VELOC: Very Low Overhead Checkpointing System

Bogdan Nicolae, Rinku Gupta, Franck Cappello (ANL)

Adam Moody, Elsa Gonsiorowski, Kathryn Mohror (LLNL)

Part 1:
Overview of VELOC

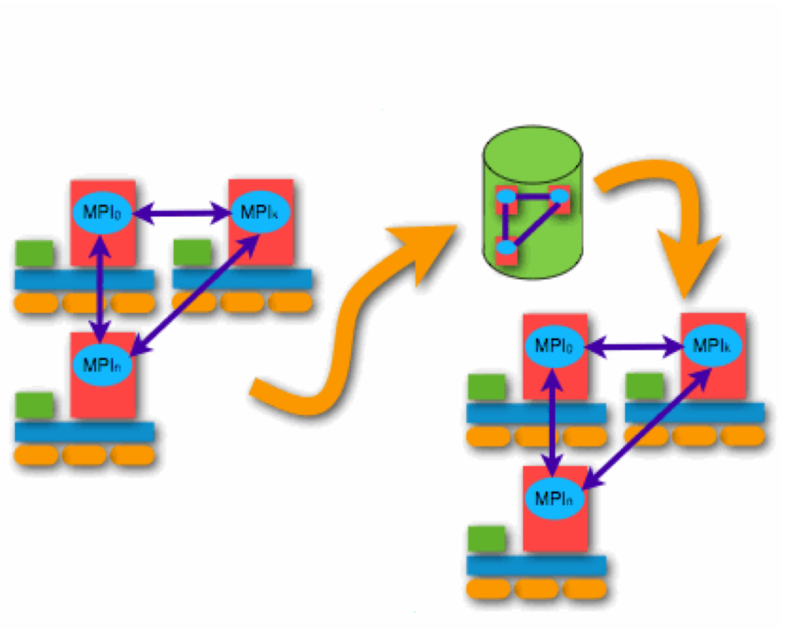
HPC Resilience: Checkpoint-Restart (CR)

- Main resilience technique for HPC due to tight coupling
- “Defensive” checkpointing: save state to parallel file system

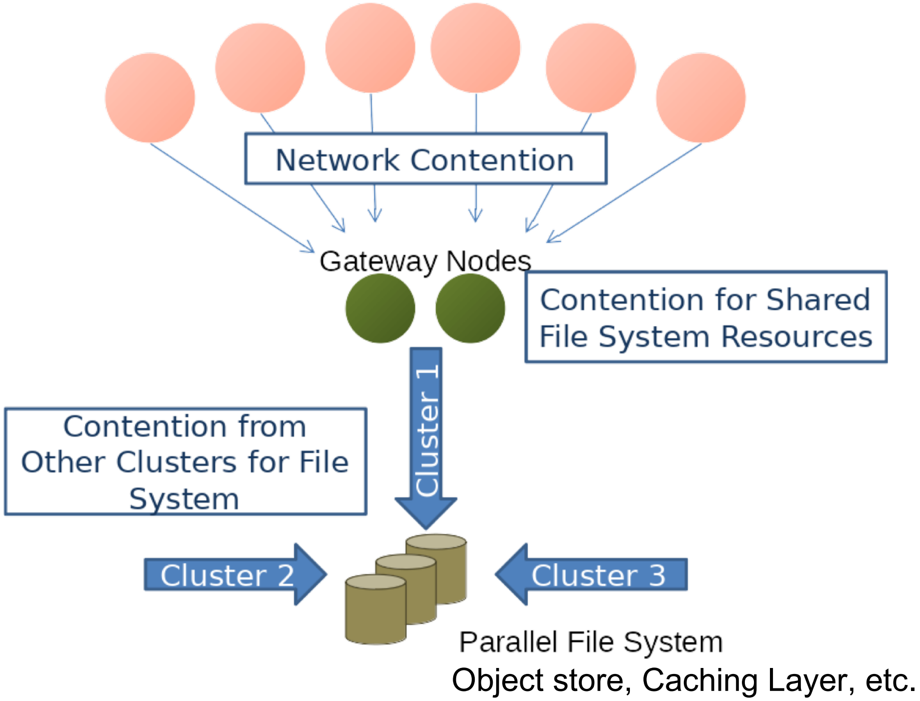
In action **games**, autosave **checkpoints** are points where a **game** will automatically save your progress and restart the player upon death. As such, the player does not need to restart the entire level over again. This reduces the frustration and tedium that is potentially felt without such a design

