

The Scalable Checkpoint / Restart (SCR) Library

Version 1.1-5 User Manual

Adam Moody (moody20@llnl.gov)
Livermore Computing Center,
Lawrence Livermore National Laboratory,
Livermore, CA 94551, USA

December 21, 2009

Abstract

The Scalable Checkpoint / Restart (SCR) library provides an interface that codes may use to write out and read in per-process checkpoint files in a scalable fashion. In the current implementation, checkpoint files are cached in storage local to the compute nodes, and a redundancy scheme is applied such that the cached files can be recovered after a failure. This technique provides scalable aggregate bandwidth and uses storage resources that are fully dedicated to the job. This approach addresses the two common drawbacks of checkpointing a large-scale application to a shared parallel file system, namely, limited bandwidth and file system contention. In fact, on current platforms, SCR scales linearly with the number of compute nodes. It has been benchmarked as high as 720GB/s on 1094 nodes of Atlas at Lawrence Livermore National Laboratory (LLNL). At this scale, it is two orders of magnitude faster than the parallel file system. The library also supports the use of spare nodes such that it is possible to rebuild the cached checkpoint files and restart a job after a failed node without the need to write out and read in the checkpoint files via the parallel file system.

The SCR library is written in C, and it currently only provides a C interface. It is well-tested on Linux clusters with RAM disk and local hard drives. It has been used in production at LLNL since late 2007 on Linux / AMD Opteron / Infiniband clusters. The current implementation assumes SLURM [1] is used as the resource manager. The library is designed to be portable to run on other platforms and resource managers. It is an open source project under a BSD license hosted at: <http://sourceforge.net/projects/scalablecr>.

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

Contents

1	Introduction	4
2	Assumptions	4
3	Implementation details	6
3.1	Scalable checkpoint	6
3.1.1	Local	7
3.1.2	Partner	7
3.1.3	XOR	7
3.1.4	Summary	9
3.2	Scalable restart	9
3.3	Multiple processes per node, multiple files per process, and arbitrary file sizes	9
3.4	Catastrophic failures	9
3.5	SCR checkpoint directories	13
4	Integrating SCR into an application	15
4.1	The SCR API	15
4.1.1	SCR_Init	15
4.1.2	SCR_Finalize	15
4.1.3	SCR_Need_checkpoint	15
4.1.4	SCR_Start_checkpoint	15
4.1.5	SCR_Complete_checkpoint	15
4.1.6	SCR_Route_file	16
4.2	Integrating the SCR API	16
4.2.1	Init/Finalize	16
4.2.2	Checkpoint	17
4.2.3	Restart	18
4.3	Building with the SCR library	19
4.4	Using the SCR interpose library	19
5	Integrating SCR into a job script	21
5.1	Distinguishing SLURM job steps from job allocations	21
5.2	Configuring the job allocation	21
5.3	SCR commands	21
5.3.1	scr_srun	22
5.3.2	scr_prerun	22
5.3.3	scr_postrun	23
5.3.4	scr_check_complete	23
5.3.5	scr_halt	23
5.3.6	Other commands	24
5.4	Example MOAB job script with SCR	24
6	Instructing an SCR job to halt	24
6.1	scr_halt and the halt file	24
6.2	Halt after next (X) checkpoint(s)	25
6.3	Halt before or after a specified time	25
6.4	Halt immediately	25
6.5	Catching a hanging job	26
6.6	Combining, listing, changing, and unsetting halt conditions	26
6.7	Removing the halt file	27

7	Environment variables	27
8	Acknowledgments	28

1 Introduction

Compared to small-scale jobs, large-scale jobs face an increased likelihood of encountering process and compute node failures during their run. To handle more frequent failures, large-scale jobs must checkpoint more frequently. Typically, the checkpoint files are written to a parallel file system.

However, there are two drawbacks with checkpointing a large-scale application to a parallel file system. Firstly, large-scale applications often must checkpoint large data sets for which the parallel file system delivers a relatively small amount of bandwidth. Secondly, the parallel file system is typically shared among multiple compute clusters, and so it may be busy servicing other applications when the job needs to checkpoint. Together, the bandwidth bottleneck and contention for the file system effectively idle the large-scale job while it waits for the storage resource.

The Scalable Checkpoint / Restart (SCR) library addresses both of these problems. SCR caches application checkpoint files in storage located on the compute nodes allocated to the job. With this approach, checkpoint storage bandwidth scales linearly with the job size, since each additional compute node serves as an additional storage resource. Also, since the compute nodes are dedicated to the job, the storage resources are dedicated to the application, so there is no contention from other jobs. In fact, this approach benefits the compute center as a whole, because by directing checkpoint data to storage on the compute nodes, a significant load is removed from the shared parallel file system.

There are design challenges associated with caching checkpoint files on the compute nodes. The compute nodes may fail at any time, and so the associated storage resource on a node may become inaccessible at any time. To deal with such failures, the library employs redundancy schemes to recover lost data. Also, the checkpoint cache must be flushed to permanent storage before the job ends. As such, a set of utility scripts must run after the job to drain the latest checkpoint set from the compute node storage to the parallel file system. Finally, there are catastrophic failures from which the library cannot recover. To withstand such failures, it is necessary to occasionally write checkpoint sets out to the parallel file system during the run. However, the frequency with which these writes must be made can be greatly reduced.

The current implementation achieves exceptional storage bandwidth and delivers near-perfect scalability on existing clusters, see Figure 1. Because applications can checkpoint so much faster with SCR, they can afford to checkpoint more frequently, and thus recover more work upon a failure. In fact, assuming the redundancy scheme is not violated in a given failure, the library is capable of supporting a restart from the cached checkpoint set. In this case, the application may restart in the same resource allocation without needing to write the checkpoint files to the parallel file system and read them back. In essence, SCR is a production-level implementation of a two-level checkpoint system of the type analyzed by Vaidya in [2].

2 Assumptions

The current SCR implementation makes a number of assumptions. If an application does not meet these assumptions, it cannot use the current implementation. If this is the case, or if you have any questions, please notify the developers. The goal is to expand the implementation to support a large number of applications.

1. The code must be an MPI application that uses two or more compute nodes.
2. The code must take globally-coordinated checkpoints written primarily as a file per process.
3. A process is only guaranteed access to its own checkpoint files, i.e., in general, a process is not able to access a checkpoint file written by another process. Note that this may limit the effectiveness of the library for codes that are capable of restarting from a checkpoint with a different number of processes than were used to write the checkpoint.
4. For each checkpoint, it must be possible to store the checkpoint files from all processes in the same directory.
5. There is no support for subdirectories within a checkpoint directory; only a single flat file space.

