The Scalable Checkpoint / Restart (SCR) Library Version 1.1-6 User Manual

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

March 22, 2010

Abstract

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 and restarting large-scale jobs. With SCR, jobs run more efficiently, lose less work upon a failure, and reduce load on critical shared resources such as the parallel file system and the network infrastructure.

In the current SCR implementation, application 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 even after a failure disables part of the system. SCR supports the use of spare nodes such that it is possible to restart a job directly from its cached checkpoint, provided the redundancy scheme holds and provided there are sufficient spares.

On current platforms, SCR scales linearly with the number of compute nodes. It has been benchmarked at 1216GB/s on 992 nodes of Coastal at Lawrence Livermore National Laboratory (LLNL). At this scale, it is two orders of magnitude faster than the parallel file system.

The SCR library is written in C, and it currently only provides a C interface. It has been used in production at LLNL since late 2007 using RAM disk on Linux / x86-64 / Infiniband clusters. Production runs are now underway using solid-state drives on the same cluster architecture. The implementation assumes SLURM [1] is used as the resource manager. The library is designed and intended to be ported to other platforms and resource managers. SCR is an open source project under a BSD license hosted at: http://scalablecr.sourceforge.net.

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

Compared to small-scale jobs, large-scale jobs face an increased likelihood of encountering system failures during their run. To handle more frequent failures, large-scale jobs must checkpoint more frequently. These checkpoints ultimately must be stored in the parallel file system.

However, there are two drawbacks to checkpointing a large-scale job to the parallel file system. Firstly, large-scale jobs must often checkpoint large data sets for which the parallel file system delivers a relatively small amount of bandwidth. Secondly, the parallel file system is often shared among multiple jobs running on multiple compute clusters, and so it may be busy servicing other jobs when the job needs to checkpoint. Either of these conditions idles the job while it waits for the storage resource.

The Scalable Checkpoint / Restart (SCR) library addresses both of these problems. The solution SCR employs derives from two key observations. Firstly, a job only needs its most recent checkpoint. As soon as the next checkpoint is written, the previous checkpoint can be discarded. Secondly, a typical failure disables a small portion of the system, but it otherwise leaves most of the system intact.

Leveraging these two properties, SCR is designed to cache checkpoints in storage local to the compute nodes rather than write them to the parallel file system. SCR caches only the most recent checkpoints, discarding an older checkpoint with each newly saved checkpoint. SCR also applies a redundancy scheme to the cache such that it is possible to recover checkpoints even after a failure disables a small portion of the system. With this design, each checkpoint is written to the cache and discarded, unless a failure occurs, at which point SCR recovers the most recent checkpoint from cache. It can then copy the recovered checkpoint to the parallel file system, or if there are spare resources available, the job may restart directly from its cached checkpoint. By writing checkpoints to the cache instead of the parallel file system, SCR delivers scalable bandwidth utilizing storage resources that are fully dedicated to the job. The current implementation achieves exceptional bandwidth and scalability, see Figure 1.

Even when using SCR, it remains necessary to write some checkpoints to the parallel file system, since some failures disable larger portions of the system than the redundancy scheme is designed to handle. However, with a well-chosen redundancy scheme, the frequency with which these writes must be made can be greatly reduced. In essence, SCR is a production-level implementation of a two-level checkpoint system of the type analyzed by Vaidya in [2].

Because large-scale jobs can checkpoint so much faster with SCR, it is often possible to configure such a job to checkpoint more frequently while simultaneously reducing its total checkpointing overhead. Such a job will then lose less work upon encountering a failure, but it will also utilize the machine more efficiently when there are no failures. Additionally, if spare resources are available, a job can be restarted faster with SCR. Finally, by shifting its checkpointing traffic to SCR, a job reduces its load on the parallel file system and the associated network, which benefits all jobs sharing those resources.

SCR is really a system of two components: a library and a set of commands. The application invokes the SCR library to determine where it should read and write its checkpoint files, and the library maintains the checkpoint 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 recover the latest checkpoint from cache and copy it to the parallel file system upon a failure. The SCR library is described in Section 4. The SCR commands are discussed in Section 5. Important concepts applicable to both are covered in Section 3. Details on configuring an SCR job are provided in Section 6, and instructions on how to stop an SCR job are provided in Section 7. However, before covering any of that, one must understand all of the assumptions made by the current SCR implementation as specified in the next section.

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 SCR developers. The goal is to expand the implementation to support a large number of applications.