"**Checkpointing** is one of these things that’s simpler in theory than it is in implementation. The reality is, you’re trying to balance many competing interests,”
Brianna Wu, head of development Giant Spacekat

Bad **checkpoints** ask players to replay large parts of the game due to their death or failure in some task, and this can lead to frustration and anger.

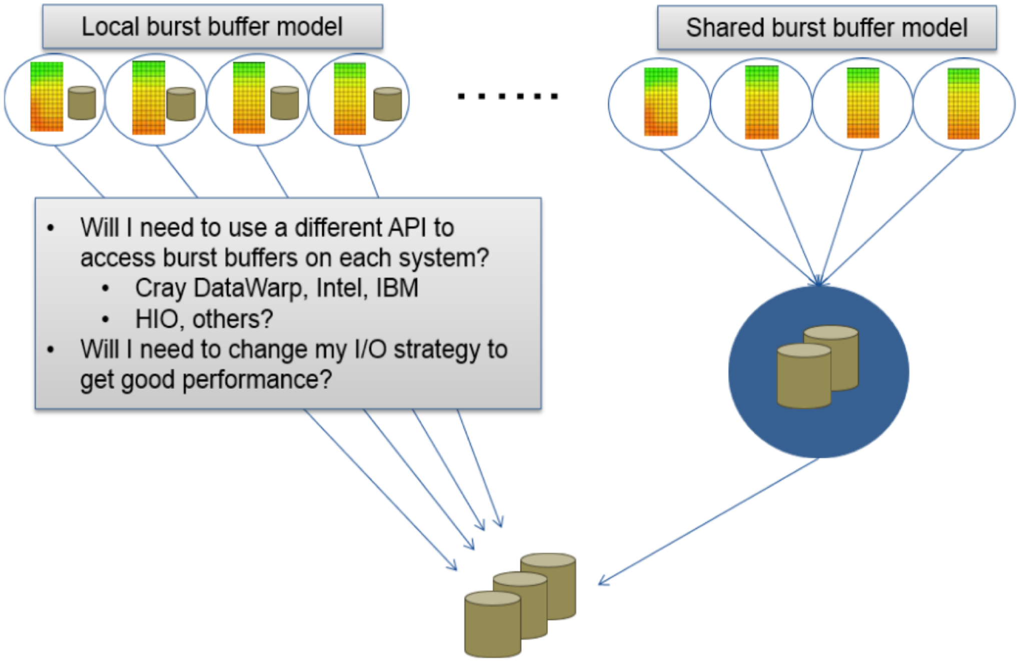


CR at Exascale: Challenges (1)


Network Contention
Gateway Nodes
Contention for Shared File System Resources
Cluster 1
Contention from Other Clusters for File System
Cluster 2
Cluster 3
Parallel File System
Object store, Caching Layer, etc.

- Checkpointing generates a lot of I/O contention to storage
- Impact on performance and scalability is significant
- At Exascale, this issue is amplified:
  - Bigger systems -> more frequent failures -> need to checkpoint more frequently
  - Large increase in CPU power but modest increase in I/O capability -> less I/O bandwidth available per processing element

CR at Exascale: Challenges (2)



- Will I need to use a different API to access burst buffers on each system?
  - Cray DataWarp, Intel, IBM
  - HIO, others?
- Will I need to change my I/O strategy to get good performance?

