How can the Scalable Checkpoint / Restart (SCR) library help MPI users?

9th Annual MVAPICH User Group (MUG) Meeting

Adam Moody
Livermore Computing

August 23, 2021

LLNL-PRES-825999
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Why use SCR in your MPI job?

Guarantees* that your job always runs for its full allotted time

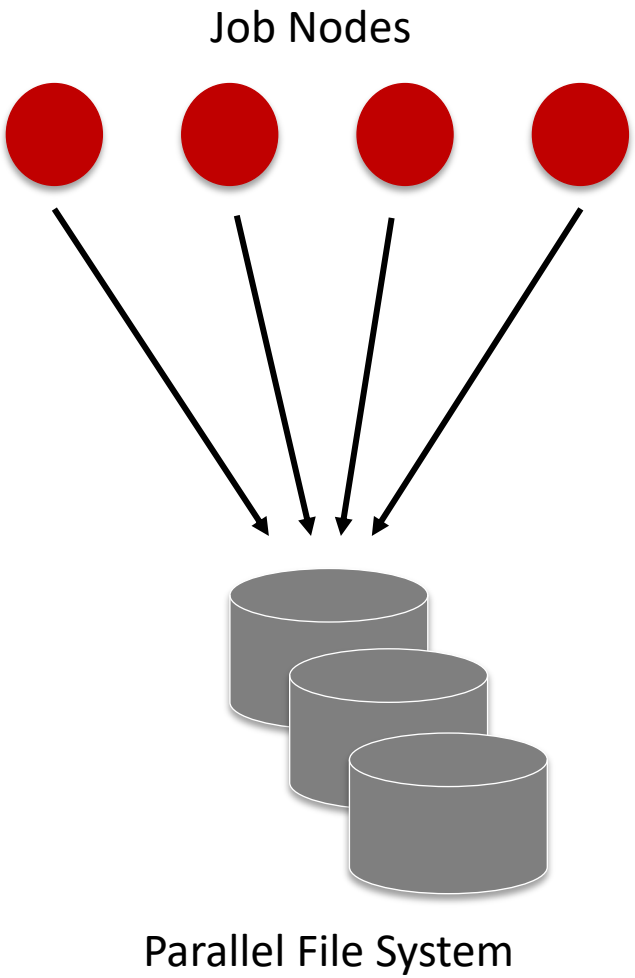
- Get your full allotted time
- During every allocation
- Even in the event of compute node failures (with spare nodes)

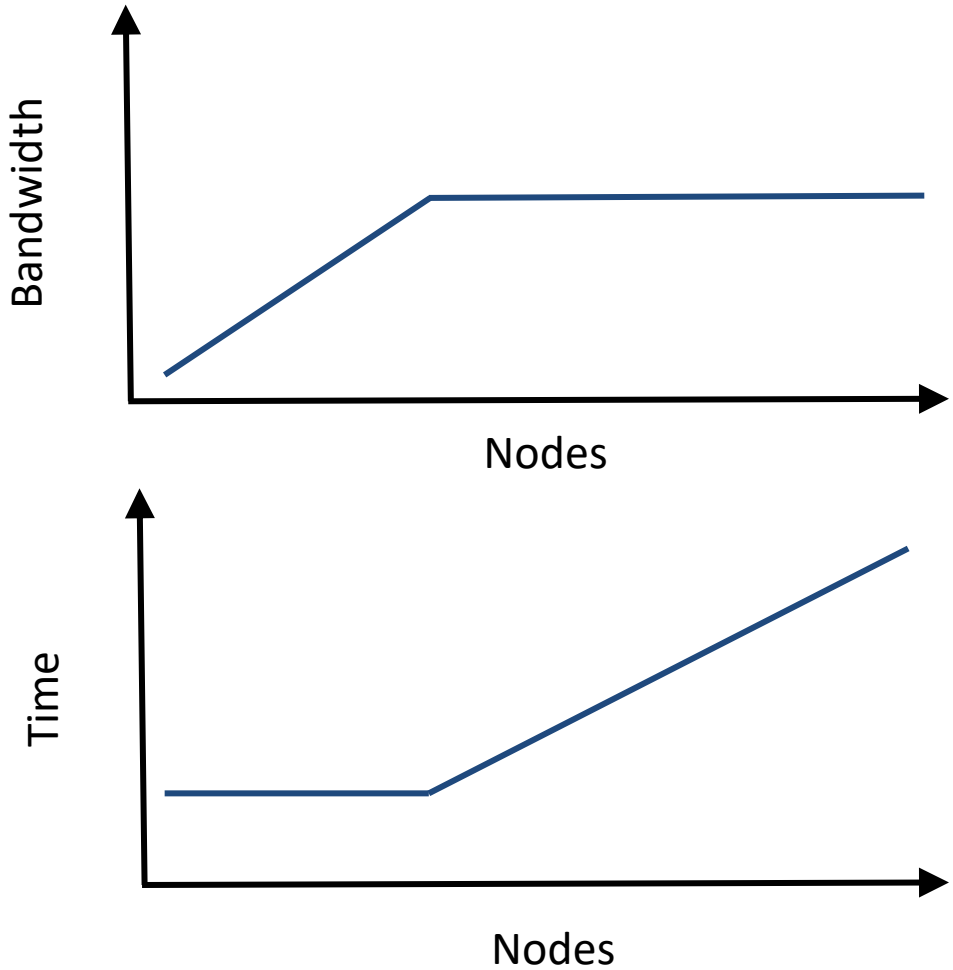
Guarantees* that your job produces results faster

- Less time writing output -> more time computing
- Less time writing checkpoints -> more time computing
- Less time reading checkpoints -> more time computing
- Checkpoint more often -> less work lost from each interruption

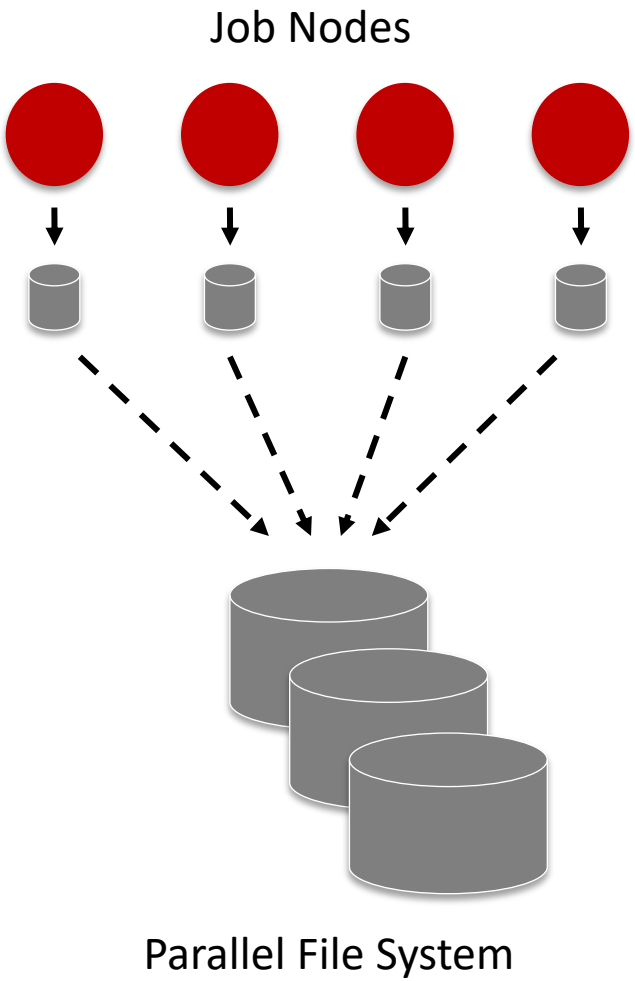
* By guarantee, we mean very highly likely, not absolutely certain. Please don't sue us.

