC	Specify the AXL transfer method. Set to one of: SYNC, PTHREAD, BBAPI, or DATAWARP.
SCR_FLUSH_WIDTH	256	Specify the number of processes that may write simultaneously to the parallel file system.
SCR_FLUSH_ON_RESTART	0	Set to 1 to force SCR to flush datasets during restart. This is useful for applications that restart without using the SCR Restart API. Typically, one should also set SCR_FETCH=0 when enabling this option.
SCR_GLOBAL_RESTART	0	Set to 1 to flush checkpoints to the prefix directory during SCR_Init and internally switch fetch to use cache bypass mode. This is needed by applications that use the SCR Restart API but require a global file system to restart, e.g., because multiple processes read the same file.
SCR_RUNS	1	Specify the maximum number of times the scr_srun command should attempt to run a job within an allocation. Set to -1 to specify an unlimited number of times.
SCR_MIN_NODES	N/A	Specify the minimum number of nodes required to run a job.
SCR_EXCLUDE_NODES	N/A	Specify a set of nodes, using SLURM node range syntax, which should be excluded from runs. This is useful to avoid particular problematic nodes. Nodes named in this list that are not part of a the current job allocation are silently ignored.
SCR_LOG_ENABLE	1	Whether to enable any form of logging of SCR events.
SCR_LOG_TXT_ENABLE	1	Whether to log SCR events to text file in prefix directory at $SCR_PREFIX/.scr/log. SCR_LOG_ENABLE must be set to 1 for this parameter to be active.
SCR_LOG_SYSLOG_ENABLE	1	Whether to log SCR events to syslog. SCR_LOG_ENABLE must be set to 1 for this parameter to be active.
SCR_LOG_DB_ENABLE	0	Whether to log SCR events to MySQL database. SCR_LOG_ENABLE must be set to 1 for this parameter to be active.
SCR_LOG_DB_DEBUG	0	Whether to print MySQL statements as they are executed.
SCR_LOG_DB_HOST	N/A	Hostname of MySQL server
SCR_LOG_DB_NAME	N/A	Name of SCR MySQL database.
SCR_LOG_DB_USER	N/A	Username of SCR MySQL user.
SCR_LOG_DB_PASS	N/A	Password for SCR MySQL user.
SCR_MPI_BUF_SIZE	1310720	Specify the number of bytes to use for internal MPI send and receive buffers when computing redundancy data or rebuilding lost files.
SCR_FILE_BUF_SIZE	1048576	Specify the number of bytes to use for internal buffers when copying files between the parallel file system and the cache.
SCR_WATCHDOG_TIMEOUT	N/A	Set to the expected time (seconds) for checkpoint writes to in-system storage (see *Catch a hanging job*).
SCR_WATCHDOG_TIMEOUT_PFS	N/A	Set to the expected time (seconds) for checkpoint writes to the parallel file system (see *Catch a hanging job*).

2.8 Run a job

In addition to the SCR library, one may optionally include SCR commands in their job script. These commands are most useful on systems where failures are common. The SCR commands prepare the cache, scavenge files from cache to the parallel file system, and check that the scavenged dataset is complete among other things. The commands also

automate the process of relaunching a job after failure.

These commands are located in the `/bin` directory where SCR is installed. There are numerous SCR commands. Any command not mentioned in this document is not intended to be executed by users.

For best performance, one should: 1) inform the batch system that the allocation should remain available even after a failure and 2) replace the command to execute the application with an SCR wrapper script. The precise set of options and commands to use depends on the system resource manager.

2.8.1 Supported platforms

At the time of this writing, SCR supports specific combinations of resource managers and job launchers. The descriptions for using SCR in this section apply to these specific configurations, however the following description is helpful to understand how to run SCR on any system. Please contact us for help in porting SCR to other platforms. (See Section *Support and Additional Information* for contact information).

2.8.2 Jobs and job steps

First, we differentiate between a *job allocation* and a *job step*. Our terminology originates from the SLURM resource manager, but the principles apply generally across SCR-supported resource managers.