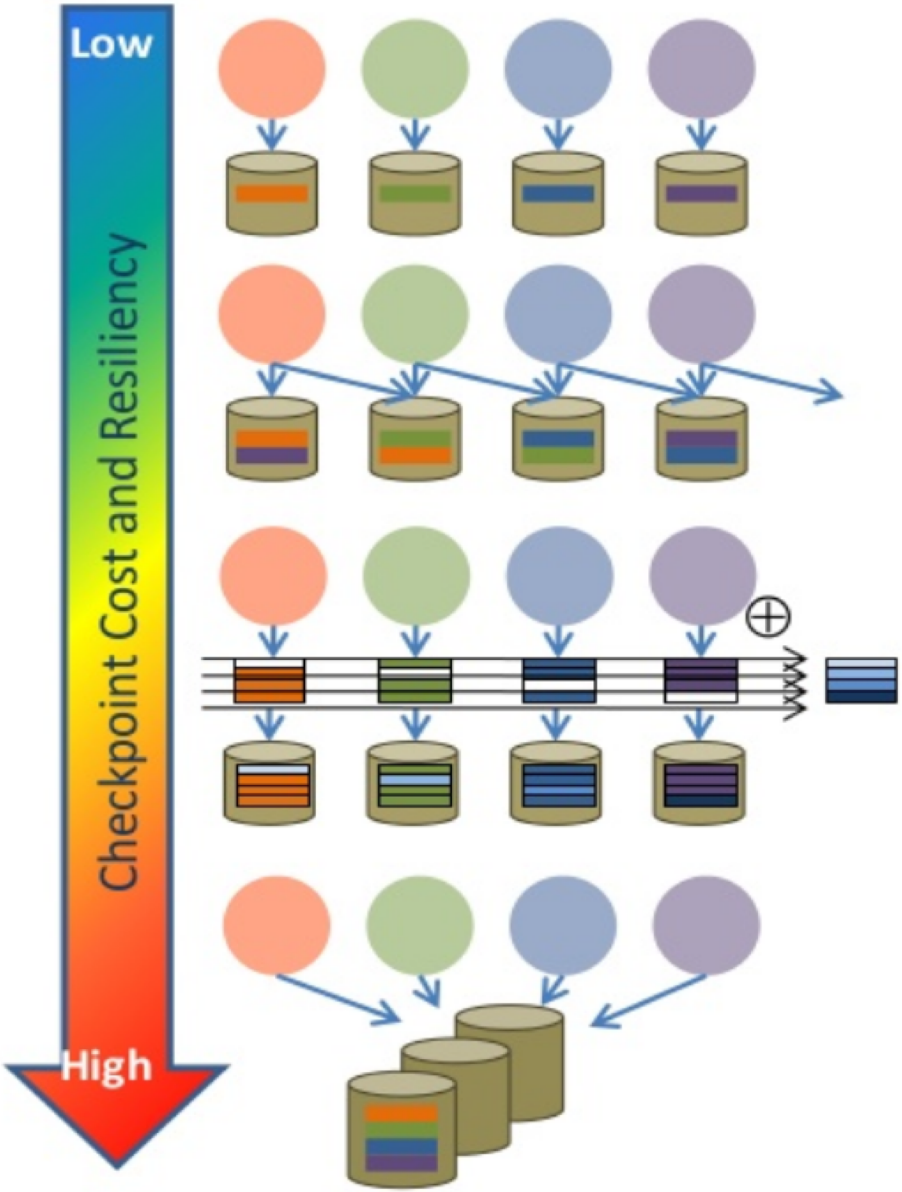
Parallel File System, Object store, Caching Layer, etc.

- Storage hierarchy is heterogeneous and complex at Exascale:
  - Many options in addition to PFS: burst buffers, object stores, caching layers, etc.
  - Each HPC machine has its own combination
  - Many vendors, each with its own API and performance characteristics
- Need to customize CR strategy reduces productivity and leads to inefficiencies as application developers are not I/O experts

VELOC: CR Solution at Exascale

Goal: Provide a checkpoint restart solution for HPC applications that delivers high performance and scalability for complex heterogeneous storage hierarchies without sacrificing ease of use and flexibility