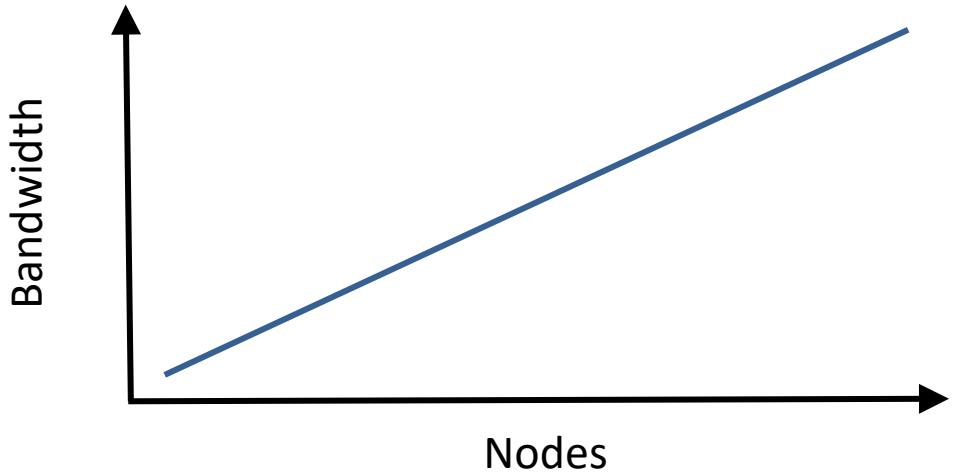
Write bandwidth to the parallel file system typically saturates at low node counts

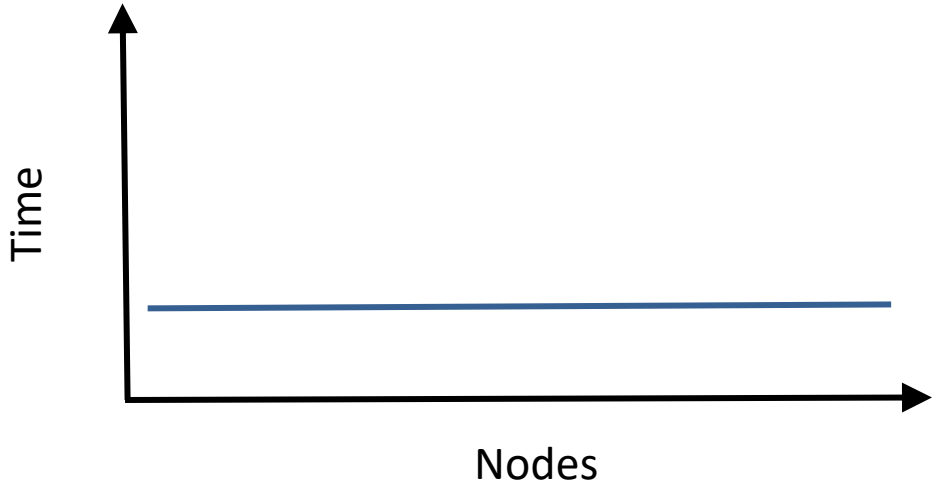




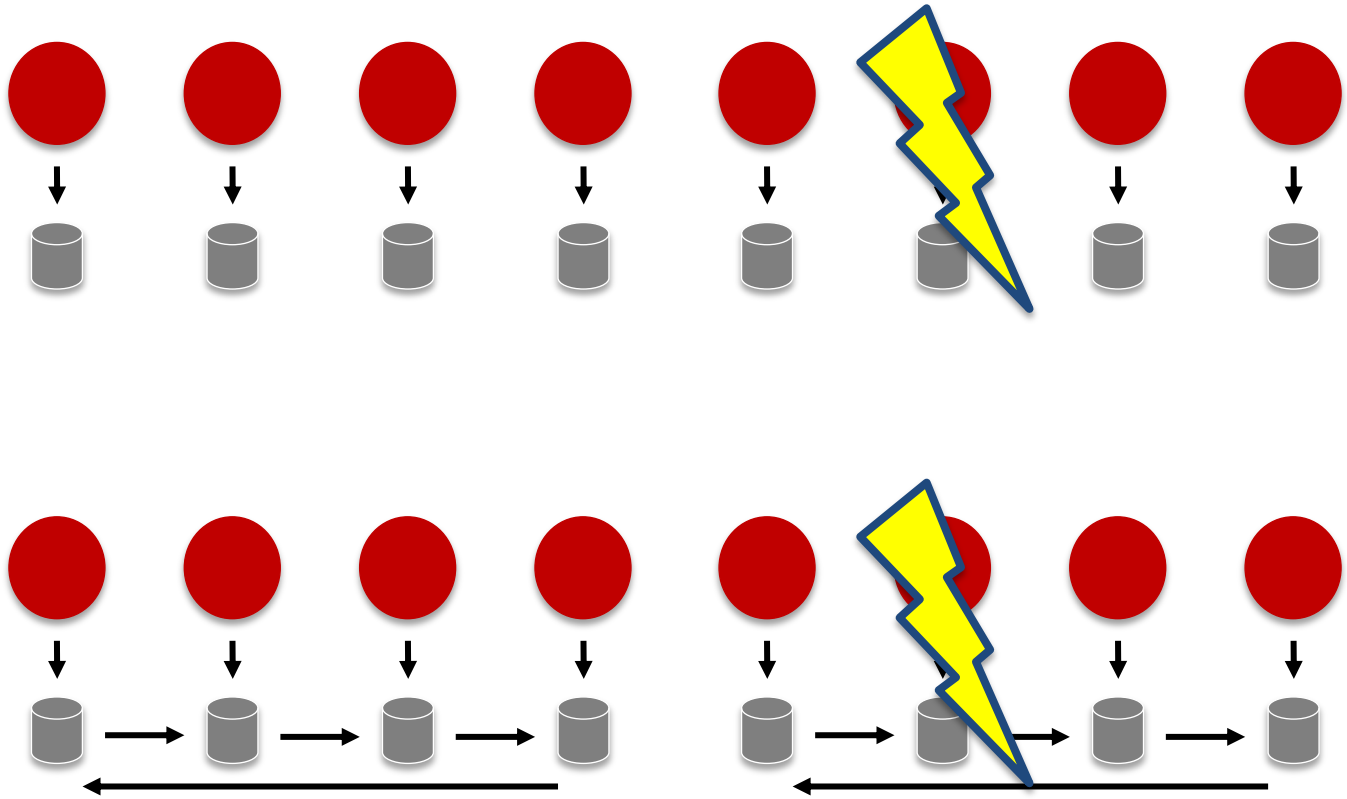
One can achieve scalable bandwidth by writing to node-local storage (then copy in background)