1. The code must be an MPI application.

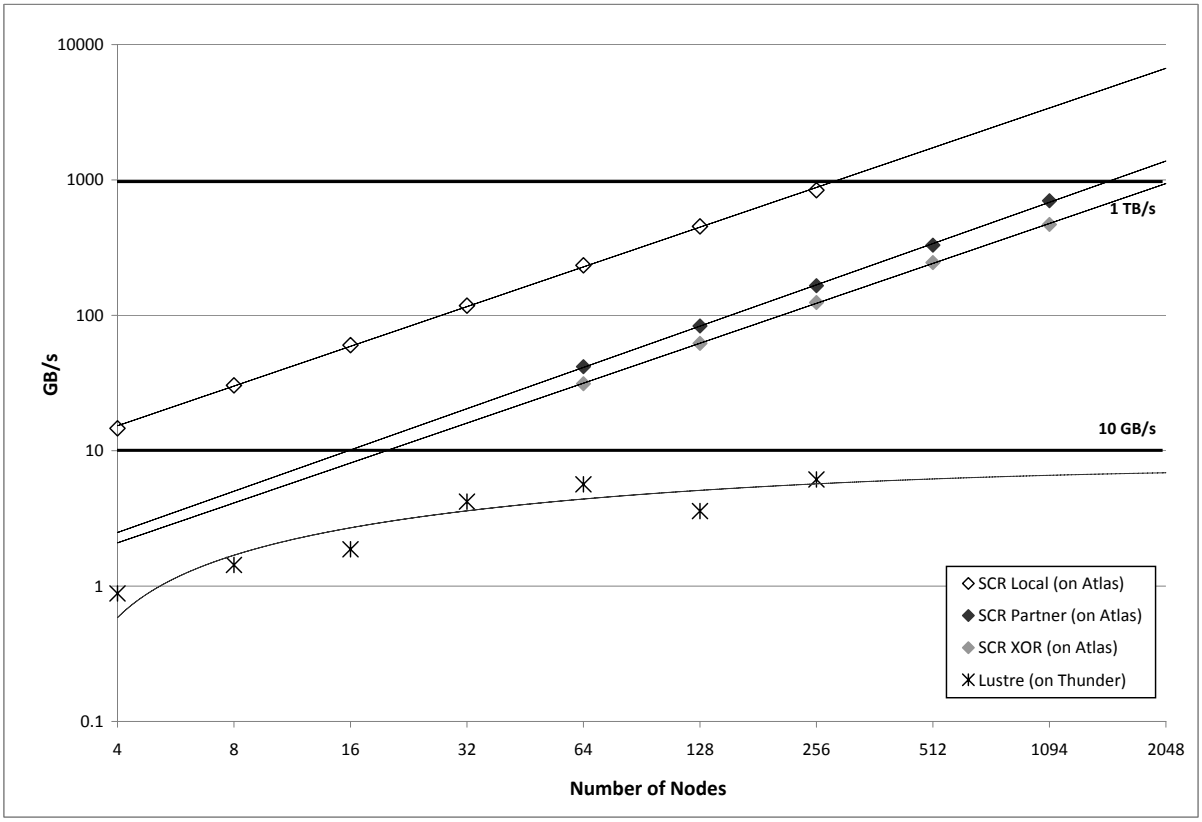


Figure 1: Aggregate I/O bandwidth

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. Such codes can often still utilize the scalable checkpoint capability, but not the scalable restart.

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.

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 as used to write the checkpoint, and each process must only access the files it wrote during the checkpoint.

10. SCR occasionally flushes files from the checkpoint cache to a checkpoint directory on the parallel file system. SCR defines and creates the checkpoint directories within a prefix directory specified by the application. See Section 3.4 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 7 for details.

3 Implementation details

This section defines and discusses concepts one should understand about the scalable checkpoint and scalable restart implementations of SCR including how SCR interacts with the parallel file system. This understanding is necessary in order to properly configure a job. The intent is to automate much of the configuration in future SCR versions, at which time, it will not be necessary to understand many of the following details.

3.1 Scalable checkpoint

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 3.1.4), however, these assumptions simplify the discussion.

In the current implementation, checkpoint files are written to storage on the compute nodes. Because the compute nodes may fail, it is 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 its network connection. However, it can withstand any failure that kills the application process but leaves the node intact, such as application bugs and file I/O errors.

To use `Local`, set the `SCR_COPY_TYPE` parameter 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 than `Local`, and it requires twice the storage space. However, it is capable of withstanding node failures. In fact, it can withstand failures of multiple nodes, so long as a 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_HOP_DISTANCE` parameter.

On many systems, $D = 1$ 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 located physically close to one another 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 hop distances may be chosen to improve reliability at the cost of reducing performance.

To use `Partner`, set the `SCR_COPY_TYPE` parameter to "`PARTNER`".

3.1.3 XOR

With `XOR`, SCR first assigns nodes to disjoint sets, each of size N. The nodes in the same set logically split and pad their checkpoint files and then collectively compute a reduce-scatter operation using bit-wise XOR, as shown in Figure 2. This effectively computes a single XOR parity file from the files on the different nodes, splits the parity file into N equal-sized segments, and scatters one segment to each node. Each node then writes its 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 can withstand multiple node failures so long as two nodes from the same 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 parity 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 parity segment, where the segment size is roughly $1/(N - 1)$ the size of a checkpoint file. Larger XOR sets demand less storage, but they also increase the probability that two nodes in the same set will fail simultaneously. Larger sets may 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` parameter to "`XOR`". The set size currently defaults to a size of 8, however this can be adjusted by setting the `SCR_SET_SIZE` parameter. The `SCR_HOP_DISTANCE` parameter can be used to specify the number of hops between nodes selected to be in the same set. If possible, one should choose the hop distance and set size such that the product of those two values evenly divides the number of nodes in the job.