When a job is scheduled resources on a system, the batch script executes inside of a job allocation. The job allocation consists of a set of nodes, a time limit, and a job id. The job id can be obtained by executing the `squeue` command on SLURM, the `apstat` command on ALPS, and the `bjobs` command on LSF.

Within a job allocation, a user may run one or more job steps, each of which is invoked by a call to `srun` on SLURM, `aprun` on ALPS, or `mpirun` on LSF. Each job step is assigned its own step id. On SLURM, within each job allocation, job step ids start at 0 and increment with each issued job step. Job step ids can be obtained by passing the `-s` option to `squeue`. A fully qualified name of a SLURM job step consists of: `jobid.stepid`. For instance, the name `1234.5` refers to step id 5 of job id 1234. On ALPS, each job step within an allocation has a unique id that can be obtained through `apstat`.

2.8.3 Ignoring node failures

Before running an SCR job, it is recommended to configure the job allocation to withstand node failures. By default, most resource managers terminate the job allocation if a node fails, however SCR requires the job allocation to remain active in order to restart the job or to scavenge files. To enable the job allocation to continue past node failures, one must specify the appropriate flags from the table below.

SCR job allocation flags

MOAB batch script	`#MSUB -l resfailpolicy=ignore`
MOAB interactive	`qsub -I ... -l resfailpolicy=ignore`
SLURM batch script	`#SBATCH --no-kill`
SLURM interactive	`salloc --no-kill ...`
LSF batch script	`#BSUB -env "all, LSB_DJOB_COMMFAIL_ACTION=KILL_TASKS"`
LSF interactive	`bsub -env "all, LSB_DJOB_COMMFAIL_ACTION=KILL_TASKS" ...`

2.8.4 The SCR wrapper script

The easiest way to integrate SCR into a batch script is to set some environment variables and to replace the job run command with an SCR wrapper script. The SCR wrapper script includes logic to restart an application within an job

allocation, and it scavenges files from cache to the parallel file system at the end of an allocation.:

```
SLURM:  scr_srun [srun_options]  <prog> [prog_args ...]
ALPS:   scr_aprun [aprun_options] <prog> [prog_args ...]
LSF:    scr_mpirun [mpirun_options] <prog> [prog_args ...]
```

The SCR wrapper script must run from within a job allocation. Internally, the command must know the prefix directory. By default, it uses the current working directory. One may specify a different prefix directory by setting the `SCR_PREFIX` parameter.

It is recommended to set the `SCR_HALT_SECONDS` parameter so that the job allocation does not expire before datasets can be flushed (Section *Halt a job*).

By default, the SCR wrapper script does not restart an application after the first job step exits. To automatically restart a job step within the current allocation, set the `SCR_RUNS` environment variable to the maximum number of runs to attempt. For an unlimited number of attempts, set this variable to `-1`.

After a job step exits, the wrapper script checks whether it should restart the job. If so, the script sleeps for some time to give nodes in the allocation a chance to clean up. Then it checks that there are sufficient healthy nodes remaining in the allocation. By default, the wrapper script assumes the next run requires the same number of nodes as the previous run, which is recorded in a file written by the SCR library. If this file cannot be read, the command assumes the application requires all nodes in the allocation. Alternatively, one may override these heuristics and precisely specify the number of nodes needed by setting the `SCR_MIN_NODES` environment variable to the number of required nodes.

For applications that cannot invoke the SCR wrapper script as described here, one should examine the logic contained within the script and duplicate the necessary parts in the job batch script. In particular, one should invoke `scr_postrun` for scavenge support.