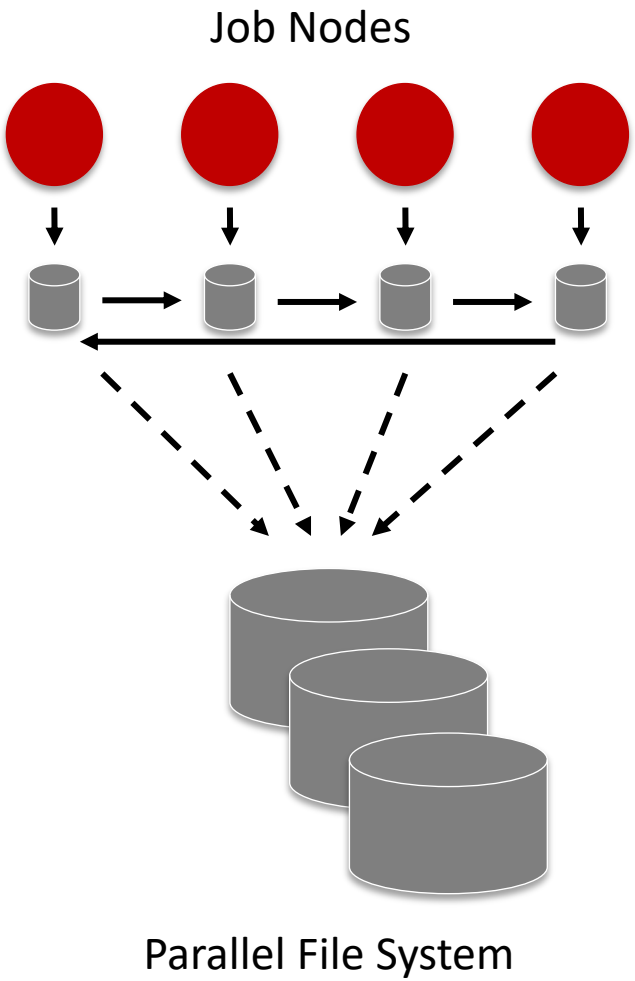


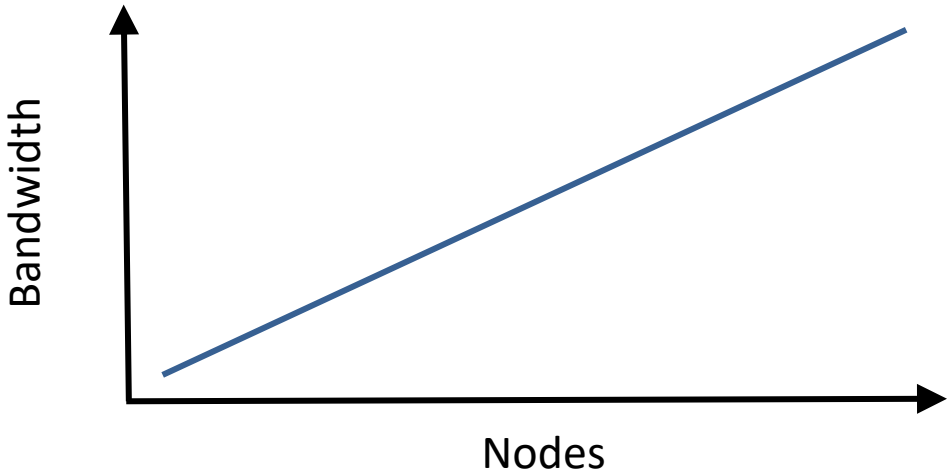


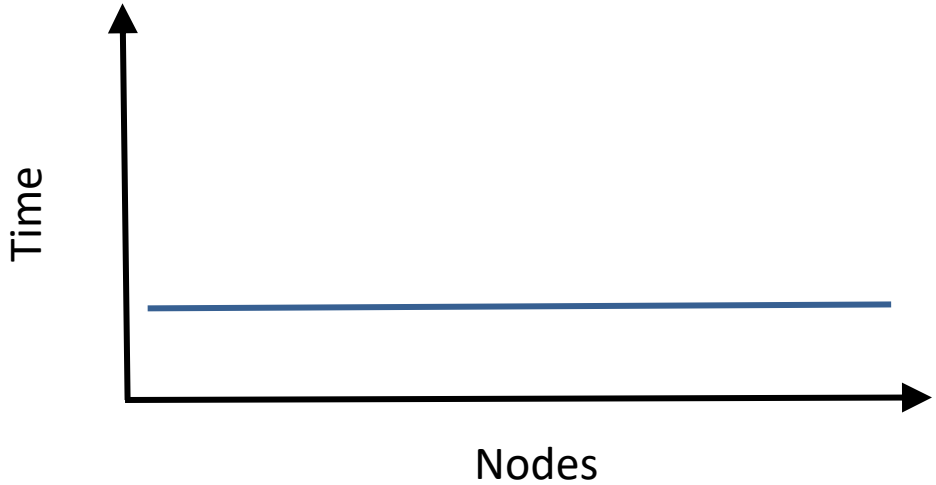
But since node failure can result in data loss, one also needs to have data redundancy



SCR in a nutshell: write files to node-local storage, apply redundancy, copy to PFS