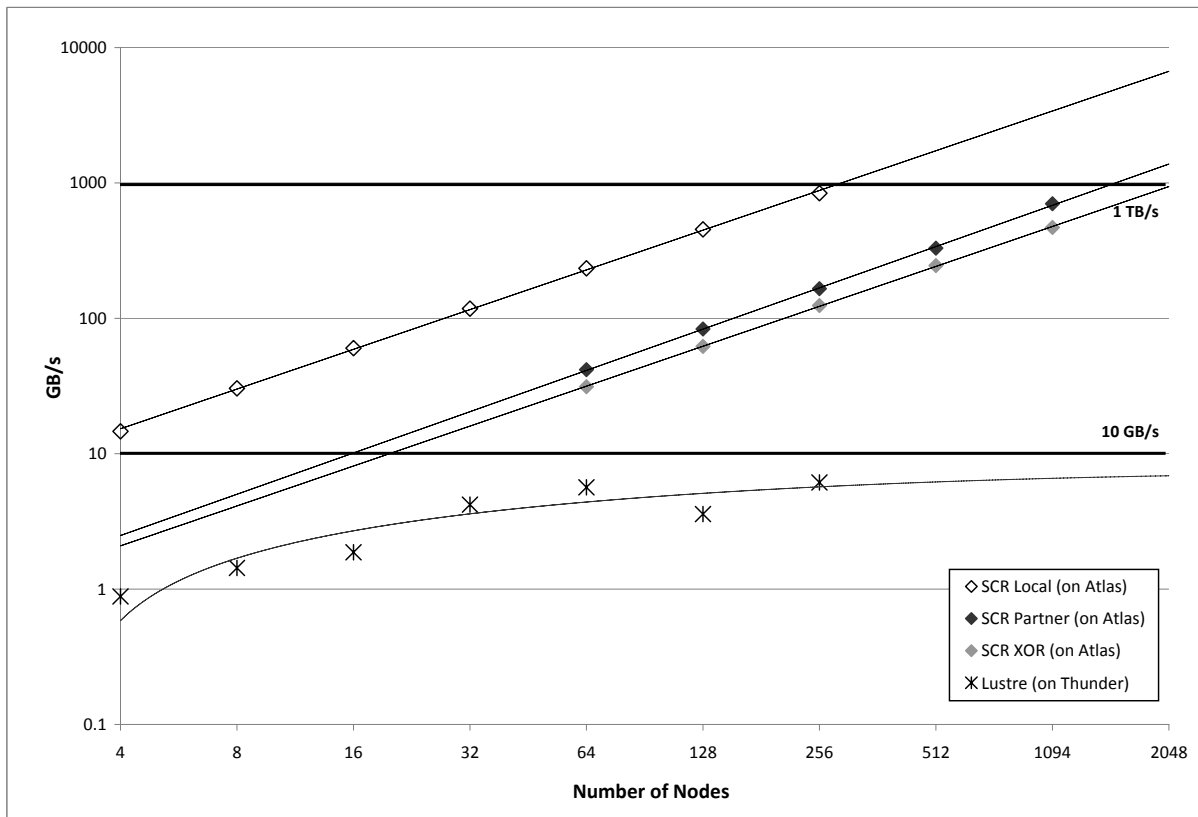


Figure 1: Aggregate I/O bandwidth

6. Checkpoint files cannot contain data that span multiple checkpoints. In particular, there is no support for appending data of the current checkpoint to a file containing data from a previous checkpoint. Each checkpoint file set must be self-contained.
7. On some systems, checkpoints are cached in RAM disk. This restricts usage of SCR on those machines to applications whose memory footprint leaves sufficient room to store checkpoint file data in memory simultaneously with the running application. The amount of storage needed depends on the redundancy scheme used. See Section 3.1 for details.
8. SCR maintains a set of meta data files, which it stores in the same directory as the application checkpoint files. The application must allow for these SCR meta data files to coexist with its own files.
9. To use the scalable restart capability, a job must be restarted with the same number of processes and checkpoint redundancy configuration as were used to write the checkpoint.
10. SCR occasionally flushes files in the checkpoint cache to a checkpoint directory on the parallel file system. The application job script must specify the prefix directory where the checkpoint directories should be placed. SCR defines and creates the checkpoint directories within this prefix directory. See Section 3.5 for details.
11. Time limits should be imposed so that the SCR library has sufficient time to flush checkpoint files from the checkpoint cache to the checkpoint directory before the job allocation expires. Additionally, care should be taken so that the run does not stop in the middle of a checkpoint. See Section 6 for details.

3 Implementation details

The basic concept behind the SCR library is to cache application checkpoint files in fast, scalable storage on the compute cluster itself rather than write them to the parallel file system. Because this cache is fast and scalable, the application may save its state frequently at little cost. Ideally, only at the end of a run or only when a failure occurs does the application need to incur the cost of writing its checkpoint set out to the parallel file system. In practice, checkpoint files must be written to the parallel file system with some moderate frequency to withstand catastrophic failures.

In addition to scalable checkpoints, the library also supports scalable restarts. In many failure cases, SCR enables the application to restart from the cached checkpoint set. This facility can be used with great effect when allocating spare nodes in a job allocation to fill in for failed nodes.

For simplicity, in the following sections, it is assumed that there is one MPI process per node, each process writes exactly one file per checkpoint, and files from all processes are equal in size. The library supports more general cases (see Section 3.3), however, these assumptions simplify the following discussion.

3.1 Scalable checkpoint

In the current implementation, checkpoint files are written to storage on the compute nodes. Because the compute nodes may fail, it is generally necessary to cache the checkpoint data redundantly in order to recover lost data upon a node failure. The SCR library implements three redundancy schemes which trade off performance, storage space, and reliability:

- **Local** - each checkpoint file is written to storage on the local node
- **Partner** - each checkpoint file is written to storage on the local node, and a full copy of each file is written to storage on a partner node
- **XOR** - each checkpoint file is written to storage on the local node, XOR parity data are computed using checkpoint files from a set of nodes, and the parity data are stored among the set.

3.1.1 Local

With **Local**, SCR simply writes checkpoint files to storage on the local node. As such, it requires sufficient storage space to write the maximum checkpoint file size. This scheme is fast, but it cannot withstand node failures, such as when the node loses power or loses its network connection. However, it can withstand any failure that kills the application process but leaves the node intact. This failure class includes common application bugs and system communication or file I/O errors that abort the application process but leave the rest of the system intact.

To use **Local**, set the `SCR_COPY_TYPE` environment variable to “**LOCAL**”.

3.1.2 Partner

With **Partner**, SCR writes checkpoint files to storage on the local node, and it also copies each checkpoint file to storage on a partner node. This scheme is slower and requires twice the storage space as **Local**, however it is capable of withstanding node failures. In fact, it can withstand failures of multiple nodes, so long as a given node and its partner do not fail simultaneously.

To pick a partner node, nodes are logically aligned in order of increasing MPI rank of the process on each node. Each node selects the node D hops up in order of increasing MPI rank to be its partner. The last node wraps back to the first node to form a ring. The hop distance, D , defaults to a value of 1, but it can be configured by setting the `SCR_PARTNER_DISTANCE` environment variable.

On many systems, `SCR_PARTNER_DISTANCE=1` generally provides the best performance, because this often amounts to picking a partner node that is physically close to the source node in the network, which in turn reduces network contention. However, nodes that are physically located nearby each other may be more likely to fail simultaneously, such as when a common network switch fails or a power breaker feeding a section of the cluster shuts off. Thus, larger partner distances may be chosen to improve reliability at the cost of reducing performance.

To use **Partner**, set the `SCR_COPY_TYPE` environment variable to “**PARTNER**”.

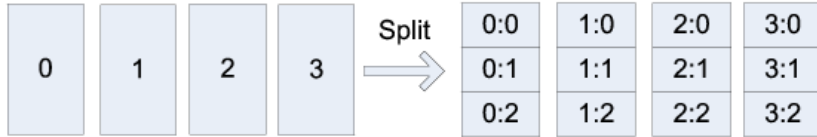
3.1.3 XOR

With **XOR**, SCR first assigns nodes to disjoint XOR sets, each of size N . Then, the nodes in the same set logically split and pad their checkpoint files and collectively compute a reduce-scatter operation using bit-wise XOR, as shown in Figure 2. This effectively computes an XOR parity file, splits it into N equal-sized segments, and scatters one segment to each node. Finally, each node writes its XOR segment along side its checkpoint file in local storage. This algorithm is based on the work found in [3], which in turn was inspired by RAID5 [4]. This scheme provides sufficient redundancy such that it can withstand multiple node failures so long as two nodes from the same XOR set do not fail simultaneously. If a node fails, its checkpoint file can be reconstructed using the remaining $N - 1$ checkpoint files and $N - 1$ XOR segments.

Computationally, this scheme 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 file segment, where the segment size is roughly $1/(N - 1)$ the size of a checkpoint file. Larger XOR sets demand less storage but also increase the probability that two nodes in the same set will fail simultaneously. Larger sets can also increase the cost of reconstructing lost files in the event of a failure.

Ideally, the XOR set size, N , should be chosen such that it evenly divides the number of nodes in the job. If N does not evenly divide the number of nodes, SCR will divide the job into as many sets of size N as possible, and then it will build a set from the remaining k nodes, where $k < N$. SCR will still function in this case, however, be aware that the XOR segments in the remainder set will be $1/(k - 1)$ the size of the checkpoint file, which will be larger than the segment size used in the sets of size N .

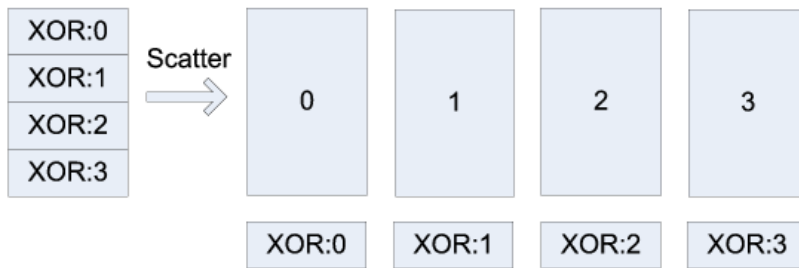
To use **XOR**, set the `SCR_COPY_TYPE` environment variable to “**XOR**”. The XOR set size currently defaults to a size of 8, however this can be adjusted by setting the `SCR_XOR_SIZE` environment variable. The `SCR_PARTNER_DISTANCE` variable can be used to specify the number of hops between nodes selected to be in the same XOR set.



Logically split checkpoint files from ranks on N different nodes into N-1 segments



Logically insert alternating zero-padded blocks and reduce



Scatter XOR segments to different nodes

Figure 2: XOR reduce of checkpoint files and scatter of segments

3.1.4 Summary

Table 1 summarizes details about the different redundancy schemes, and Figure 3 illustrates the different storage patterns used.

Table 1: Redundancy scheme summary

SCR_COPY_TYPE	Relevant variables	Storage required for B bytes
LOCAL	None	B
PARTNER	SCR_PARTNER_DISTANCE	$2 * B$
XOR	SCR_XOR_SIZE, SCR_PARTNER_DISTANCE	$B + B/(N - 1)$, where N is the size of the XOR set

3.2 Scalable restart

So long as a failure does not violate the redundancy scheme, an application can restart within the same job allocation using the cached checkpoint files. This saves the application from having to write out and read in its checkpoint set via the parallel file system. In addition, SCR provides support for the use of spare nodes. With this capability, a job can allocate more nodes than it needs and use the extra nodes to fill in for any failed nodes during a restart, provided there are enough spares.