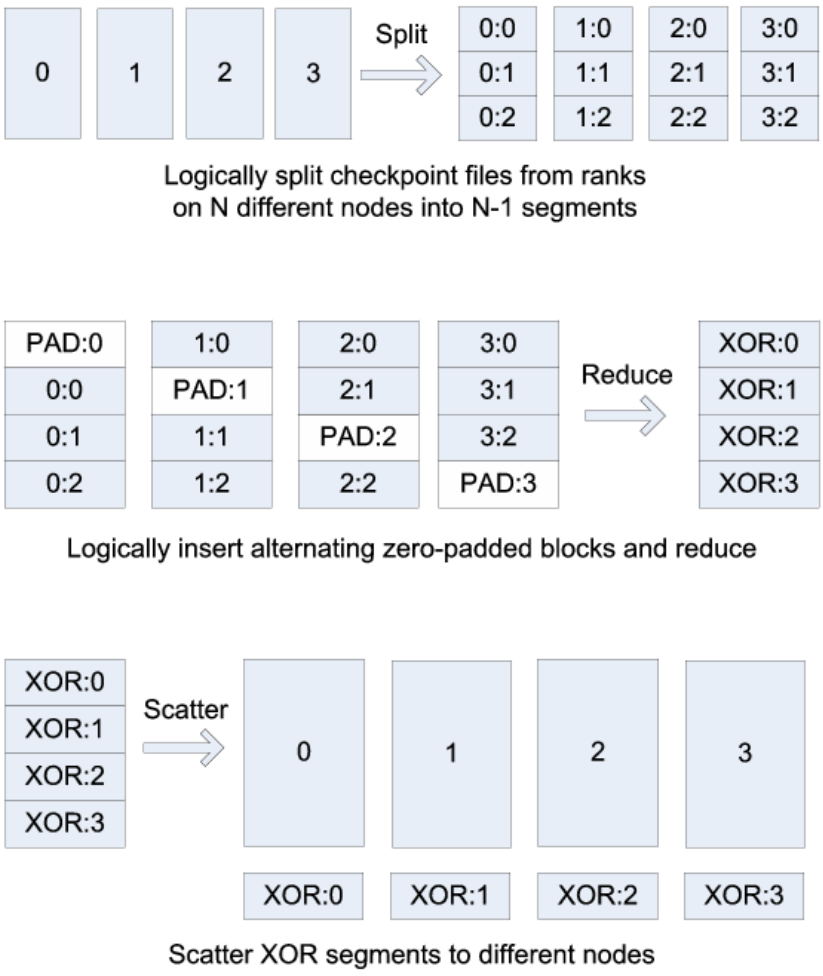


Figure 2: XOR reduce of checkpoint files and scatter of segments

3.1.4 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.

For example, Figure 3 shows how the partner ranks are selected for hop distances of 1 and 2 for a job with 8 nodes with one MPI task on one node and three MPI tasks per node on the other seven nodes. Similarly, Figure 4 shows the XOR set selection for hop distances of 1 and 2 and a set size of 4 for a job with the same process layout.

It is straight-forward to handle `Local` and `Partner` when each process writes an arbitrary number of files each having an arbitrary size. 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 has 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.

3.1.5 Summary

Table 1 summarizes details about the different redundancy schemes, and Figure 5 illustrates the different storage patterns used.

Table 1: Redundancy scheme summary

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

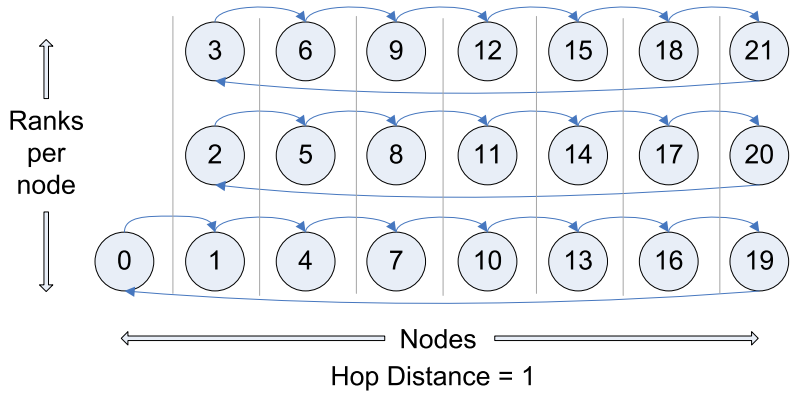
3.2 Scalable restart

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 the cost of writing its checkpoint files out to the parallel file system and then reading them back in for a restart. 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.

Upon encountering a failure, SCR relies on the MPI library, the resource manager, or some other external service to kill the currently running job. Once the job has been killed, and if there are sufficient healthy nodes remaining in the job allocation, the same job can be restarted on the remaining resources of the allocation. In practice, such a restart typically amounts to issuing another "`srun`" or "`mpirun`" in the job batch script.

Of the set of nodes used by the previous run, the restarted run should be configured 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 it ran on during the previous run. As part of the scalable restart, SCR distributes cached files among nodes to match the rank-to-node mapping of the restarted run. SCR includes a set of scripts which encode much of the restart logic, see Section 5.

By default, when a job starts, SCR inspects the cache for existing checkpoints. Starting with the most recent checkpoint in cache, provided that one exists, it attempts to recover the files of that checkpoint. If it succeeds, SCR rebuilds the redundancy data for that checkpoint, and it deletes all older checkpoints from cache. If it fails, it deletes that checkpoint from cache, and it attempts to recover the next most recent checkpoint in cache. It