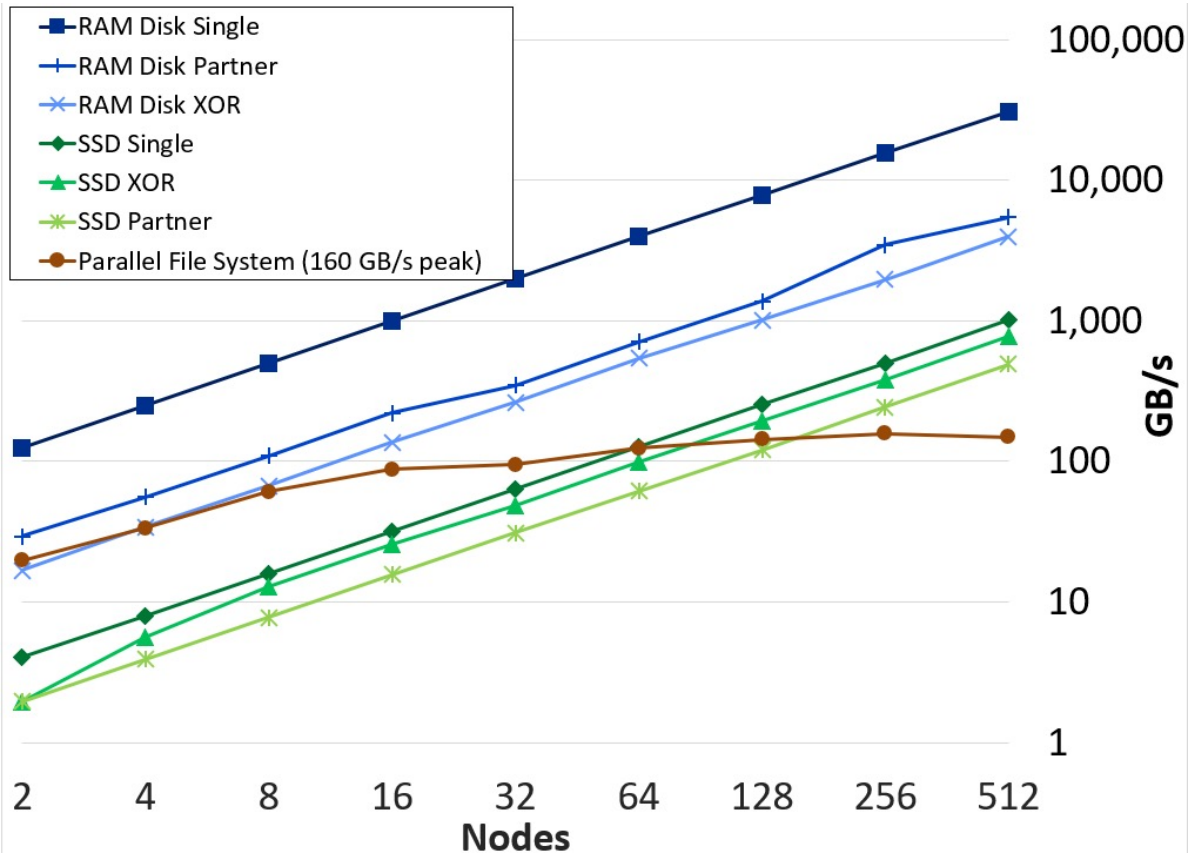






SCR performance vs. the parallel file system increases with scale

- SCR supports multiple node-local storage locations, e.g.,
  - /dev/shm
  - SSD
- SCR implements multiple redundancy schemes, e.g.,
  - Single
  - Partner
  - XOR
- Orders of magnitude faster at large scale
  - 10-100x at 512 nodes



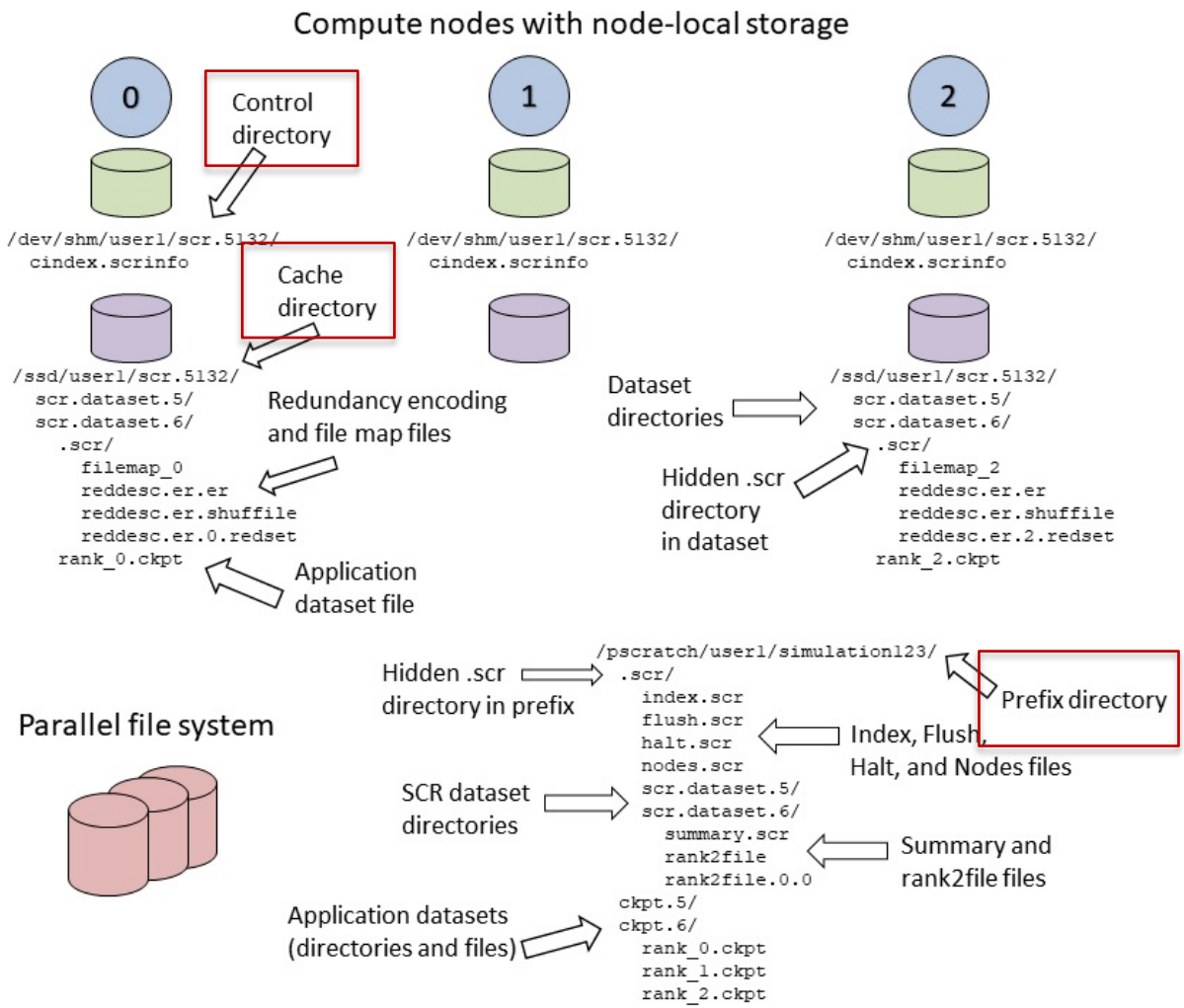
Effective write bandwidth on LLNL's Lassen system with /dev/shm, SSD, and parallel file system