By default, when a job starts, SCR checks whether all checkpoint files exist or can be recovered in the cache. If so, SCR recovers and distributes the checkpoint files according to the (possibly different) MPI rank-to-node mapping. Then, SCR immediately rebuilds the redundancy data to handle another failure. To disable this feature, set the SCR_DISTRIBUTE environment variable to 0.

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

3.3 Multiple processes per node, multiple files per process, and arbitrary file sizes

The current implementation supports multiple processes per node, including a varying number of processes per node. The following algorithm is applied to select partner nodes or XOR sets in such cases. For each node, the processes on the node are ordered via increasing MPI rank. According to this order, each process on the node is assigned a unique *level* number which counts up from 0. Then, the processes of the job are split into groups such that all processes of the same level are in the same group. This splits the job into groups where each group has at most one process per node. Within each such level group, nodes are then ordered via increasing MPI rank of the process on the node. Finally, the partner and XOR set selection algorithms described in the previous sections are applied internally to each level group using this node ordering.

It is straight-forward to handle **Local** and **Partner** when each process writes an arbitrary number of files with arbitrary sizes. This case is more interesting, however, with **XOR**. When using **XOR**, the files a process writes during a checkpoint are logically concatenated to form a single, larger file. This logical file may have no length if the process wrote no files during the checkpoint. Then, processes in the same XOR set compute the maximum logical file size across their set. This maximum file size is used to determine the size of the XOR segment. Finally, to compute the XOR data, each process pads the end of its logical file with 0 up to the maximum file size.

3.4 Catastrophic failures

There are some failures from which the SCR library cannot recover. In such cases, the application will be forced to fall back to the last checkpoint set successfully written to the parallel file system. Here are some examples.