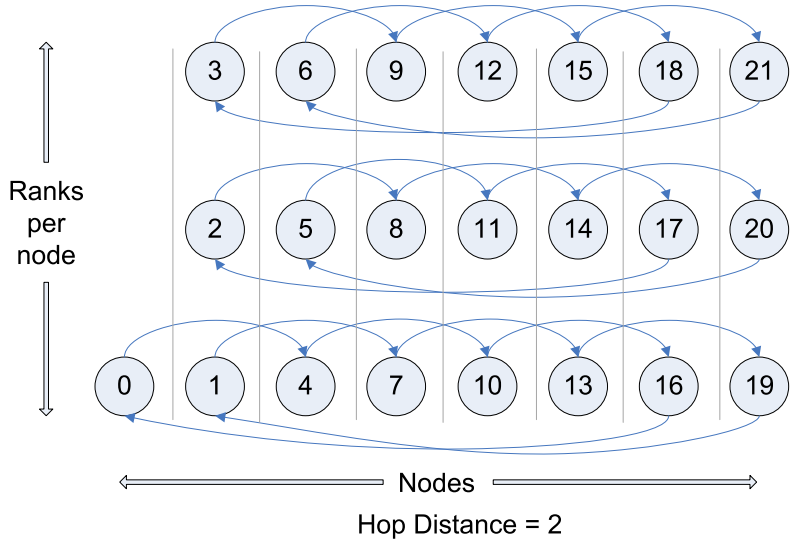


Figure 3: Selecting partner ranks with multiple MPI tasks per node

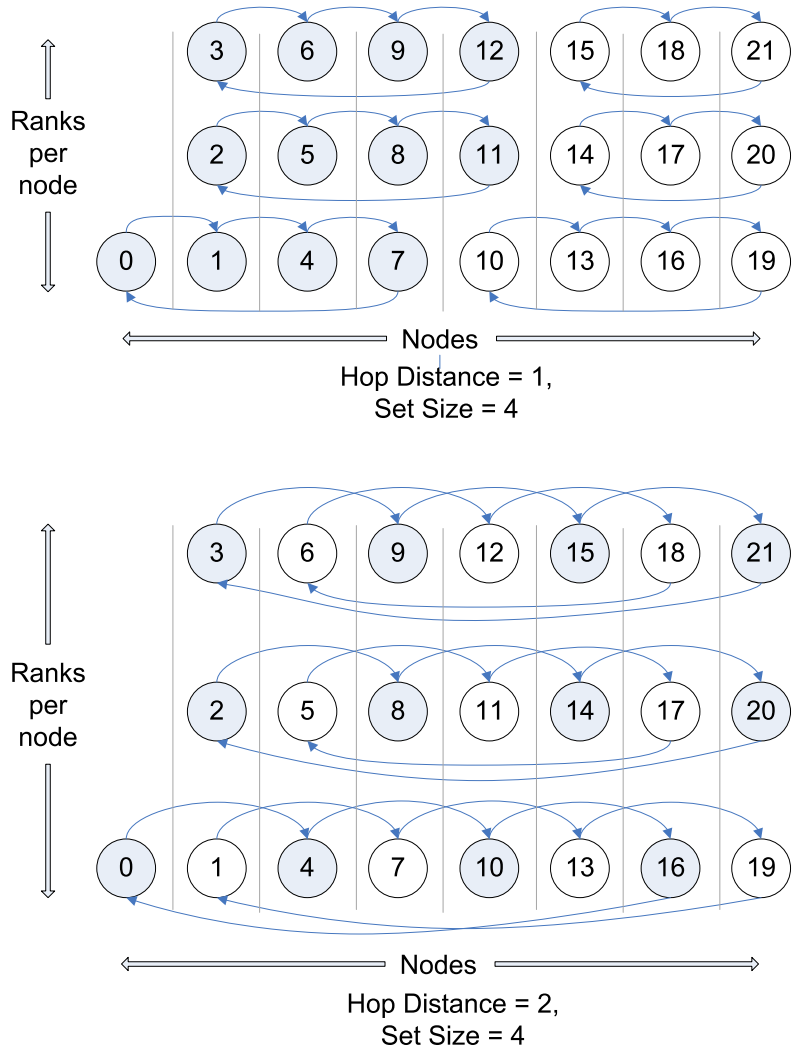


Figure 4: Selecting XOR sets with multiple MPI tasks per node

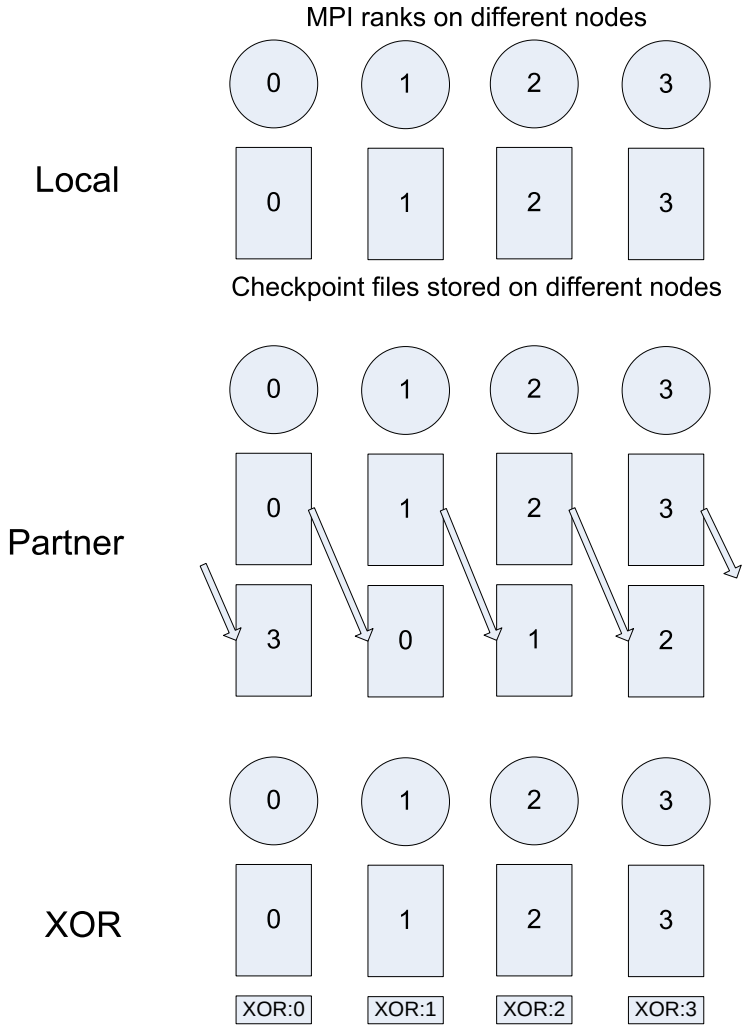


Figure 5: Storage pattern of different redundancy schemes

continues this process until it either succeeds in recovering a checkpoint or it deletes all checkpoints from cache. To disable this feature, set the SCR_DISTRIBUTE parameter to 0.