SCR is a library to accelerate write bandwidth and job script commands to enable fast restarts

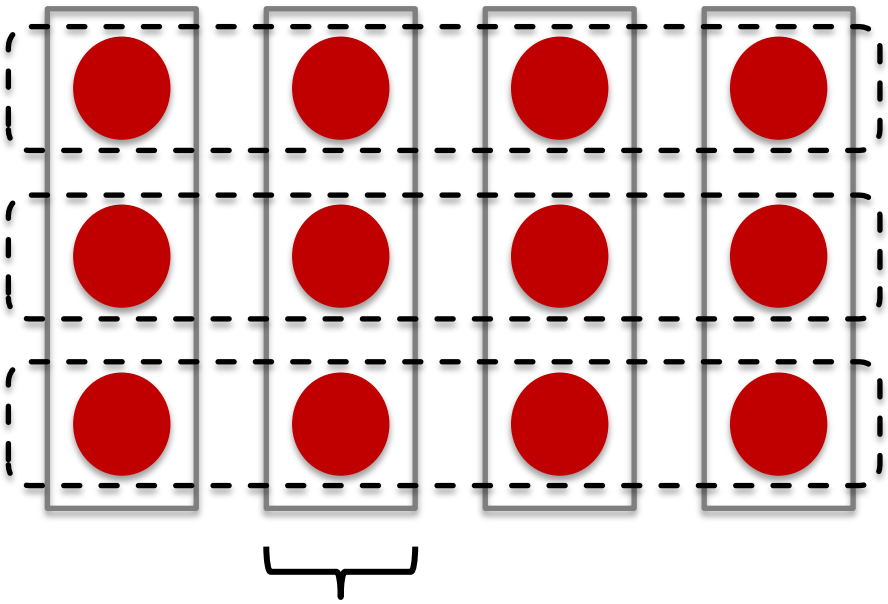
- Library to accelerate I/O write bandwidth for MPI applications
  - Enables scalable write bandwidth
  - 10-100x faster than the parallel file system
  - 7-10 API calls to integrate into existing application I/O logic
  - Bindings for C/C++, Fortran, and Python
- Commands to add to the job batch script (optional)
  - Detect and recover from application failures, system failures, and application hangs
  - Relaunch application within the same job allocation, even after node failures
- Provides the most benefit for:
  - MPI apps that write file-per-process in globally synchronous I/O phases
  - Large-scale jobs that write lots of data
  - Especially valuable if application or system failures are common

SCR refers to node-local storage as “cache” and stores files on PFS in a “prefix” directory

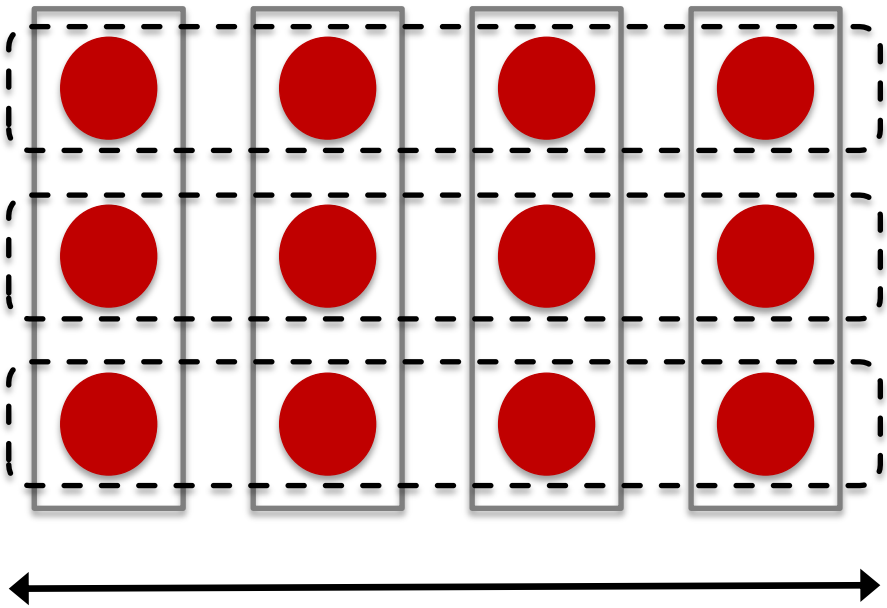
- Control directory stores internal SCR state and uses /dev/shm for speed
- Cache directories store dataset files and redundancy data, can use more than one cache directory in a run
  - /dev/shm
  - SSD
- All dataset files must be written somewhere within the prefix directory on PFS
  - Must be unique to a simulation
  - Files must be distinct between datasets (no files shared across datasets)
  - Otherwise supports arbitrary directory and file layouts
- Transfers between cache and PFS
  - Fetch: PFS to cache
  - Flush: cache to PFS



Processes are split across “failure groups” into “redundancy sets” of size N



Failure group:
likely to fail at same time



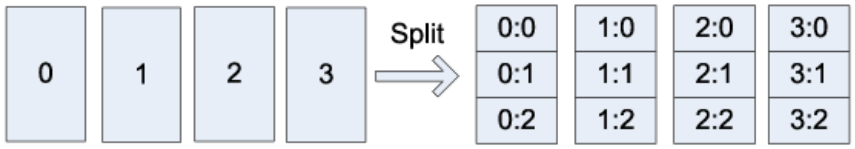
N procs per redundancy set

Default failure group is all procs on the same compute node (same hostname). Other failure groups can be defined.

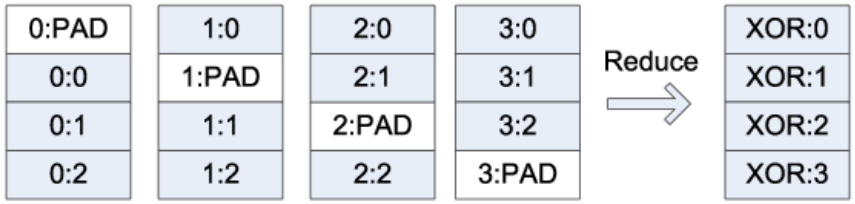
Processes in the same redundancy set are selected from different failure groups.