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

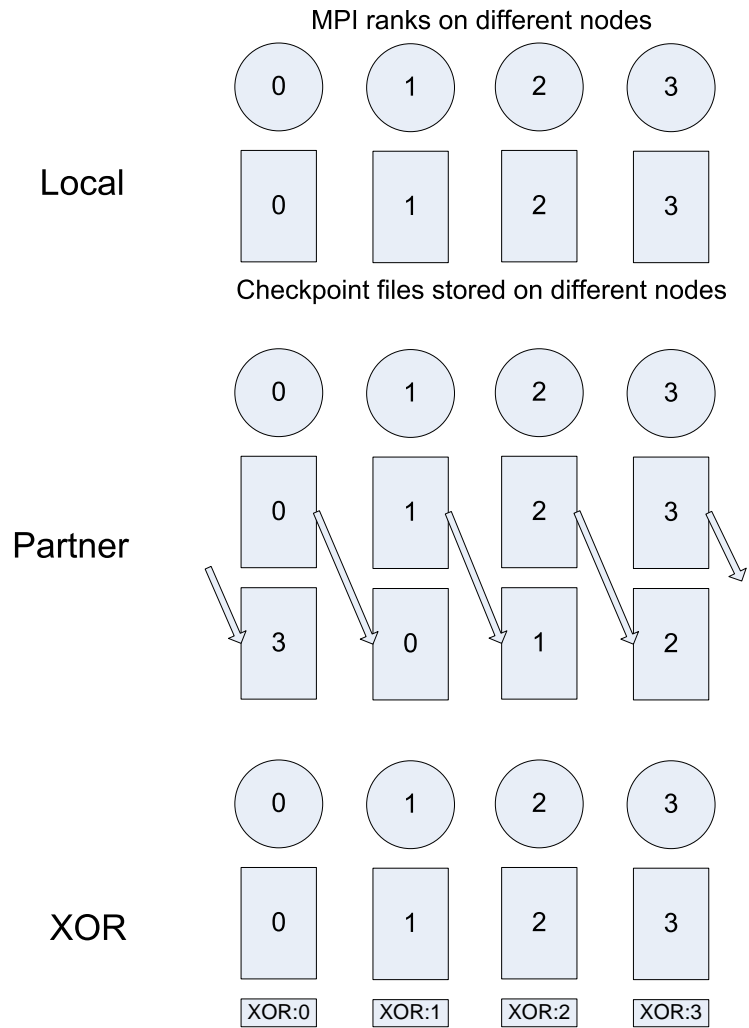


Figure 3: Storage pattern of different redundancy schemes

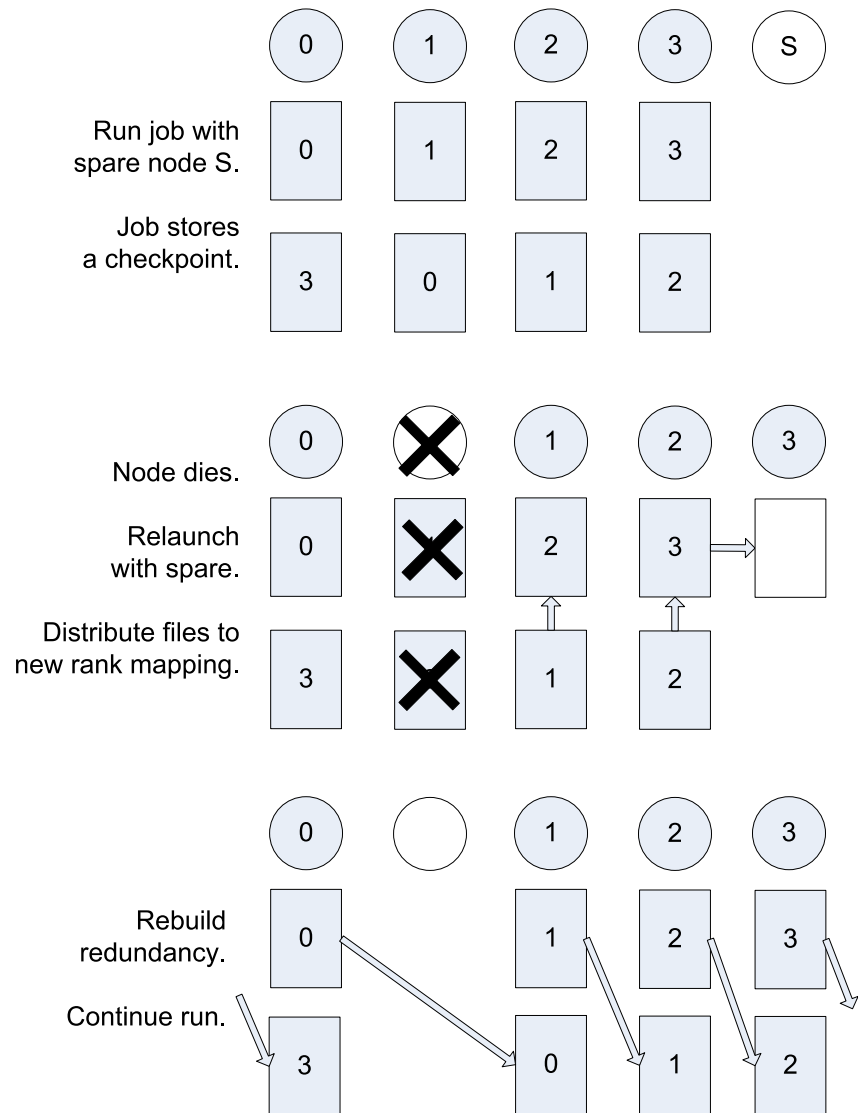


Figure 4: Example restart after a failed node with **Partner**

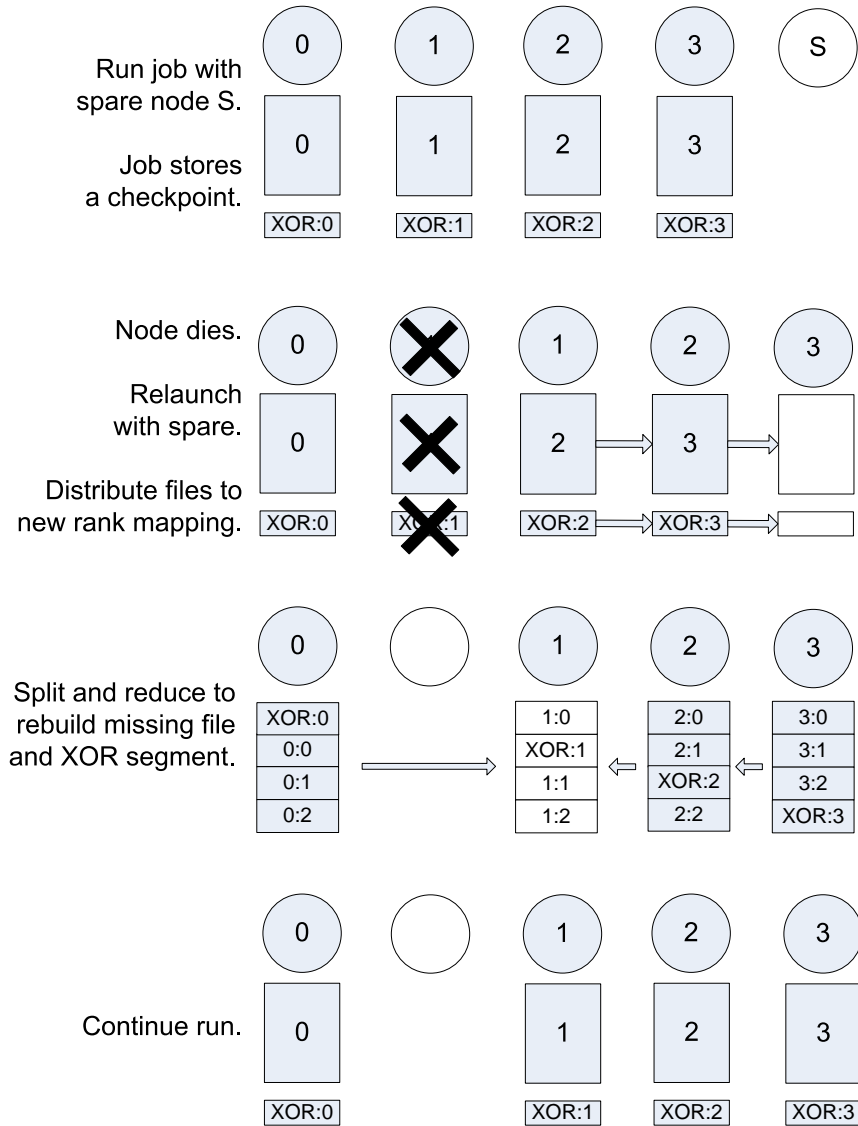


Figure 5: Example restart after a failed node with XOR

Failure during a checkpoint. The current implementation is capable of storing more than one checkpoint in the cache. However, due to cache size limitations, some applications may only be able to fit one checkpoint in cache at a time. For such cases, a failure may occur after the library has deleted the previous checkpoint and before the next checkpoint has completed. In this case, there is no valid checkpoint set in the cache to recover.

Failure of the node running the job batch script. The logic which executes at the end of the job to flush the latest checkpoint set from the cache to the parallel file system runs as part of the job batch script. If this node this script is running on fails, the flush logic will not be executed, the allocation will terminate, and the latest checkpoint set stored in the cache will be lost.

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

There are other catastrophic failure cases not listed here. To handle such failures, checkpoint sets must be written out 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 3.5 provides details on how to configure SCR to make occasional writes to the parallel file system.

By default, the current implementation stores the two most recent checkpoints in cache. When a new checkpoint is started, the implementation deletes the oldest checkpoint set from cache to make room for the next set. One can change the number of checkpoint sets stored in cache by setting the `SCR_CACHE_SIZE` environment variable. With large checkpoint data sets and/or limited cache sizes, it may be necessary to set this value to 1 to store just a single checkpoint, e.g., when the cache is using RAM disk on a machine with limited memory. However, remember that it is possible to encounter a failure while writing a checkpoint. When caching just one checkpoint set, such a failure is always catastrophic. When caching more than one checkpoint set, such a failure is only catastrophic if it also violates the redundancy scheme. Thus, reliability is improved if there is room to run with a cache size of at least 2.

3.5 SCR checkpoint directories

When SCR writes checkpoint sets out to the parallel file system, it writes checkpoint files to a *checkpoint directory*, which it creates within a *prefix directory* specified by the user. To specify the prefix directory, set the `SCR_PREFIX` environment variable to the full path where SCR should read and write checkpoint directories, e.g.,:

```
export SCR_PREFIX=/p/lscratchb/username/run1/checkpoints
```

Within the prefix directory, SCR creates each new checkpoint directory named according to the following format:

```
scr.YYYY-MM-DD_HH:MM:SS.jobid.checkpointid
```

where `YYYY-MM-DD` is the current date, `HH:MM:SS` is the current time, `jobid` is the job id of the job, and `checkpointid` is the internal SCR checkpoint iteration number.

For a given checkpoint set, SCR copies the files from all processes to the checkpoint directory. Then, if SCR determines that the checkpoint set is complete and valid, SCR creates a symlink called `scr.current` which points to the new checkpoint directory. If there was an existing `scr.current` symlink, SCR also creates an `scr.old` symlink that points to the previous `scr.current` directory. If the checkpoint set is determined to be incomplete or invalid, SCR leaves the new checkpoint directory and its files in place, but SCR does not update the `scr.current` or `scr.old` symlinks. In this manner, the `scr.current` and `scr.old` symlinks always point to complete and valid checkpoint sets.

As an example, if the prefix directory is set to `/p/lscratchb/username/run1/checkpoints`, after several runs of an SCR application, the contents of the prefix directory may look like the following:

```
>>: ls -l /p/lscratchb/username/run1/checkpoints
lrwxrwxrwx 1 username username 29 Oct 20 14:23 scr.current -> scr.2008-10-20_14:22:10.23167.50
lrwxrwxrwx 1 username username 29 Oct 20 14:23 scr.old -> scr.2008-10-20_11:30:45.23167.40
drwxr-xr-x 1 username username 4096 Oct 20 14:22 scr.2008-10-20_14:22:10.23167.50
drwxr-xr-x 1 username username 4096 Oct 20 11:30 scr.2008-10-20_11:30:45.23167.40
drwxr-xr-x 1 username username 4096 Oct 14 14:38 scr.2008-10-14_14:38:32.5283.33
drwxr-xr-x 1 username username 4096 Oct 14 13:38 scr.2008-10-14_13:38:32.5283.30
drwxr-xr-x 1 username username 4096 Oct 14 10:35 scr.2008-10-14_10:35:16.5283.20
drwxr-xr-x 1 username username 4096 Oct 09 11:25 scr.2008-10-09_11:25:05.188467.17
drwxr-xr-x 1 username username 4096 Oct 09 09:33 scr.2008-10-09_09:33:52.188467.10
```

In addition to the application checkpoint files, SCR also copies checkpoint redundancy data files and meta data files to the checkpoint directory. These files are used to reconstruct missing checkpoint files and to verify that the set of checkpoint files constitutes a complete and valid set. Meta data files end with an `“.scr”` extension. When using the XOR scheme, checkpoint redundancy data files end with an `“.xor”` extension. SCR also writes an `“scr_summary.txt”` file in each checkpoint directory, which records whether the set is complete and valid and to which MPI rank each file belongs. This file is needed to scatter the checkpoint files to the appropriate ranks upon a restart.

Each time an SCR job starts, SCR first inspects the checkpoint cache and attempts to distribute files for a scalable restart, as discussed in Section 3.2. If the cache is empty or the distribute operation fails or is disabled, SCR will attempt to fetch files from the `scr.current` checkpoint directory to fill the cache. For the fetch operation to succeed, `SCR_PREFIX` must be set, an `scr.current` symlink must exist in the prefix directory, and the specified checkpoint directory must contain an `scr_summary.txt` file. If the fetch from `scr.current` fails, SCR deletes the `scr.current` symlink, renames `scr.old` to `scr.current`, and tries again. If both fetch operations fail, SCR prints an error and continues the run. This condition is not treated as fatal as a convenience to simplify the process of starting a new run, since in this case, there is no existing checkpoint set (i.e., neither `scr.current` nor `scr.old`) to restart from. To disable the fetch operation, set the `SCR_FETCH` environment variable to 0. If an application disables the fetch feature, the application is responsible for reading its checkpoint set directly from the parallel file system upon a restart.

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

As an example of using the fetch and flush features, look again to the checkpoint directory example shown above. In this case, the checkpoint directories were produced by a long-running simulation that ran during three distinct time slots on three different days with `SCR_FETCH=1` and `SCR_FLUSH=10`. The job produced periodic flushes every 10 checkpoints (10, 20, 30, 40, and 50). In addition, it produced flushes at irregular checkpoint iterations (17 and 33), which occurred when the job flushed its most recent checkpoint set at the end of each of its first two time slots.

By default, SCR computes a CRC32 checksum value for each checkpoint file during a flush. Then, it uses this value, which is stored in the meta data file, to check the integrity of each file as it is read back into cache during a fetch. If file corruption is detected, the SCR library falls back to fetch an earlier checkpoint set. To disable this checksum feature, set the `SCR_CRC_ON_FLUSH` environment variable to 0.

4 Integrating SCR into an application

There are two ways to integrate the SCR library into an application. The direct method is to insert calls to the SCR API into application source code and compile and link with the library at build time. Alternatively, some applications may be able to use the library transparently via an interpose library. In such cases, one may use the library with an existing binary without having to change code, recompile, or relink.

4.1 The SCR API

The SCR API is designed to support a common checkpointing method used by large-scale codes, which is globally-coordinated checkpoints written primarily as a file per process. For such codes, the API is designed to be simple, scalable, and portable. The SCR API consists of a handful of C function calls that wrap around existing application logic. In most cases, one may fully integrate SCR into an application with fifteen to twenty lines of code. 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 each function 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. All functions return `SCR_SUCCESS` if successful.

4.1.1 SCR_Init

```
int SCR_Init();
```

Initialize the SCR library: identify partner nodes, prepare the storage cache, distribute and rebuild files upon a restart, etc.. This function must be called after `MPI_Init`, and generally, it is good practice to call this function just after calling `MPI_Init`. A process should only call `SCR_Init` once during its execution. No other SCR calls are valid until a process has returned from `SCR_Init`.