An example restart scenario is illustrated in Figure 6 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 7.

3.3 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.

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 the node this script is running on fails, the flush logic will not be executed 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 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 parameter. 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. 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.

3.4 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. By default the current working directory is used as the prefix directory. To specify a different prefix directory, set the SCR_PREFIX parameter, e.g.,:

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

Within the prefix directory, SCR creates and names each new checkpoint directory using the following format:

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

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

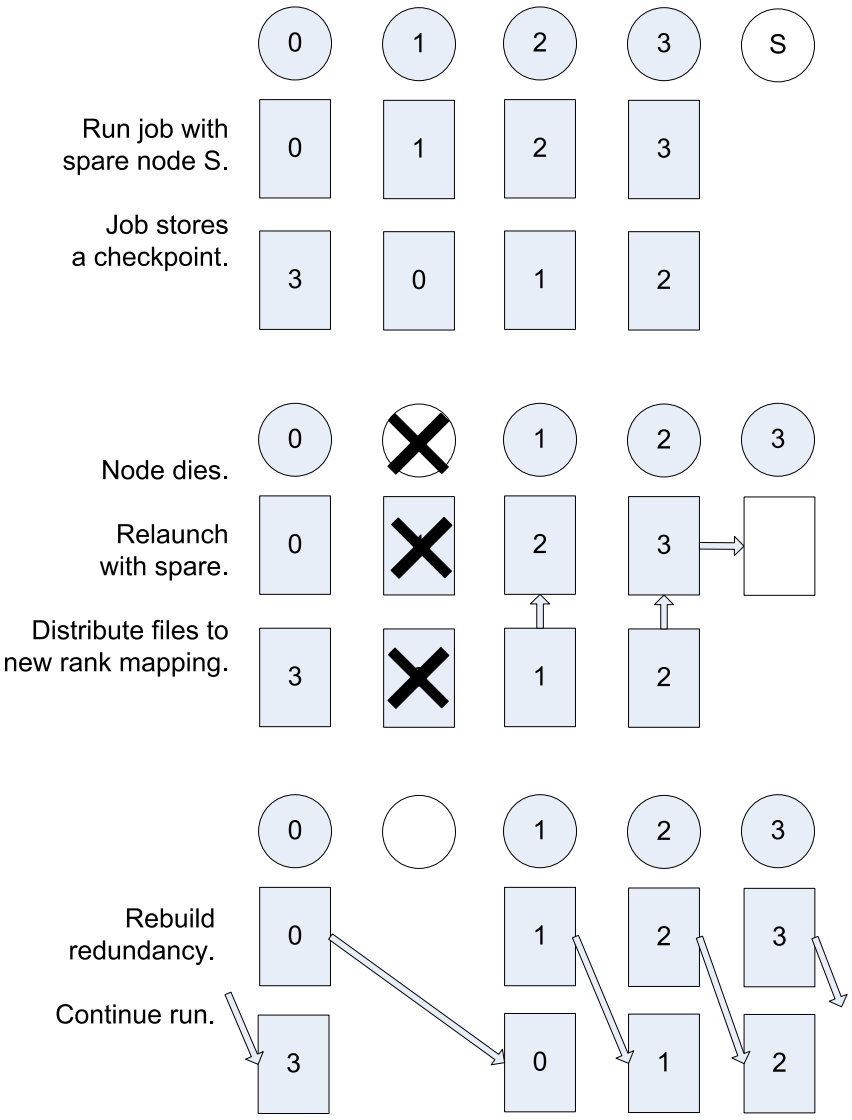


Figure 6: Example restart after a failed node with `Partner`

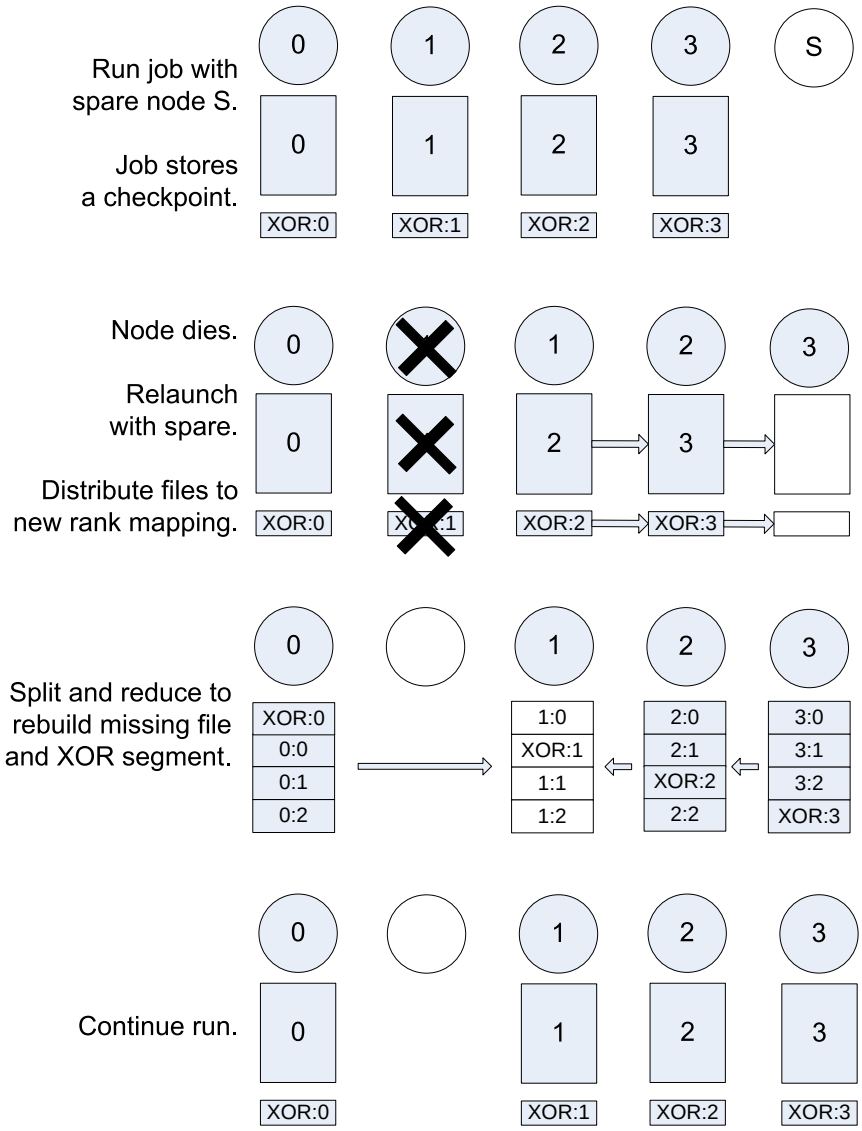


Figure 7: Example restart after a failed node with `XOR`

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 in place, but it 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 on the parallel file system may look like the following:

```
>>: ls -l /p/lscratchb/username/run1/checkpoints
lrwxrwxrwx   1 jdoe jdoe      29 Oct 20 14:23 scr.current -> scr.2008-10-20_14:22:10.23167.50
lrwxrwxrwx   1 jdoe jdoe      29 Oct 20 14:23 scr.old -> scr.2008-10-20_11:30:45.23167.40
drwxr-xr-x   1 jdoe jdoe    4096 Oct 20 14:22 scr.2008-10-20_14:22:10.23167.50
drwxr-xr-x   1 jdoe jdoe    4096 Oct 20 11:30 scr.2008-10-20_11:30:45.23167.40
drwxr-xr-x   1 jdoe jdoe    4096 Oct 14 14:38 scr.2008-10-14_14:38:32.5283.33
drwxr-xr-x   1 jdoe jdoe    4096 Oct 14 13:38 scr.2008-10-14_13:38:32.5283.30
drwxr-xr-x   1 jdoe jdoe    4096 Oct 14 10:35 scr.2008-10-14_10:35:16.5283.20
drwxr-xr-x   1 jdoe jdoe    4096 Oct 09 11:25 scr.2008-10-09_11:25:05.188467.17
drwxr-xr-x   1 jdoe jdoe    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. This file records whether the set is complete and valid, and it records to which MPI rank each file belongs.

3.5 Fetch and flush operations

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 attempts to fetch files from the `scr.current` checkpoint directory to fill the cache. For the fetch operation to succeed 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 the second fetch attempt also fails, SCR deletes `scr.current` symlink, 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` parameter to 0. If an application disables the fetch feature, the application is responsible for reading its checkpoint set directly from the parallel file system upon a restart.

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