Example: encoding for XOR redundancy scheme

- Each process writes file of B bytes
- Logically divide files into N-1 segments
- Insert zero-padded segment in alternating positions
- Execute reduce-scatter with XOR
- Each process stores
  - original file of B bytes
  - XOR redundancy data of size B/(N-1)
- Can support arbitrary file sizes and an arbitrary number of files per process by logically concatenating source files



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



Logically insert alternating zero-padded chunk and reduce

Scatter XOR chunks among the different ranks

Different redundancy schemes have different storage requirements and failure tolerance

- SCR defaults
  - N = 8
  - k = 2
  - scheme: XOR
  - Values can be adjusted
- e.g., XOR requires
  = B * N / (N – 1)
  = B * 8 / 7
  = B * (1 + 1/7)

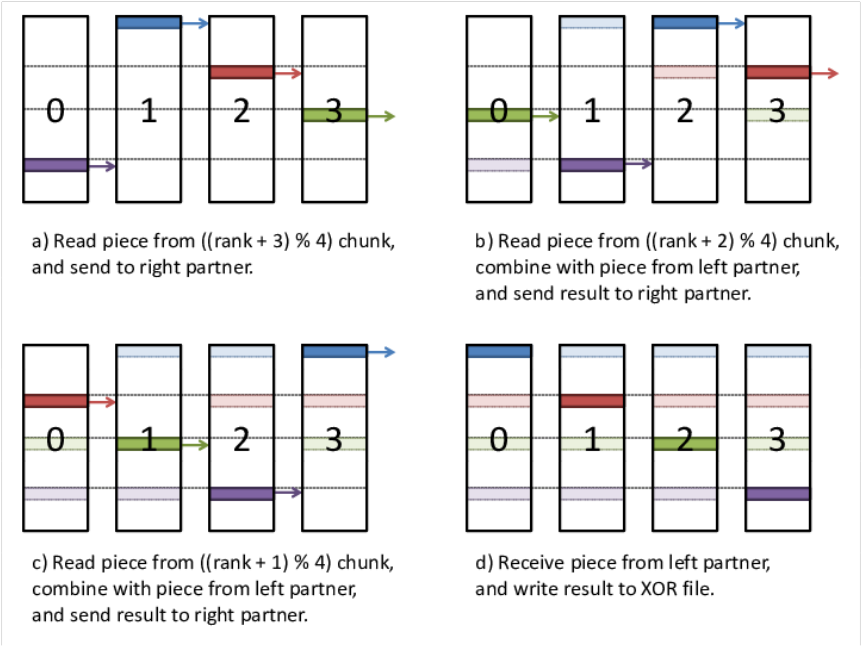
  1/7 = ~14% extra space

Redundancy scheme	Storage requirements per process	Maximum failures per set
Single	B	0
Partner	B * 2	1+
XOR	B * N / (N – 1)	1
Reed-Solomon	B * N / (N – k)	k with 1 <= k < N

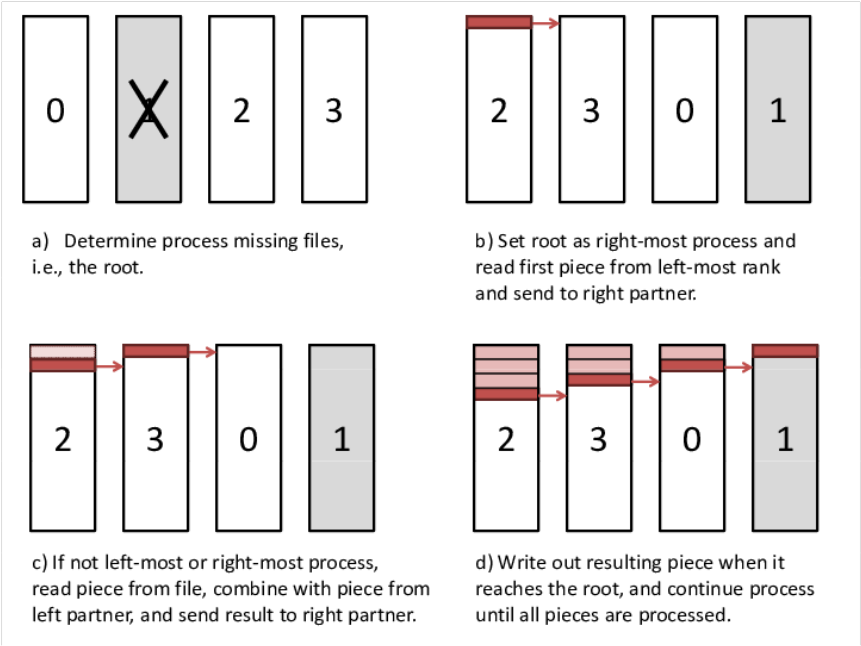
Summary of redundancy schemes assuming each process writes B bytes and is grouped with other processes into a set of size N

SCR uses MPI for efficient encode/rebuild operations

Parallel XOR Encode



Pipelined XOR Rebuild



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

SCR operates within an MPI environment, and most calls are collective over COMM_WORLD

- SCR_Init
  - Call after MPI_Init
  - Rebuilds cached datasets
  - Fetches checkpoint from file system
- SCR_Need_checkpoint (optional)
  - Whether job should checkpoint
  - Guide to optimal checkpoint frequency
  - Accounts for write cost, and expected failure rate
- SCR_Should_exit (optional)
  - Tracks allocation end time
  - Stop early so SCR can flush cached datasets to file system
- SCR_Finalize
  - Call before MPI_Finalize
  - Flushes cached datasets to file system

```
#include "mpi.h"
#include "scr.h"

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

  for (int t = start; t < end; t++)
  {
    // do work for timestep "t" …

    int need_ckpt;
    SCR_Need_checkpoint(&need_ckpt);
    if (need_ckpt) checkpoint();

    int should_exit;
    SCR_Should_exit(&should_exit);
    if (should_exit) break;
  }

  SCR_Finalize();
  MPI_Finalize();
}
```

SCR Output API supports both checkpoint and non-checkpoint datasets

- SCR_Start_output
  - Initiates an output phase
  - Provide "name" for the dataset
  - Bit flags:

    `SCR_FLAG_OUTPUT`
    dataset must be copied to file system

    `SCR_FLAG_CHECKPOINT`
    dataset can be used to restart the application

- SCR_Route_file for each file
  - Local to each process (not collective)
  - Caller can write zero or more files
  - Caller provides path to file on PFS
  - Registers file as member of output set
  - SCR returns path to open file

- SCR_Complete_output
  - Terminates the output phase
  - Caller indicates whether it wrote its file(s) successfully
  - SCR applies redundancy scheme, initiate flush
  - Returns SCR_SUCCESS if all procs succeeded

```
char name[] = "timestep.1";
int flags = SCR_FLAG_OUTPUT;
SCR_Start_output(name, flags);

char file[256];
sprintf(file, "%s/rank_%d.ckpt", dir, rank);

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

int valid = 1;
FILE* fs = fopen(scr_file, "w");
if (fs != NULL) {
  int rc = fwrite(data, ..., fs);
  if (rc == 0) valid = 0;

  fclose(fs);
  if (rc != 0) valid = 0;
} else {
  valid = 0;
}

SCR_Complete_output(valid);
```