4.1.2 SCR_Finalize

```
int SCR_Finalize();
```

Shut down the SCR library: free resources, flush checkpoints to permanent storage, etc.. This function must be called before `MPI_Finalize`, and generally, it is good practice to call this function just before `MPI_Finalize`. A process should only call `SCR_Finalize` once during its execution.

4.1.3 SCR_Need_checkpoint

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

Since the failure frequency and the cost of checkpointing varies from platform to platform, `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.

4.1.4 SCR_Start_checkpoint

```
int SCR_Start_checkpoint();
```

Inform SCR that a new checkpoint is about to start. `SCR_Start_checkpoint` must be called by all processes. A process must call this function before it opens any files belonging to the new checkpoint. Ideally, this function should be called as soon as possible when initiating a new checkpoint. An SCR implementation may use 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_checkpoint` must be followed by a corresponding call to `SCR_Complete_checkpoint`.

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

4.1.5 SCR_Complete_checkpoint

```
int SCR_Complete_checkpoint(int valid);
```

Inform SCR that all files for the current checkpoint are complete (i.e., done writing and closed) and whether they are valid (i.e., written without error). `SCR_Complete_checkpoint` must be called by all processes. A process must close all of its checkpoint files before calling `SCR_Complete_checkpoint`. The parameter `valid` should be set to 1 if the calling process wrote all of its files successfully or if it had no files to write during the checkpoint. Otherwise, the process should call `SCR_Complete_checkpoint` with `value` set to 0. SCR will determine whether all processes wrote their checkpoint files successfully. The SCR implementation may also use this call as the stopping point to time the cost of the checkpoint that started with the preceding call to `SCR_Start_checkpoint`. Each call to `SCR_Complete_checkpoint` must be preceded by a corresponding call to `SCR_Start_checkpoint`.

In the current implementation, SCR applies the redundancy scheme during `SCR_Complete_checkpoint`. Also, before returning from the function, MPI rank 0 determines whether the job should be halted (see Section 6) and signals this condition to all other ranks. If the job should be halted, rank 0 records a reason in the halt file, and then all tasks call `exit`.

4.1.6 SCR_Route_file

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

Obtain the full path and file name a process must use to access a specified checkpoint file. During a restart, after a process calls `SCR_Init`, the process must call `SCR_Route_file` to obtain the full path and file name to use to open each of its files for reading. During a checkpoint, after a process calls `SCR_Start_checkpoint`, the process must call `SCR_Route_file` to: 1) register each of its checkpoint files as part of the current checkpoint set, and 2) obtain the full path and file name to use to open each of its files for writing. The name of the checkpoint file the process intends to access must be passed in the `name` argument. A pointer to a character buffer of at least `SCR_MAX_FILENAME` bytes must be passed in `file`. Upon returning from `SCR_Route_file`, the full path to access the file named in `name` is written to the buffer pointed to by `file`. The process must use this character string to open the file for reading or writing. A process does not need to make any directories listed in the string returned in `file`; the SCR implementation will create the necessary directories before returning from the call. `SCR_Route_file` always returns the full path and file name to the most recent checkpoint of a given file. A call to `SCR_Route_file` is local to the calling process; it is not a collective call.

4.2 Integrating the SCR API

There are three basic areas to consider when integrating the SCR API into an application's source code: Init/Finalize, Checkpoint, and Restart. One may employ the scalable checkpoint capability of SCR without the scalable restart capability. While it is most valuable to utilize both capabilities, some applications cannot use the scalable restart.

4.2.1 Init/Finalize

To use the SCR library, it is necessary to add calls to `SCR_Init` and `SCR_Finalize` in order to start up and shut down the library. The SCR library runs on top of the MPI library, and all calls made to SCR must be from within a well defined MPI environment, i.e., between `MPI_Init` and `MPI_Finalize`. In practice, it is often easiest to find the location where an application calls `MPI_Init` and insert the call to `SCR_Init` right after `MPI_Init`. Similarly, find the calls to `MPI_Finalize` and insert calls to `SCR_Finalize` right before `MPI_Finalize`. For example, modify the source to look something like this:

```
// Include the SCR header
#include "scr.h"

...

// Initialization
MPI_Init (...);
SCR_Init ();
```



```
...
```

```
// Finalization
SCR_Finalize ();
MPI_Finalize ();
```

Some applications have a number of distinct calls to `MPI_Finalize` depending on different code paths. Be sure to account for each call. The same applies to `MPI_Init`, but typically applications contain just a single call to this function.

4.2.2 Checkpoint

For checkpointing, first know that the application may rely on the SCR library to determine how often it should checkpoint. The SCR library can be configured with details on the failure frequency of the host platform, and it can compute the cost of taking a checkpoint as well as the time between checkpoints, so it has sufficient information to determine the optimal checkpoint frequency for the application on a given platform. To take advantage of this capability, the application should call `SCR_Need_checkpoint` at each natural opportunity it has to checkpoint, e.g., at the end of each timestep, and then initiate a checkpoint when SCR advises it to do so. In the current implementation, the library makes the decision on MPI rank 0, and it broadcasts this decision to the rest of the processes. As such, this call involves some amount of global synchronization, so it should not be called *too* often.

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. This function also serves as an interface to external instructions. For example, it can be used to inform the application to write a checkpoint just before the run is shut down via an external command like `scr_halt` (see Section 6).

To actually write a checkpoint, there are three phases. First, the application must call `SCR_Start_checkpoint` to define the start boundary of a new checkpoint. It must do this before it opens any file it writes as part of the checkpoint. Then, the application must call `SCR_Route_file` to register each file and to determine the full path and filename it must use to open each file that it writes during the checkpoint. Finally, after the application has registered, written, and closed each of its files for the checkpoint, it must call `SCR_Complete_checkpoint` 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_checkpoint` and `SCR_Complete_checkpoint` as these functions are collective. All files registered through a call to `SCR_Route_file` between a given `SCR_Start_checkpoint` and `SCR_Complete_checkpoint` pair are considered to be part of the same checkpoint file set. Some example checkpoint code with SCR may look like the following:

```
// Include the SCR header
#include "scr.h"

...

// Determine whether we need to checkpoint
int flag;
SCR_Need_checkpoint(&flag);
if (flag) {
    // Tell SCR that a new checkpoint is starting
    SCR_Start_checkpoint ();

    // Define the checkpoint filename for this process
    int rank;
    char name[256];
    MPI_Comm_rank(MPLCOMM_WORLD, &rank);
    sprintf(name, "rank_%d.ckpt", rank);

    // Register our file, and get the full path to open it
```

```

char file[SCR_MAX_FILENAME];
SCR_Route_file(name, file);

// Open, write, and close the file
int valid = 0;
FILE* fs = open(file, "w");
if (fs != NULL) {
    valid = 1;
    size_t n = fwrite(checkpoint_data, 1, sizeof(checkpoint_data), fs);
    if (n != sizeof(checkpoint_data)) { valid = 0; }
    if (fclose(fs) != 0) { valid = 0; }
}

// Tell SCR that this process is done writing its checkpoint files
SCR_Complete_checkpoint(valid);
}

```

4.2.3 Restart

There are two options to access files during a restart: with and without SCR. If an application is designed such that each MPI task, on a restart, only needs access to the files it wrote during the previous checkpoint, then the application can utilize the scalable restart capability of the SCR library. This enables the application to restart from a cached checkpoint in the existing job allocation without having to write files out and read files in via the parallel file system. To use SCR on a restart, the application must call `SCR_Route_file` to determine the full path and file name it must use to open each of its checkpoint files.

If the fetch operation is enabled and there are no files in cache or there is an incomplete set of files in cache, the library will fetch files from the `scr.current` directory to fill the cache. Then `SCR_Route_file` will return the path and file name to the cached version of the file. If one would like to restart from an earlier checkpoint set than the one pointed to by the `scr.current` symlink, one should manually reassign the `scr.current` symlink to point to the appropriate directory before starting.

To use SCR during a restart, some example restart code may look like the following:

```

// Include the SCR header
#include "scr.h"

...

// Define the checkpoint filename for this process
int rank;
char name[256];
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
sprintf(name, "rank_%d.ckpt", rank);

// Get the full path to open our file
char file[SCR_MAX_FILENAME];
SCR_Route_file(name, file);

// Open, read, and close the file
FILE* fs = open(file, "r");
size_t n = fread(checkpoint_data, 1, sizeof(checkpoint_data), fs);
fclose(fs);

```

If the application does not use SCR during restarts, it should not make calls to `SCR_Route_file` during the restart. Instead, it should access files directly from the parallel file system. The application must account for the SCR file and directory structure used to store checkpoint sets, e.g., to start from the most recent checkpoint, the application should read its checkpoint files from the `scr.current` directory. Be aware that this file and directory

structure may change between library version updates. Also, know that when restarting without SCR, the precise value of the `SCR_FLUSH` counter at the time the last checkpoint was written will not be preserved in the restart. The counter will be reset to its upper limit with each restart. Thus, each restart may introduce some fixed offset in a series of periodic SCR flushes.

4.3 Building with the SCR library

To integrate SCR into an application, first add calls to the SCR API into the source code as necessary. Remember to include the “`scr.h`” header file in those files where calls to the SCR API are made. Then, to compile and link with the SCR library, add the flags in Table 2 to your compile and link lines.