As an example of using the fetch and flush features, look again to the example listing of checkpoint directories given in the previous section. In this case, the checkpoint directories were produced by a long-running simulation that ran during three distinct time slots on three different days (Oct 9, 14, and 20) 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, and it stores that value in the meta data file for that file. It then uses the checksum to verify the integrity of each file as it is read back into cache during a fetch. If data corruption is detected in any file, the SCR library falls back to fetch an earlier checkpoint set. To disable this checksum feature, set the `SCR_CRC_ON_FLUSH` parameter 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 the 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. The interpose library intercepts existing application calls to particular POSIX I/O and MPI functions, and it internally calls the appropriate SCR functions before returning control to the application. In such cases, one may use the library with an existing binary without having to modify source code, recompile, or relink.

4.1 The SCR API

The SCR API is designed to support a common checkpointing technique used by large-scale codes, which is globally-coordinated checkpoints written primarily as a file per process. The API is designed to be simple, scalable, and portable; it consists of a handful of C function calls that wrap around existing application checkpoint logic. In most cases, one may fully integrate SCR into an application with fifteen to twenty lines of source 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 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.

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`.

In the current implementation, configuration parameters are processed during `SCR_Init`, process groups are established for redundancy computation, and internal cache and control directories are created. The library also attempts to acquire the latest checkpoint for the job. It first inspects the cache for existing checkpoints as discussed in Section 3.2. If this fails, it attempts to fetch the latest checkpoint from the parallel file system as discussed in Section 3.5. If that also fails, the library prints a warning and continues on with the assumption that the job is new and has not taken a previous checkpoint. Finally, before returning from the `SCR_Init`, MPI rank 0 determines whether the job should be halted (see Section 7) 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`.

4.1.2 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.

In the current implementation, if SCR_FLUSH is enabled and if the latest cached checkpoint needs to be flushed, SCR_Finalize flushes the latest checkpoint to the prefix directory. It also updates the halt file to indicate that SCR_Finalize has been called. This halt condition prevents the job from restarting in the current job allocation.

4.1.3 SCR_Need_checkpoint

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

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.

In addition to guiding an application to the optimum checkpoint frequency on a given platform, this call also enables a running application to react to external commands. For instance, if the application has been instructed to halt, then SCR_Need_checkpoint may indicate that a checkpoint should be taken.

4.1.4 SCR_Start_checkpoint