2.8.5 Example batch script for using SCR restart capability

An example SLURM batch script with `scr_srun` is shown below

```
#!/bin/bash
#SBATCH --partition pbatch
#SBATCH --nodes 66
#SBATCH --no-kill

# above, tell SLURM to not kill the job allocation upon a node failure
# also note that the job requested 2 spares -- it uses 64 nodes but allocated 66

# specify where datasets should be written
export SCR_PREFIX=/parallel/file/system/username/run1

# instruct SCR to flush to the file system every 20 checkpoints
export SCR_FLUSH=20

# halt if there is less than an hour remaining (3600 seconds)
export SCR_HALT_SECONDS=3600

# attempt to run the job up to 3 times
export SCR_RUNS=3

# run the job with scr_srun
scr_srun -n512 -N64 ./my_job
```

2.9 Halt a job

When SCR is configured to write datasets to cache, one needs to take care when terminating a job early so that SCR can copy datasets from cache to the parallel file system before the job allocation ends. This section describes methods to cleanly halt a job, including detection and termination of a hanging job.

2.9.1 scr_halt and the halt file

The recommended method to stop an SCR application is to use the `scr_halt` command. The command must be run from within the prefix directory, or otherwise, the prefix directory of the target job must be specified as an argument.

A number of different halt conditions can be specified. In most cases, the `scr_halt` command communicates these conditions to the running application via the `halt.scr` file, which is stored in the hidden `.scr` directory within the prefix directory. The application can determine when to exit by calling `SCR_Should_exit`.

Additionally, one can set `SCR_HALT_EXIT=1` to configure SCR to exit the job if it detects an active halt condition. In that case, the SCR library reads the halt file when the application calls `SCR_Init` and during `SCR_Complete_output` after each complete checkpoint. If a halt condition is satisfied, all tasks in the application call `exit`.

2.9.2 Halt after next checkpoint

You can instruct an SCR job to halt after completing its next successful checkpoint:

```
scr_halt
```

To run `scr_halt` from outside of a prefix directory, specify the target prefix directory like so:

```
scr_halt /p/lscratcha/user1/simulation123
```

You can instruct an SCR job to halt after completing some number of checkpoints via the `--checkpoints` option. For example, to instruct a job to halt after 10 more checkpoints, use the following:

```
scr_halt --checkpoints 10
```

If the last of the checkpoints is unsuccessful, the job continues until it completes a successful checkpoint. This ensures that SCR has a successful checkpoint to flush before it halts the job.

2.9.3 Halt before or after a specified time

It is possible to instruct an SCR job to halt *after* a specified time using the `--after` option. The job will halt on its first successful checkpoint after the specified time. For example, you can instruct a job to halt after "12:00pm today" via:

```
scr_halt --after '12:00pm today'
```

It is also possible to instruct a job to halt *before* a specified time using the `--before` option. For example, you can instruct a job to halt before "8:30am tomorrow" via:

```
scr_halt --before '8:30am tomorrow'
```

For the "halt before" condition to be effective, one must also set the SCR_HALT_SECONDS parameter. When SCR_HALT_SECONDS is set to a positive number, SCR checks how much time is left before the specified time limit. If the remaining time in seconds is less than or equal to SCR_HALT_SECONDS, SCR halts the job. The value of SCR_HALT_SECONDS does not affect the "halt after" condition.

It is highly recommended that SCR_HALT_SECONDS be set so that the SCR library can impose a default "halt before" condition using the end time of the job allocation. This ensures the latest checkpoint can be flushed before the allocation is lost.

It is important to set SCR_HALT_SECONDS to a value large enough that SCR has time to completely flush (and rebuild) files before the allocation expires. Consider that a checkpoint may be taken just *before* the remaining time is less than SCR_HALT_SECONDS. If a code checkpoints every X seconds and it takes Y seconds to flush files from the cache and rebuild, set SCR_HALT_SECONDS = X + Y + Delta, where Delta is some positive value to provide additional slack time.

One may also set the halt seconds via the --seconds option to scr_halt. Using the scr_halt command, one can set, change, and unset the halt seconds on a running job.

NOTE: If any scr_halt commands are specified as part of the batch script before the first run starts, one must then use scr_halt to set the halt seconds for the job rather than the SCR_HALT_SECONDS parameter. The scr_halt command creates the halt file, and if a halt file exists before a job starts to run, SCR ignores any value specified in the SCR_HALT_SECONDS parameter.