Table 2: SCR build flags

Compile Flags	<code>-I/usr/local/tools/scr-1.1/include</code>
Link Flags	<code>-L/usr/local/tools/scr-1.1/lib -lscr -Wl,-rpath,/usr/local/tools/scr-1.1/lib</code>

4.4 Using the SCR interpose library

An application that has not integrated calls to the SCR API may still be able to use SCR via an interpose library if it adheres to certain conditions. Using the `LD_PRELOAD` environment variable on Linux platforms, the interpose library transparently calls functions in the SCR library upon intercepting existing application calls to the following functions: `MPI_Init`, `MPI_Finalize`, `open/close`, `fopen/fclose`, and `mkdir`. While FORTRAN applications typically do not make direct calls to these C functions, the equivalent functions in the FORTRAN runtime library often do call these C functions internally. Thus, the interpose library can often be used with FORTRAN applications. To use the interpose library, an application must:

- adhere to all constraints specified for the standard SCR library,
- open and close checkpoint files via calls to `open/close` or `fopen/fclose`,
- open all checkpoint files in read-only mode during restarts,
- open all checkpoint files in write-only or read-write mode during restarts,
- open at least one checkpoint file per process per checkpoint,
- open and close each checkpoint file for each process exactly once per checkpoint,
- be able to open the first checkpoint file on each process synchronously across all processes during each checkpoint, i.e., be able to execute an `MPI_Barrier` over `MPI_COMM_WORLD` during the call to open for the first checkpoint file that each process opens during the checkpoint,
- be able to close the last checkpoint file for each process synchronously across all processes during each checkpoint, i.e., be able to execute an `MPI_Barrier` over `MPI_COMM_WORLD` during the call to close for the last checkpoint file that each process closes during the checkpoint,

The interpose library intercepts the application’s call to `MPI_Init`, and it calls `MPI_Init` and then `SCR_Init` before returning. Similarly, it intercepts the application’s call to `MPI_Finalize`, and it calls `SCR_Finalize` before calling `MPI_Finalize`.

Additionally, the interpose library intercepts each file open call. It checks the file name and the rank of the calling process against a list of specified checkpoint file names (discussed later). If the file is a checkpoint file and it is opened in read-only mode, the interpose library calls `SCR_Route_file` and uses the SCR path and filename to open the file. Then, it returns the newly created file descriptor (or file stream).

If the file is a checkpoint file and it is opened in write mode (or read-write mode), the interpose library assumes a checkpoint set is being written. If the interpose library has not already started a new checkpoint, it calls `SCR_Start_checkpoint`. Since `SCR_Start_checkpoint` is a collective call, each process must open at least one file

per checkpoint, and all processes must be able to open their first file synchronously during each checkpoint. After starting a new checkpoint set, the interpose library assumes each process will open each of its listed checkpoint files in write mode (or read-write mode). During the open for each file, the library calls `SCR_Route_file`, opens the file using the path and file name provided by SCR, and returns the associated file descriptor (or file stream) to the application. To determine the end of a new checkpoint set, the interpose library intercepts and tracks each file close call. When a process closes the file descriptor for its last checkpoint file, the interpose library calls `SCR_Complete_checkpoint`. Since `SCR_Complete_checkpoint` is a collective call, all processes must be able to close their last file synchronously during each checkpoint.

If a process writes multiple files per checkpoint, it may open and close its files for that checkpoint in any order. A process must open and close each of its listed checkpoint files for writing exactly once per checkpoint. However, the order in which it makes the corresponding open and close calls does not matter. All that matters is that the first open and the last close of each checkpoint set on each process are able to synchronize across all processes in the job.

To specify the checkpoint file list, set the `SCR_CHECKPOINT_PATTERN` environment variable. The `SCR_CHECKPOINT_PATTERN` environment variable must specify the file names that each process writes during each checkpoint. A list of files may be specified separated by commas. To use a different separation character, one may specify the desired character in the `SCR_CHECKPOINT_PATTERN_TOKEN` environment variable. For each file, a range specifier lists which MPI ranks write the file and a regular expression describes the file name. The range specifier is of the form `X-Y` where `X` and `Y` define the inclusive bounds of the MPI ranks that write the file. The special value `N` may be used to specify the maximum rank in a job. The range specifier is separated from the file name regular expression using a `:` character. The file name regular expressions are processed within the interpose library via calls to `regcomp` and `regexexec`. See the man pages for these functions for full details on the regular expression syntax. In order to inhibit shell interpretation of regular expression characters, it is recommended to write the file list within quotes when setting `SCR_CHECKPOINT_PATTERN`.

As an example, if all ranks write a checkpoint file called `rank_X.ckpt` where `X` is the MPI rank of the process, the following could be used:

```
export SCR_CHECKPOINT_PATTERN='0-N:rank_[0-9]+.ckpt'
```

Above, the range `0-N` specifies that ranks from rank 0 to the maximum rank in the job (i.e., all ranks in the job), write a file called `rank_X.ckpt` where `X` is some number. As another example, if all ranks write a file like above *and* rank 0 writes an additional file called `root.ckpt`, the following could be used:

```
export SCR_CHECKPOINT_PATTERN='0-0:root.ckpt,0-N:rank_[0-9]+.ckpt'
```

This setting lists two files (note the comma which splits them). The first denotes that ranks from rank 0 to rank 0 (i.e., rank 0 only) writes a file called `root.ckpt` and that all ranks write a file like in the previous example.

Many applications maintain multiple checkpoint sets on the parallel file system throughout their run in case one set becomes corrupted or to enable a restart from various points in the computation. Such applications typically create a new subdirectory in the parallel file system to contain each checkpoint. When using the interpose library, many checkpoint sets will be stored and overwritten in cache without ever being written out to the parallel file system. However, since the application does not know it is running on top of the SCR interpose library, it will still attempt to create a directory for each of its checkpoints. For such applications, it is useful to configure the interpose library to intercept directory creation calls to avoid creating these numerous empty checkpoint subdirectories. To do this, specify the name of the checkpoint directory as a regular expression via the `SCR_CHECKPOINT_DIR_PATTERN` environment variable. When `SCR_CHECKPOINT_DIR_PATTERN` is set, the SCR interpose library inspects the directory name being created in each call to `mkdir`. If the name matches the regular expression specified in `SCR_CHECKPOINT_DIR_PATTERN`, the interpose library does not create the directory.

As an example, if the application creates checkpoint directories such as `ckpt_T` where `T` is the simulation time step number, the following setting could be used:

```
export SCR_CHECKPOINT_DIR_PATTERN='ckpt_[0-9]+'
```

Note that the regular expression for checkpoint directory names is a single item (not a list), and it applies to all ranks in the job (there is no MPI rank range specifier).

Finally, to use the interpose library, invoke `scr_srun` (see Section 5.3) after setting `SCR_CHECKPOINT_PATTERN`. This command will automatically set `LD_PRELOAD` to load the interpose library when it detects that `SCR_CHECKPOINT_PATTERN` is set. Otherwise, if you run without using `scr_srun`, you'll need to set `LD_PRELOAD` manually, and since this variable applies to all commands, you'll want to unset it immediately after the run. For example, when not using `scr_srun`, you'll want to do something like the following:

```
# set LD_PRELOAD, run the job, unset LD_PRELOAD
export LD_PRELOAD='/usr/local/tools/scr-1.1/lib/libscr-interpose.so'
srun <srun_args ...> <prog> <prog_args ...>
unset LD_PRELOAD
```

5 Integrating SCR into a job script

In addition to the runtime library, an SCR job must be correctly configured for the batch system and a set of SCR commands must be integrated into the job script. There are two primary goals here: 1) inform the system that the allocation should remain available even after a node failure, and 2) add logic to the job script to copy the latest checkpoint set from cache before the allocation expires.

5.1 Distinguishing SLURM job steps from job allocations

When using SCR, it is useful to make the distinction between a *job allocation* and a *job step* in SLURM. When a job is scheduled resources on a system running under SLURM, the job script executes inside of a SLURM 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.

Within a job allocation, a user may run one or more job steps, each of which is invoked by a call to `srun`. Each job step is assigned its own step id. Within each job allocation, the job step ids start at 0 and increment upwards 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 is in the following format: `jobid.stepid`. For instance, the name `1234.5` refers to step id 5 of job id 1234. Throughout the rest of this document, the terms *job allocation*, *job*, and *allocation* may all be used interchangeably to refer to a SLURM job allocation, while the terms *job step* and *step* may be used interchangeably to refer to a SLURM job step.

5.2 Configuring the job allocation

Before running an SCR job, it is first necessary to configure the job allocation to withstand node failures. By default, MOAB / SLURM will kill a job allocation if a node fails. SCR requires the allocation to remain active in order to flush files, so the first task is to instruct MOAB / SLURM to not kill the allocation if a node fails.

If you are running a MOAB job, add the “`#MSUB -l resfailpolicy=ignore`” option to your job script. If you are running an interactive MOAB job via `mxterm`, add the “`-l resfailpolicy=ignore`” option to your `mxterm` parameter list. Finally, if you are running interactively within a SLURM allocation outside of MOAB (e.g., in a `pdebug` pool), be sure to create your SLURM allocation with the “`--no-kill`” option. These flags are summarized in Table 3.