```
int SCR_Start_checkpoint();
```

Inform SCR that a new checkpoint is about to start. A process must call this function before it opens any files belonging to the new checkpoint. SCR_Start_checkpoint must be called by all processes, including processes that do not write any files as part of the current checkpoint. 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). A process must close all of its checkpoint files before calling SCR_Complete_checkpoint. SCR_Complete_checkpoint must be called by all processes, including processes that did not write any files as part of the current 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 7) 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);
```

A process calls SCR_Route_file to obtain the full path and file name it must use to access a checkpoint file. 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 successfully returning from SCR_Route_file, 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 open the file for reading or writing. A process does not need to create any directories listed in the string returned in file; the SCR implementation will create

the necessary directories before returning from the call. A call to SCR_Route_file is local to the calling process; it is not a collective call. The precise behavior of SCR_Route_file depends on the calling context as described below.

During a checkpoint, a process must call SCR_Route_file after it calls SCR_Start_checkpoint and before it calls SCR_Complete_checkpoint to: 1) register a file as part of the current checkpoint set, and 2) obtain the full path and file name to use to open the file. In this case, it is assumed the file will be opened for writing, so SCR_Route_file returns SCR_SUCCESS even if the file specified by the string returned in file cannot be accessed for reading. SCR returns the same value in file for a given value in name if a process calls SCR_Route_file multiple times within a given SCR_Start_checkpoint / SCR_Complete_checkpoint pair. However, after a process calls SCR_Complete_checkpoint, SCR is free to relocate checkpoint files. Thus, if a process must read a file it just wrote as part of the previous checkpoint, it must call SCR_Route_file again to obtain the (possibly) new path and file name to access the file.

When using SCR during a restart, or to read a file just written as part of the last checkpoint, a process must call SCR_Route_file outside of a SCR_Start_checkpoint / SCR_Complete_checkpoint pair to obtain the full path and file name to open a file. In this case, if a file cannot be accessed for reading, SCR_Route_file returns a value other than SCR_SUCCESS, and the string returned in file is undefined. When successful, SCR_Route_file always returns the full path and file name of the most recent checkpoint of a given file.

In the current implementation, SCR only specifies the directory that should be used to open a file. It strips any directory components listed in name down to the base file name. Then, it prepends the appropriate directory to the base file name and returns the full path and file name in file. Only the directory where the file is opened is changed; the base file name is left unchanged.

4.2 Integrating the SCR API

There are three basic areas to consider when integrating the SCR API into the application 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 if there are multiple calls 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. SCR 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 time step. Then, it should initiate a checkpoint whenever 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 receive instructions from commands external to the job. 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 7).

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` for each file it writes as part of the checkpoint to register each file and to determine the full path and file name to use to open each file. 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 file name for this process
  int rank;
  char name[256];
  MPI_Comm_rank(MPI_COMM_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, on a restart, each MPI task only needs access to the files it wrote during the previous checkpoint, then the application can utilize the scalable restart capability of SCR. This enables the application to restart from a cached checkpoint in the existing job allocation, which saves it the cost of writing to and reading from 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 for reading. If this call succeeds, it always returns the path and name of the most recent checkpoint of a given file.

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 attempts to fetch files from the `scr.current` directory to fill the cache. 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 file name 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];
if (SCR_Route_file(name, file) == SCR_SUCCESS) {
  // Open, read, and close the file
  FILE* fs = open(file, "r");
  size_t n = fread(checkpoint_data, 1, sizeof(checkpoint_data), fs);
  fclose(fs);
} else {
  // There is no existing file to restart from
}
```

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 directory structure used to store checkpoint sets as described in Section 3.4. In particular, to start from the most recent checkpoint, the application should read its checkpoint files from the `scr.current` directory. Be aware that this 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	`-I/usr/local/tools/scr-1.1/include`
Link Flags	`-L/usr/local/tools/scr-1.1/lib -lscr -Wl,-rpath,/usr/local/tools/scr-1.1/lib`

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 run time 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 checkpoints,
- 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 internally it calls `MPI_Init` and then `SCR_Init` before returning. Similarly, it intercepts the application's call to `MPI_Finalize`, and internally calls `SCR_Finalize` and then `MPI_Finalize` before returning.

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 file name 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 regexec. 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 single 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 run time 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 to the parallel file system 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 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	`#MSUB -l resfailpolicy=ignore`
MOAB via `mxterm`	`mxterm ... -l resfailpolicy=ignore`
Interactive SLURM	`salloc --no-kill ...`

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` parameter as described in Section 3.4. On systems with `libyogrt`, it is highly recommended that one also set the `SCR_HALT_SECONDS` parameter so the job allocation does not expire before the latest checkpoint can be flushed (see Section 7).

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` parameter or calling `scr_prerun` with the "`-p`" option, e.g.:

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

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` parameter or calling `scr_postrun` with the "`-p`" option, e.g.:

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

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, it rebuilds missing checkpoint files if there are sufficient redundant data, and it writes the `scr_summary.txt` file.