SCR Restart API supports looping in case application detects a problem

- SCR_Have_restart
  - Indicates whether SCR has a checkpoint
- SCR_Start_restart
  - Initiates a restart phase
  - Returns name of loaded checkpoint
- SCR_Route_file for each file
  - Local to each process (not collective)
  - Caller can read zero or more files
  - Caller provides path to file on PFS
  - SCR returns path to open file
- SCR_Complete_restart
  - Terminates the restart phase
  - Caller indicates whether it read its file(s) successfully
  - Returns SCR_SUCCESS only if all procs succeeded

```
int restarted = 0;
while (! restarted) {
  int have_restart;
  char name[SCR_MAX_FILENAME];
  SCR_Have_restart(&have_restart, name);
  if (! have_restart) break;

  SCR_Start_restart(name);

  char file[256];
  sprintf(file, "%s/rank_%d.ckpt", dir, rank);

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

  int valid = 1;
  FILE* fs = fopen(scr_file, "r");
  if (fs != NULL) {
    int rc = fread(data, ..., fs);
    if (rc == 0) valid = 0;

    fclose(fs);
  } else {
    valid = 0;
  }

  int scr_rc = SCR_Complete_restart(valid);
  restarted = (scr_rc == SCR_SUCCESS);
}
```

Applications can increase checkpoint frequency, while decreasing their checkpoint overhead

- How SCR handles datasets marked with SCR_FLAG_CHECKPOINT
  - Always flush any checkpoint also marked with SCR_FLAG_OUTPUT
  - Flush every N^{th} checkpoint, according to SCR_FLUSH parameter (default=10)
  - Flush most recent cached checkpoint during SCR_Finalize, or "scavenge" most recent checkpoint from job script in case SCR_Finalize.
- Applications can use "purely defensive" checkpoints to maximize benefit.
- SCR can be configured to discard non-output checkpoints.
  - Discard an older checkpoint from cache when a new checkpoint is written.
  - No need to write checkpoint to the file system.
  - Enables app to checkpoint at higher frequency than parallel file system allows.

Cost 20 mins ⟹ 4 secs

Freq 4 hours ⟹ 15 mins

SCR run wrapper detects failed nodes, restarts on spare nodes, and scavenges cached datasets

Original application job script

```
#!/bin/bash
#SBATCH --nodes 128

srun -n 128 -N 128 ./app
```

SCR-enabled job script

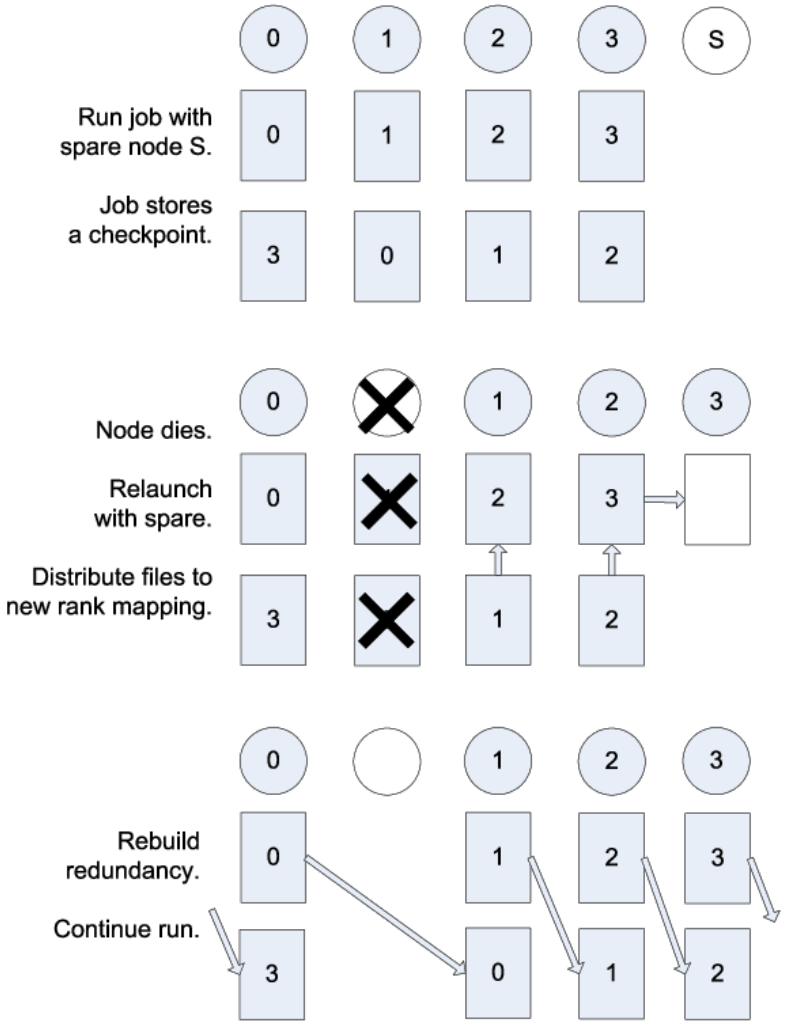
```
#!/bin/bash
#SBATCH --nodes 129
#SBATCH --no-kill

scr_srun -n 128 -N 128 ./app
```

scr_srun relaunches the app if possible

```
#!/bin/bash

# prepare job allocation for SCR
scr_prerun

while [ 1 ] ; do
    # launch user job, avoid failed nodes
    srun --exclude $failed "$@"

    # above command returns on any failure
    # or detected hang, when using a watchdog

    # run ended, determine whether to stop
    if "run completed normally" or
       "user requested halt" or
       "out of healthy nodes" or
       "out of time" ; then
      break
    fi

    # identify failed nodes, if any
    failed=`scr_list_down_nodes`
done

# scavenge cached datasets, if needed
scr_postrun
```

“Scalable restart” can use spare nodes to rebuild cached checkpoints from redundancy data

- Allocate an extra node or two
- Spare nodes not used initially
- If a node dies, current run dies
- Add logic to batch script to
  - Detect any down nodes
  - Start new run on remaining nodes
- Benefits of scalable restart
  - Continue running in same allocation so long as have enough spares
  - No need to write data to parallel file system to read it back, restart in place
  - Restart large-scale simulations in seconds
- Fall back to “scavenge” and exit if:
  - Not enough healthy nodes
  - Out of time
  - Been told to exit



New in v3.0: Cache bypass mode

- Option to configure SCR to bypass cache
  - Route_file directs app to read/write directly from/to parallel file system
  - Counter-intuitive, but really useful (in fact, new default mode)
  - Lose cache performance, though enables one to still use other features, like automated restart logic, spare nodes, checkpoint tracking, logging
- Enables an application to always use SCR, even on systems without cache and for problems that are too big
  - Use SCR on any system, no hard requirement for cache hardware
  - Simpler app integration, no need for developers to write one code path with SCR and one without
- Enables applications that read or write shared files to use SCR
  - file-per-process is still the fastest, but it's no longer a requirement

New in v3.0 (cont.)