2.9.4 Halt immediately

Sometimes, you need to halt an SCR job immediately, and there are two options for this. You may use the --immediate option:

```
scr_halt --immediate
```

This command first updates the halt file, so that the job will not be restarted once stopped. Then, it kills the current run.

If for some reason the --immediate option fails to work, you may manually halt the job.[1] First, issue a simple scr_halt so the job will not restart, and then manually kill the current run using mechanisms provided by the resource manager, e.g., scancel for SLURM and apkill for ALPS. When using mechanisms provided by the resource manager to kill the current run, be careful to cancel the job step and not the job allocation. Canceling the job allocation destroys the cache.

For SLURM, to get the job step id, type: squeue -s. Then be sure to include the job id *and* step id in the scancel argument. For example, if the job id is 1234 and the step id is 5, then use the following commands:

```
scr_halt
scancel 1234.5
```

Do *not* just type scancel 1234 – be sure to include the job step id.

For ALPS, use apstat to get the apid of the job step to kill. Then, follow the steps as described above: execute scr_halt followed by the kill command apkill <apid>.

2.9.5 Catch a hanging job

If an application hangs, SCR may not be given the chance to copy files from cache to the parallel file system before the allocation expires. To avoid losing significant work due to a hang, SCR attempts to detect if a job is hanging, and if so, SCR attempts to kill the job step so that it can be restarted in the allocation.

[1] On Cray/ALPS, scr_halt --immediate is not yet supported. The alternate method described in the text must be used instead.

On some systems, SCR employs the `io-watchdog` library for this purpose. For more information on this tool, see http://code.google.com/p/io-watchdog.

On systems where `io-watchdog` is not available, SCR uses a generic mechanism based on the expected time between checkpoints as specified by the user. If the time between checkpoints is longer than expected, SCR assumes the job is hanging. Two SCR parameters determine how many seconds should pass between I/O phases in an application, i.e. seconds between consecutive calls to `SCR_Start_output`. These are `SCR_WATCHDOG_TIMEOUT` and `SCR_WATCHDOG_TIMEOUT_PFS`. The first parameter specifies the time to wait when SCR writes checkpoints to in-system storage, e.g. SSD or RAM disk, and the second parameter specifies the time to wait when SCR writes checkpoints to the parallel file system. The reason for the two timeouts is that writing to the parallel file system generally takes much longer than writing to in-system storage, and so a longer timeout period is useful in that case.

When using this feature, be careful to check that the job does not hang near the end of its allocation time limit, since in this case, SCR may not kill the run with enough time before the allocation ends. If you suspect the job to be hanging and you deem that SCR will not kill the run in sufficient time, manually cancel the run as described above.

2.9.6 Combine, list, change, and unset halt conditions

It is possible to specify multiple halt conditions. To do so, simply list each condition in the same `scr_halt` command or issue several commands. For example, to instruct a job to halt after 10 checkpoints or before "8:30am tomorrow", which ever comes earlier, you could issue the following command:

```
scr_halt --checkpoints 10 --before '8:30am tomorrow'
```

The following sequence also works:

```
scr_halt --checkpoints 10
scr_halt --before '8:30am tomorrow'
```

You may list the current settings in the halt file with the `--list` option, e.g.,:

```
scr_halt --list
```

You may change a setting by issuing a new command to overwrite the current value.

Finally, you can unset some halt conditions by prepending `unset-` to the option names. See the `scr_halt` man page for a full listing of unset options. For example, to unset the "halt before" condition on a job, type the following:

```
scr_halt --unset-before
```

2.9.7 Remove the halt file

Sometimes, especially during testing, you may want to run in an existing allocation after halting a previous run. When SCR detects a halt file with a satisfied halt condition, it immediately exits. This is the desired effect when trying to halt a job, however this mechanism also prevents one from intentionally running in an allocation after halting a previous run. Along these lines, know that SCR registers a halt condition whenever the application calls `SCR_Finalize`.