Key idea: Multi-Level CR



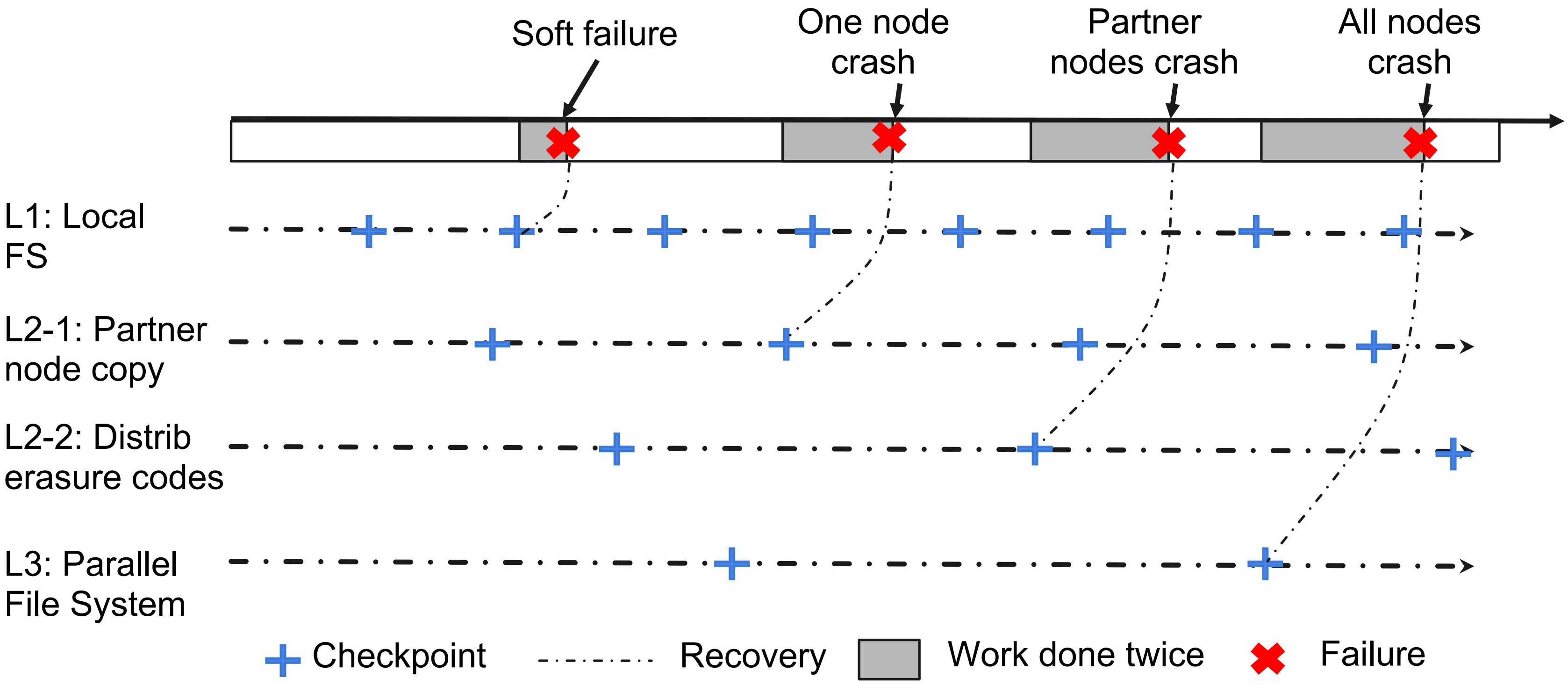
- Multi-level checkpoint-restart uses a layered approach with increasing resilience guarantees but higher checkpointing overhead:
  - L1: local checkpoints
  - L2: partner copies, erasure codes
  - L3: parallel file system
- Higher levels defend against more complex types of failures, which typically happen less frequently
- Cost of higher levels can be masked asynchronously