Table 3: SCR job allocation flags

MOAB job script	<code>#MSUB -l resfailpolicy=ignore</code>
MOAB via <code>mxterm</code>	<code>mxterm ... -l resfailpolicy=ignore</code>
Interactive SLURM	<code>salloc --no-kill ...</code>

5.3 SCR commands

SCR includes a set of commands that, among other things, prepare the checkpoint cache, flush files from the cache to the parallel file system, and check that the flushed checkpoint set is complete. These commands are located in

the SCR /bin directory. To add these commands to your environment, use the “scr-1.1” dotkit, e.g.:

```
source /usr/local/tools/dotkit/init; use scr-1.1
```

Typically these commands are called from the job batch script before and after the job step is run. Some commands may be invoked manually outside of the job script. Most commands include man pages and, in many cases, the command usage will print to `stdout` when passed a “-h” option.

5.3.1 scr_srun

The easiest way to integrate SCR commands into a job batch script is to set a few environment variables and replace the `srun` command with `scr_srun`. The `scr_srun` command wraps calls to a number of SCR commands and other logic that simplifies the use of SCR in the common case. Internally, it invokes `scr_prerun`, `srun`, `scr_postrun`, and `scr_check_complete`. The `scr_srun` command also executes logic to optionally restart an application within an existing allocation.

SYNOPSIS: `scr_srun [srun_options] <prog> [prog_args ...]`

The `scr_srun` command requires the prefix directory to be specified. By default, it will use the current working directory as the prefix directory. One may specify a different prefix directory by setting the `SCR_PREFIX` environment variable as described in Section 3.5. On systems with `libyogrt`, it is highly recommended that one also set the `SCR_HALT_SECONDS` environment variable so the job allocation does not expire before the latest checkpoint can be flushed (see Section 6).

By default, `scr_srun` will not attempt to restart an application after the first job step exits. If you would like the command to attempt to restart the application in a new job step within the current allocation, set the `SCR_RUNS` environment variable to the maximum number of runs the command should attempt. For an unlimited number of attempts, set this variable to -1. After the first job step exits, the command will check whether it should make any additional attempts. If so, the command will sleep for some time to give the nodes in the allocation a chance to clean up. After this delay, the command checks whether there are sufficient healthy nodes remaining in the allocation to run the application. By default, the command assumes the next run requires the same number of nodes as the last run, which is recorded in a file written in the checkpoint cache 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.

In the current implementation, `scr_srun` must be run from within a SLURM job allocation. The command processes no parameters – it passes all parameters directly to `srun`. An example job script that uses `scr_srun` is provided in Section 5.4. If you elect to use `scr_srun`, then you may skip the following sections on `scr_prerun`, `scr_postrun`, and `scr_check_complete` as `scr_srun` will internally invoke these commands for you.

5.3.2 scr_prerun

The `scr_prerun` command must be run within a SLURM job allocation *before* attempting the first run of an SCR application. This command executes various tasks to prepare the compute nodes in the job allocation for the SCR library. It may check that the compute nodes are capable of storing checkpoint data. It may also scatter checkpoint files from the parallel file system to the compute nodes to bootstrap the cache.

SYNOPSIS: `scr_prerun [-p prefix_dir]`

A job script must check that the return code of `scr_prerun` is 0 before it may continue. If a non-zero exit code is returned, the `scr_prerun` command failed, in which case, the job script should exit immediately. In particular, if `scr_prerun` fails, the job script should not attempt to launch the application or flush checkpoint files via `scr_postrun`.

The `scr_prerun` command does not need to be nor should it be run before attempting to restart an SCR application within the same job allocation. In fact, doing so may destroy any existing cached checkpoint set, as the command may clear the checkpoint cache as part of its operation.

The `scr_prerun` command requires the prefix directory to be specified. By default, it will use the current working directory as the prefix directory. One may specify a different prefix directory by setting the `SCR_PREFIX` environment variable or calling `scr_prerun` with the “-p” option, e.g.:

```
scr_prerun -p /p/lscratchb/username/run1/checkpoints
```

The `scr_prerun` command accepts a number of other optional parameters not detailed here.

5.3.3 `scr_postrun`

The `scr_postrun` command should be run within a SLURM job allocation *after* the last SCR application job step exits. Its primary purpose is to flush the checkpoint files from the compute node cache to the parallel file system. It executes logic to identify failed nodes and flush redundant file data as needed. It also calls `scr_check_complete` after the flush to rebuild any missing checkpoint files, provided there are sufficient redundant data to reconstruct the lost files.

SYNOPSIS: `scr_postrun [-p prefix_dir]`

The `scr_postrun` command should be run after the final job step in an allocation in order to flush the latest checkpoint set. It may be inefficient, but it is not erroneous to run the command after each job step in an allocation. This technique can be useful to flush files to the parallel file system between restarts for codes which do use not utilize the scalable restart feature.

The `scr_postrun` command requires the prefix directory to be specified. By default, it will use the current working directory as the prefix directory. One may specify a different prefix directory by setting the `SCR_PREFIX` environment variable or calling `scr_postrun` with the “-p” option, e.g.:

```
scr_postrun -p /p/lscratchb/username/run1/checkpoints
```

The `scr_postrun` command accepts a number of other optional parameters not detailed here.

5.3.4 `scr_check_complete`

The `scr_check_complete` command checks whether the files in a checkpoint directory on the parallel file system constitute a complete and valid set, and it writes the `scr_summary.txt` file. It rebuilds missing checkpoint files if there are sufficient redundant file data.

SYNOPSIS: `scr_check_complete <checkpoint_dir>`

The `scr_check_complete` command requires a single parameter that specifies the checkpoint directory (not the prefix directory) on the parallel file system where the checkpoint file set was written to, e.g.:

```
scr_check_complete /p/lscratchb/username/run1/checkpoints/scr.current
```

One may invoke `scr_check_complete` outside of a SLURM job allocation. This is useful to check and restore a checkpoint set in which `scr_postrun` may have failed to complete. In this case, it is left to the user to update the `scr.current` and `scr.old` symlinks if so desired.

5.3.5 `scr_halt`

The `scr_halt` command is used to instruct a running SCR application to exit. The SCR application can be instructed to exit after its next successful checkpoint, exit before or after a specified time, or exit immediately. More detailed discussion of this command is provided in [Section 6](#).

SYNOPSIS: `scr_halt [options] [jobid ...]`

5.3.6 Other commands

There are a number of other commands in the SCR `/bin` directory. Any command not mentioned in the previous sections is considered to be a layer below the user interface.

5.4 Example MOAB job script with SCR

```
#!/bin/bash
#MSUB -l partition=atlas
#MSUB -l nodes=66
#MSUB -l resfailpolicy=ignore

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

# add the scr commands to the job environment
. /usr/local/tools/dotkit/init.sh
use scr-1.1

# specify where checkpoint directories should be written
export SCR_PREFIX=/p/lscratchb/username/run1/checkpoints

# 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
```

6 Instructing an SCR job to halt

It is important to give SCR sufficient time to flush files from the checkpoint cache to the parallel file system before a job allocation ends. Otherwise, the latest checkpoint set stored in the cache will be lost. There are several mechanisms available to instruct a running SCR application to halt.

6.1 `scr_halt` and the halt file

A convenient method to stop an SCR application is to use the `scr_halt` command. To add this command to your environment, use the “scr-1.1” dotkit, e.g.,:

```
source /usr/local/tools/dotkit/init; use scr-1.1
```

A number of different halt conditions can be specified as described in the following sections. In most cases, the `scr_halt` command communicates its halt conditions to the running SCR application by writing them to the halt file (“`halt.scrinfo`”), which is located in the checkpoint cache. SCR reads the halt file when the application calls `SCR.Init` and each time the application successfully completes a checkpoint during a call to `SCR.Complete_checkpoint`. If a halt condition is satisfied, all tasks in the SCR application call `exit`.

6.2 Halt after next (X) checkpoint(s)

You can instruct an SCR job to halt after completing its next successful checkpoint. For instance, for a job with id 1234, you could halt the job using the following command:

```
scr_halt 1234
```

You can also instruct an SCR job to halt after completing X checkpoints via the `--checkpoints` option. If the last of the X checkpoints is unsuccessful, SCR continues the job until it completes its next successful checkpoint. This ensures that SCR has a successful checkpoint set to flush before it halts the job. For example, to instruct job 1234 to halt after 10 more checkpoints, use the following:

```
scr_halt --checkpoints 10 1234
```

6.3 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 after completing its next successful checkpoint after the given time. For example, you can instruct job 1234 to halt after “12:00pm today” via:

```
scr_halt --after '12:00pm today' 1234
```

It is also possible to instruct a job to halt *before* a specified time using the `--before` option. For example, you can instruct job 1234 to halt before “8:30am tomorrow” via:

```
scr_halt --before '8:30am tomorrow' 1234
```

For the “halt before” condition to really be effective, one must also set the `SCR_HALT_SECONDS` environment variable before running the job. When `SCR_HALT_SECONDS` is set to some positive number of seconds, SCR checks how much time is left before the specified time limit. If the remaining time is less than or equal to `SCR_HALT_SECONDS`, SCR halts the job.