```
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 7.

```
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 Configuring an SCR job

SCR processes a number of configuration parameters. In most cases, the default configuration will suffice. However, significant performance improvement or additional functionality can often be gained via proper configuration. In the current implementation, this configuration must be done manually. The intent is to automate much of this configuration in future SCR versions.

6.1 Setting parameters

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

1. Environment variables,
2. User configuration file,
3. System configuration file,
4. Compile-time constants.

The location of the system configuration file is hard-coded into SCR at build time. On LLNL systems, one may find this file at `/etc/scr.conf`.

To use a user configuration file, one must specify the name and location of the file by setting the `SCR_CONF_FILE` environment variable at run time, e.g.,

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

A few parameters, such as the location of the control directory, cannot be specified by the user. Such parameters must be set in the system configuration file or be hard-coded into SCR as compile-time constants.

To set an SCR parameter in a configuration file, list the parameter on its own line specifying 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:

```
# set the halt seconds to one hour
SCR_HALT_SECONDS=3600

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

6.2 Configuring multiple checkpoints

In many cases, a single checkpoint configuration suffices. One may fully specify a single checkpoint configuration through any of the means listed above. However, in some cases, it is useful to configure multiple checkpoints within a single job. For example, one may configure SCR to write some checkpoints to RAM disk and others to solid-state drives, or one may configure some checkpoints to use one size of XOR set and others to use a different size of XOR set. When running with multiple cache locations or multiple checkpoint types, one must specify the configuration in a configuration file. In addition, one must set the `SCR_COPY_TYPE` parameter to "`FILE`".

NOTE: This method is likely to change in future releases to make it more consistent with other configuration settings.

To use multiple cache directories in a single run, one must specify a series of cache descriptors. A cache descriptor must specify the base directory and the maximum number of checkpoints to keep in that directory. Each descriptor must be listed on a single line, and the series must be numbered sequentially starting from 0, e.g.:

```
# the following instructs SCR to keep at most 1 checkpoint in /tmp and 3 in /ssd
CACHEDESC=0 BASE=/tmp SIZE=1
CACHEDESC=1 BASE=/ssd SIZE=3
```

To use multiple checkpoint types, one must specify a series of checkpoint descriptors. The descriptor should specify the base directory of the cache, the checkpoint interval, the copy type, and other associated parameters, such as the hop distance and set size. Each descriptor must be listed on a single line, and the series must be numbered sequentially starting from 0, e.g.:

```
# instruct SCR to use the CKPTDESC list
SCR_COPY_TYPE=FILE

# the following instructs SCR to run with three checkpoint configurations:
# - save every 8th checkpoint to /ssd using the PARTNER scheme with a hop distance of 2
# - save every 4th checkpoint (not divisible by 8) to /ssd using XOR with a hop distance
#   of 4 and set size of 8
# - save every other checkpoint to /tmp using XOR with a hop distance of 1 and set size
#   of 16
CKPTDESC=0 BASE=/tmp INTERVAL=1 TYPE=XOR     HOP_DISTANCE=1 SET_SIZE=16
CKPTDESC=1 BASE=/ssd INTERVAL=4 TYPE=XOR     HOP_DISTANCE=4 SET_SIZE=8
CKPTDESC=2 BASE=/ssd INTERVAL=8 TYPE=PARTNER HOP_DISTANCE=2
```

To determine which checkpoint descriptor to apply in a given checkpoint, SCR selects the checkpoint descriptor that has the largest interval value that evenly divides the internal SCR checkpoint iteration number. It is necessary that one checkpoint descriptor has an interval of 1.

6.3 SCR parameters

Table 4: SCR parameters

Name	Default	Description
SCR_ENABLE	1	Set to 0 to disable SCR at run time.
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 in the current job allocation is less than the specified number of seconds.
SCR_COPY_TYPE	XOR	Set to one of: LOCAL, PARTNER, XOR, or FILE.
SCR_CACHE_BASE	2	Specify the base directory SCR should use to cache checkpoints.
SCR_CACHE_SIZE	2	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_HOP_DISTANCE	1	Set to a positive integer to specify the number of hops taken to select a partner node for PARTNER or the number of hops between nodes of the same XOR set for XOR. 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).
SCR_SET_SIZE	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. However, higher values have an increased likelihood of encountering a catastrophic error. Higher values may also require more time to reconstruct lost files from redundancy data.
SCR_PREFIX	N/A	Specify the prefix (root) directory on the parallel file system where checkpoint directories should be read from and written to.
SCR_DISTRIBUTE	1	Set to 0 to disable file distribution during SCR_Init. File distribution enables the application to replace failed nodes with spare nodes during a restart from the checkpoint cache.
SCR_FETCH	1	Set to 0 to disable SCR from fetching files from the parallel file system to bootstrap the compute node cache during SCR_Init.
SCR_FETCH_WIDTH	256	Specify the number of processes that may read simultaneously from the parallel file system.
SCR_FLUSH	10	Specify the number of checkpoints between periodic SCR flushes to the parallel file system. Set to 0 to disable periodic flushes.
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 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 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 in an allocation 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 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_MPI_BUF_SIZE	131072	Specify the number of bytes to use for internal MPI send and receive buffers when computing redundancy data or rebuilding lost files.

Table 4 – continued from previous page

Name	Default	Description
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.
SCR_CRC_ON_DELETE	0	Set to 1 to enable CRC32 checking when deleting files from cache.
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.

7 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.

7.1 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`.

7.2 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
```

7.3 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 parameter 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 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 parameter. 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 parameter.

7.4 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.

7.5 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.

7.6 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
```

7.7 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
```

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.