When there is a halt file with a satisfied halt condition, a message is printed to `stdout` to indicate why SCR is halting. To run in such a case, first remove the satisfied halt conditions. You can unset the conditions or reset them to appropriate values. Another approach is to remove the halt file via the `--remove` option. This deletes the halt file, which effectively removes all halt conditions. For example, to remove the halt file from a job, type:

```
scr_halt --remove
```

2.10 Manage datasets

SCR records the status of datasets that are on the parallel file system in the `index.scr` file. This file is written to the hidden `.scr` directory within the prefix directory. The library updates the index file as an application runs and during scavenge operations.

While restarting a job, the SCR library reads the index file during `SCR_Init` to determine which checkpoints are available. The library attempts to restart with the most recent checkpoint and works backwards until it successfully fetches a valid checkpoint. SCR does not fetch any checkpoint marked as "incomplete" or "failed". A checkpoint is marked as incomplete if it was determined to be invalid during the flush or scavenge. Additionally, the library marks a checkpoint as failed if it detected a problem during a previous fetch attempt (e.g., detected data corruption). In this way, the library avoids invalid or problematic checkpoints.

One may list or modify the contents of the index file via the `scr_index` command. The `scr_index` command must run within the prefix directory, or otherwise, one may specify a prefix directory using the `--prefix` option. The default behavior of `scr_index` is to list the contents of the index file, e.g.:

```
>>: scr_index
DSET VALID FLUSHED              CUR NAME
  18 YES   2014-01-14T11:26:06   * ckpt.18
  12 YES   2014-01-14T10:28:23     ckpt.12
   6 YES   2014-01-14T09:27:15     ckpt.6
```

When listing datasets, `scr_index` lists a field indicating whether the dataset is valid, the time it was flushed to the parallel file system, and finally the dataset name.

One checkpoint may also be marked as "current". When restarting a job, the SCR library starts from the current dataset and works backwards. The current dataset is denoted with a leading `*` character in the `CUR` column. One can change the current checkpoint using the `--current` option, providing the dataset name as an argument:

```
scr_index --current ckpt.12
```

If no dataset is marked as current, SCR starts with most recent checkpoint that is valid.

One may drop entries from the index file using the `--drop` option. This operation does not delete the corresponding dataset files. It only drops the entry from the `index.scr` file:

```
scr_index --drop ckpt.50
```

This is useful if one deletes a dataset from the parallel file system and then wishes to update the index.

If an entry is removed inadvertently, one may add it back with:

```
scr_index --add ckpt.50
```

This requires all SCR metadata files to exist in their associated `scr.dataset` subdirectory within the hidden `.scr` directory within the prefix directory.

In most cases, the SCR library or the SCR commands add all necessary entries to the index file. However, there are cases where they may fail. In particular, if the `scr_postrun` command successfully scavenges a dataset but the resource allocation ends before the command can rebuild missing files, an entry may be missing from the index file. In such cases, one may manually add the corresponding entry using the `--build` option.

When adding a new dataset to the index file, the `scr_index` command checks whether the files in a dataset constitute a complete and valid set. It rebuilds missing files if there are sufficient redundant data, and it writes the `summary.scr` file for the dataset if needed. One must provide the SCR dataset id as an argument. To obtain the SCR dataset id value, lookup the trailing integer on the names of `scr.dataset` subdirectories in the hidden `.scr` directory within the prefix directory:

```
scr_index --build 50
```

Bibliography

[Vaidya] "A Case for Two-Level Recovery Schemes", Nitin H. Vaidya, IEEE Transactions on Computers, 1998, http://doi.ieeecomputersociety.org/10.1109/12.689645.

[Patterson] "A Case for Redundant Arrays of Inexpensive Disks (RAID)", D. Patterson, G. Gibson, and R. Katz, Proc. of 1988 ACM SIGMOD Conf. on Management of Data, 1988, http://web.mit.edu/6.033/2015/wwwdocs/papers/Patterson88.pdf.

[Gropp] "Providing Efficient I/O Redundancy in MPI Environments", William Gropp, Robert Ross, and Neill Miller, Lecture Notes in Computer Science, 3241:7786, September 2004. 11th European PVM/MPI Users Group Meeting, 2004, http://www.mcs.anl.gov/papers/P1178.pdf.