It is highly recommended that `SCR_HALT_SECONDS` be set on systems with `libyogrt`. On such system, the SCR library imposes a default “halt before” condition using the end time of the job allocation. This ensures the latest checkpoint set 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 possibly rebuild) files in `scr_postrun` before the allocation expires. Also consider that a checkpoint may be taken just *before* the point where the remaining time is less than `SCR_HALT_SECONDS`. Hence, if a code checkpoints every X seconds and it takes Y seconds to flush files from the cache and rebuild, set $\text{SCR_HALT_SECONDS} = X + Y + \Delta$, where Δ is some positive value to provide some additional slack.

The value of `SCR_HALT_SECONDS` does not affect the “halt after” condition.

One may also set the halt seconds via the `--seconds` option to `scr_halt`. This option enables one to set and change the halt seconds on a running job. One may unset the halt seconds using the `--unset-seconds` option.

NOTE: If any `scr_halt` commands are specified as part of the batch script before the first job step is launched, one must then use `scr_halt` to set the halt seconds for the job rather than the `SCR_HALT_SECONDS` environment variable. This is because 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` environment variable.

6.4 Halt immediately

Sometimes, you may need to halt an SCR job immediately. In this case, take precaution not to inadvertently stop the job if it is in the middle of a checkpoint. Such a mistake is especially costly when SCR is configured to only store one checkpoint set in cache, since in this case, there is no complete checkpoint set to flush after the job is stopped. It is less costly to make this mistake if SCR has multiple checkpoint sets in cache, since it can still flush the next most recent checkpoint. On the other hand, if the goal is to intentionally kill the job while it is in the

middle of a checkpoint (because maybe it is hanging), this method is also the right approach to use. There are two options to halt an SCR job immediately.

You may use the `--immediate` option, e.g.,:

```
scr_halt --immediate 1234
```

This command first updates the halt file, so that the job will not be restarted once stopped. Then, it kills all running job steps for the job via `scancel`.

If for some reason the `--immediate` option fails to work, you may manually halt the job. First, issue a simple `scr_halt` so the job will not restart, and then manually kill the currently running job step via `scancel`. When using `scancel`, be careful to cancel the job step and not the job allocation. Canceling the job kills the allocation, which destroys the files stored in the cache. 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 1234
scancel 1234.5
```

Note that `scr_halt` is given the job id (without the step id) but `scancel` is given the the full job step name (job id and step id). In particular, do *not* just type “`scancel 1234`” – be sure to include the job step id.

6.5 Catching a hanging job

If the job step hangs, then the SCR library will not get a chance to halt the job per any of the above conditions. In order to avoid losing significant compute time due to a hang, the `io-watchdog` plugin in SLURM is quite helpful. For more information on this tool, see `/usr/local/tools/io-watchdog/README`.

However, even with `io-watchdog`, you must be careful to check that the job does not hang near the end of its time limit, since in this case, `io-watchdog` may not kill the job step in time before the allocation ends. If you suspect the job to be hanging and you deem that `io-watchdog` will not kill the job step in time, manually cancel the job step as described above.

6.6 Combining, listing, changing, and unsetting 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 job 1234 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' 1234
```

The following sequence also works:

```
scr_halt --checkpoints 10 1234
scr_halt --before '8:30am tomorrow' 1234
```

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

```
scr_halt --list 1234
```

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 job 1234, type the following:

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

6.7 Removing the halt file

Sometimes, especially during testing, you may want to rerun a job step in an existing allocation after halting the previous job step. When SCR detects a halt file with a satisfied halt condition during `SCR_Init` or after a checkpoint, it immediately exits. This is the desired effect when trying to halt a job, however, this mechanism also prevents one from intentionally re-running a job step in an allocation after halting a previous job step (or after a job step has run to completion, since SCR also produces a halt file when the application calls `SCR_Finalize`.)

When a job step runs in an allocation containing a halt file with a satisfied halt condition, a message is printed to `stdout` which indicates why SCR is halting. For example, you may see something like the following:

```
SCR: rank 0 on hype55: Job exiting: Reason: SCR_FINALIZE_CALLED.
```

To rerun in such a case, it is necessary to first remove the satisfied halt conditions. This can be accomplished by unsetting any satisfied conditions or resetting them to appropriate values. Another approach is to completely remove the halt file via the `--remove` option. This action effectively removes all halt conditions. For example, to remove the halt file from job allocation 1234, type:

```
scr_halt --remove 1234
```

7 Environment variables

Table 4: SCR environment variables

Name	Default	Description
<code>SCR_ENABLE</code>	1	Set to 0 to disable SCR at runtime.
<code>SCR_COPY_TYPE</code>	XOR	Set to one of: <code>LOCAL</code> , <code>PARTNER</code> , <code>XOR</code> .
<code>SCR_CACHE_SIZE</code>	2	Set to a non-negative integer to specify the maximum number of checkpoints SCR should keep in cache.
<code>SCR_PARTNER_DISTANCE</code>	1	Set to a positive integer to specify the number of hops taken to select a partner node for <code>PARTNER</code> or the number of hops between nodes of the same XOR set for <code>XOR</code> . In general, 1 will give the best performance, but a higher value may enable SCR to recover from more severe failures which take down multiple consecutive nodes (e.g., a power breaker which supplies a rack of consecutive nodes).
<code>SCR_XOR_SIZE</code>	8	Specify the number of nodes to use in a single XOR set. Increasing this value decreases the amount of storage required to cache the checkpoint data on the compute nodes. However, higher values have an increased likelihood of encountering a catastrophic error. Higher values also require more time to reconstruct lost files from redundancy data.
<code>SCR_PREFIX</code>	N/A	Specify the prefix (root) directory where checkpoint directories should be read from and written to.
<code>SCR_FETCH</code>	1	Set to 0 to disable SCR from fetching files from the parallel file system to bootstrap the compute node cache during <code>SCR_Init</code> .
<code>SCR_FETCH_WIDTH</code>	256	Specify the number of processes that may read simultaneously from the parallel file system.
<code>SCR_DISTRIBUTE</code>	1	Set to 0 to disable file distribution during <code>SCR_Init</code> . File distribution enables the application to replace failed nodes with spare nodes during a restart from the checkpoint cache.
<code>SCR_FLUSH</code>	10	Specify the number of checkpoints between periodic SCR flushes to the parallel file system. Set to 0 to disable periodic flushes.
<code>SCR_FLUSH_WIDTH</code>	256	Specify the number of processes that may write simultaneously to the parallel file system.

Table 4 – continued from previous page

Name	Default	Description
SCR_FLUSH_ON_RESTART	0	Set to 1 to force SCR to flush a checkpoint during restart. This may be useful for codes which cannot use the scalable restart.
SCR_RUNS	1	Specify the maximum number of times the <code>scr_srun</code> command should attempt to run a job within an allocation.
SCR_MIN_NODES	N/A	Specify the minimum number of nodes in an allocation required to run a job.
SCR_EXCLUDE_NODES	N/A	Specify a set of nodes, in SLURM node range syntax, which should be excluded from runs. This is useful to avoid particular nodes that consistently fail for an application before they have been fixed by system administrators. Nodes in this list which are not in the current allocation are silently ignored.
SCR_HALT_SECONDS	0	Set to a positive integer to instruct SCR to halt the job after completing a successful checkpoint if the remaining time is less than the specified number of seconds.
SCR_MPI_BUF_SIZE	131072	Specify the number of bytes to use for internal MPI send and receive buffers when chunking checkpoint 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 compute node cache.
SCR_CRC_ON_COPY	0	Set to 1 to enable crc32 checking when copying files during the selected redundancy scheme. (Currently only applies to <code>PARTNER</code> .)
SCR_CRC_ON_FLUSH	1	Set to 0 to disable crc32 checking during fetch and flush operations.
SCR_DEBUG	0	Set to 1 or 2 for increasing verbosity levels of debug messages.

8 Acknowledgments

A number of people have contributed to the development of SCR. In particular, Greg Bronevetsky has contributed ideas, research, and development work to the effort. Many thanks are also owed to Denise Hinkel, Bruce Langdon, Ed Williams, Steve Langer, and especially Bert Still as the early adopters of this library. They devoted much of their time to testing and experimenting with new releases, and many features were developed in response to feedback from their experience with early versions.

References

- [1] M. Jette, “Simple Linux Utility for Resource Management (SLURM).” <https://computing.llnl.gov/linux/slurm>.
- [2] N. H. Vaidya, “A case for two-level recovery schemes,” *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 656–666, 1998.
- [3] W. Gropp, R. Ross, and N. Miller, “Providing Efficient I/O Redundancy in MPI Environments,” in *Lecture Notes in Computer Science, 3241:7786, September 2004. 11th European PVM/MPI Users Group Meeting*, 2004.
- [4] D. Patterson, G. Gibson, and R. Katz, “A Case for Redundant Arrays of Inexpensive Disks (RAID),” in *Proc. of 1988 ACM SIGMOD Conf. on Management of Data*, 1988.