- Python bindings added
- Asynchronous flush improvements
  - pthreads, IBM Burst Buffer API, Cray Datawarp
  - Support multiple outstanding async flushes
- Reed-Solomon encoding
  - Simplify configuration in cases where multi-node failures are likely
- Preserve file metadata during flush
  - permission bits, timestamps, and user/group ownership
- Logging (syslog and text)
  - Log output dataset byte size, time, proc/file count
  - Log node failures
  - Collect I/O stats about jobs (e.g., burst size)
  - Compute optimal checkpoint frequency per application

```
from mpi4py import MPI
import scr

scr.init()
for t in range(start, end):
  // do work for timestep t …

  if scr.need_checkpoint():
    name = f"ckpt_{t}"
    scr.start_output(name, scr.FLAG_CHECKPOINT)

    rank = MPI.COMM_WORLD.rank
    filename = f"{name}_{rank}"
    scr_file = scr.route_file(filename)

    valid = True
    try:
      with open(scr_file, "w") as f:
        f.write(data)
    except Exception:
      valid = False

    scr.complete_output(valid)

  if scr.should_exit():
    break
scr.finalize()
```

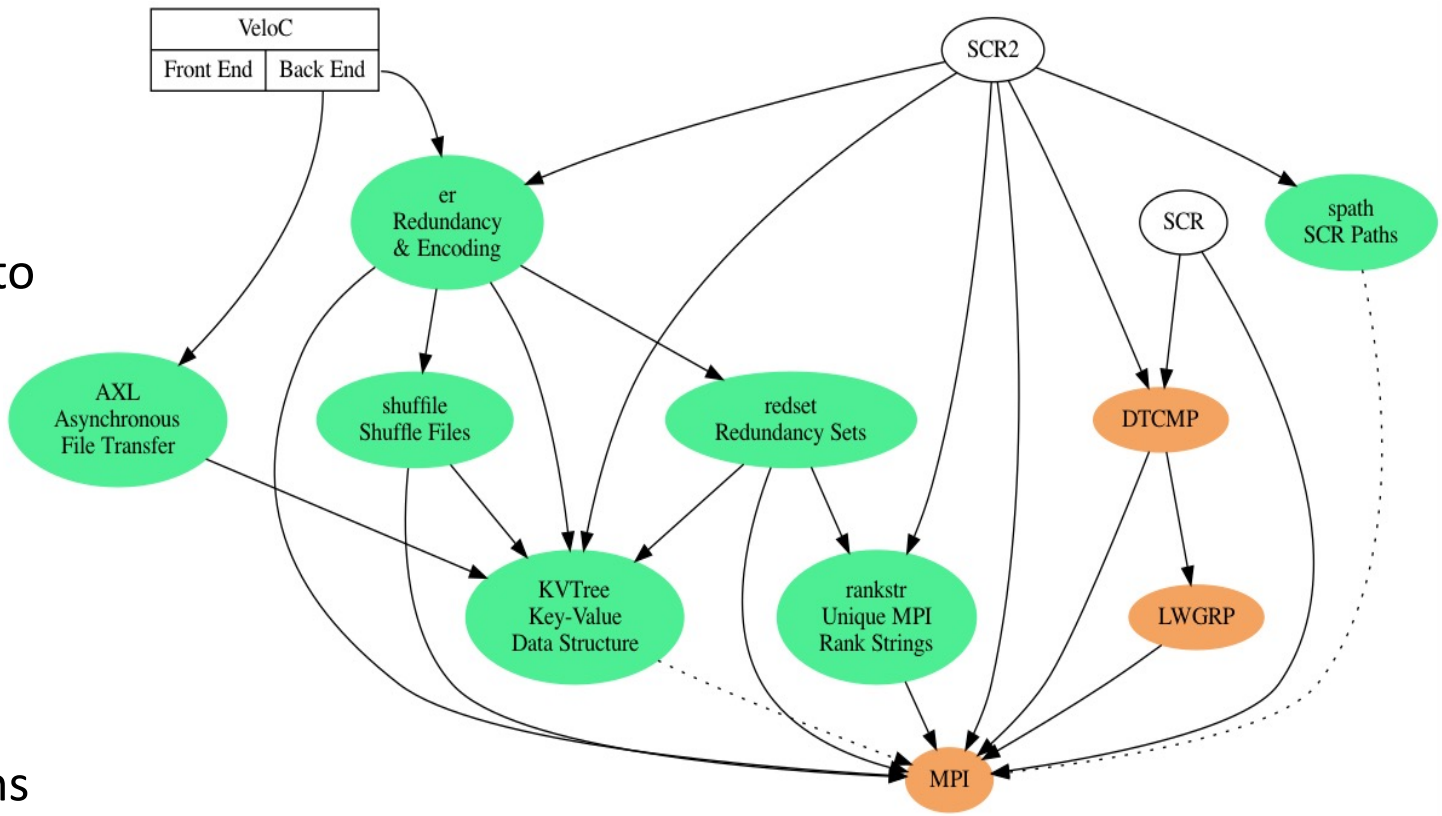
VeloC Exascale Computing Project (ECP) software components shared with SCR

Joint development effort between Argonne and LLNL

https://github.com/ECP-VeloC

Goals:

- Combine SCR and FTI APIs into a single library
- Execute encoding step in background
- Integrate into ECP applications



X-ScaleSolutions awarded Phase 1 DOE SBIR to build SCR-Exa

- LLNL SCR team is working as a subcontractor for X-ScaleSolutions to develop SCR-Exa
- Port SCR-Exa to new HPC sites and customers
- Refactor and rewrite SCR job script commands
- Simplify task to add new resource managers, job launchers, node health checks, remote command execution
- And more… see Donglai's talk tomorrow

SCR supplies lots of logic to keep your job jogging

- Fast output, checkpoints, and restarts
- Data redundancy schemes
- Auto discard, auto flush checkpoints
- Hang detection
- Detect failed nodes
  - Ping
  - Per node health checks
  - User control to exclude nodes
- Relaunch on healthy nodes
  - mpirun, srun, jsrun, aprun
- Restart-in-place
- Scavenge w/ rebuild
- Track checkpoint sequence
  - Restart from most recent
- Rollback to older checkpoint
  - On failed read
  - Detected data corruption
  - Told to do so by user
- Transparently use available system storage
  - Ram disk
  - Local SSD
  - IBM Burst Buffer
  - Cray Datawarp
- Async flush to parallel file system

Code: https://github.com/llnl/scr

Docs: https://scr.readthedocs.io

Lawrence Livermore National Laboratory