VELOC improves performance and scalability by using multi-level CR

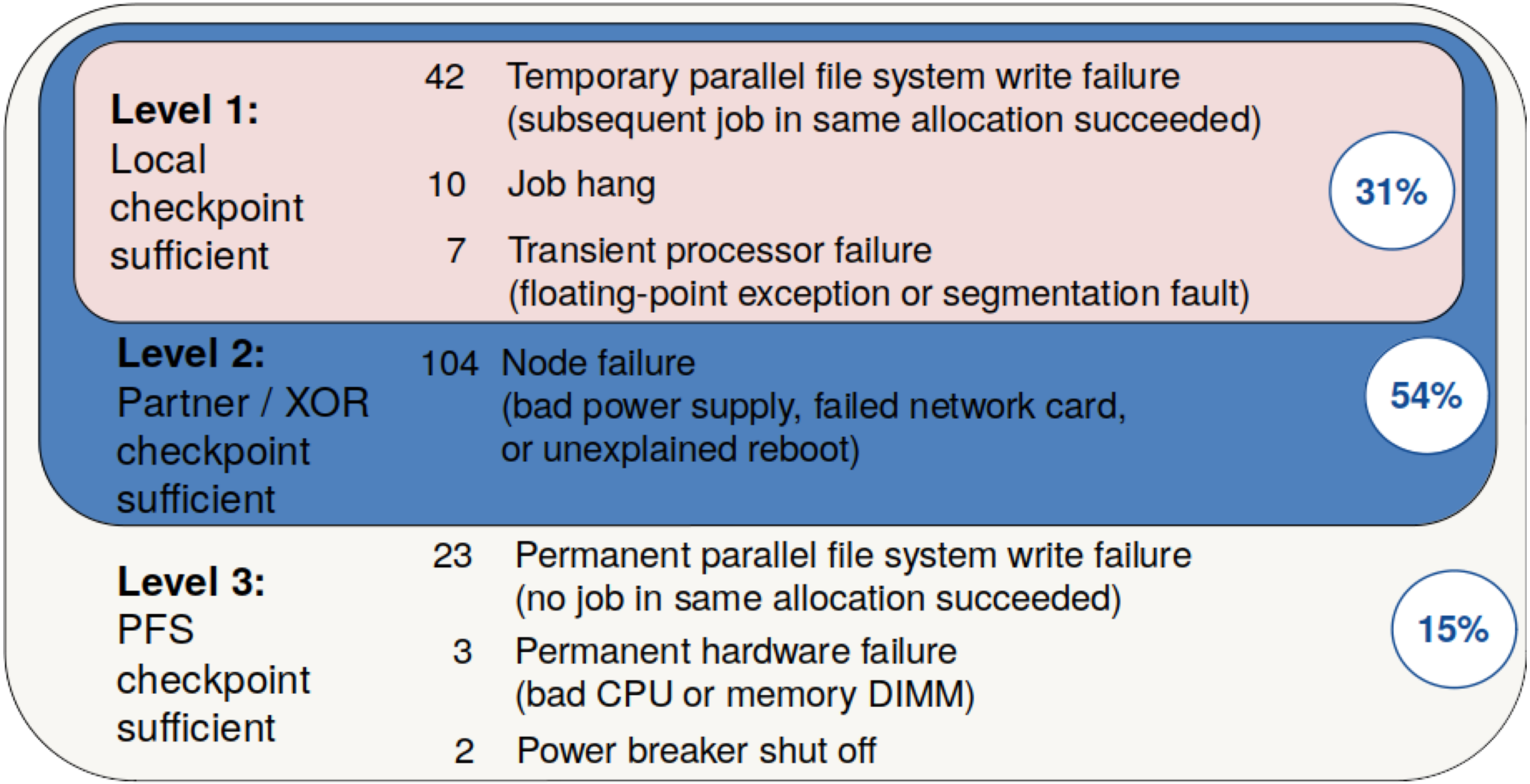
How to use multiple levels

The checkpoint interval of each level is optimized for the type of failures not covered by the previous levels

- L1 survives software errors
- L2 survives a majority of simultaneous node failures
- L3 survives catastrophic failures (rack or system down)



Example of observed failures by level



Level 1: Local checkpoint sufficient

42 Temporary parallel file system write failure (subsequent job in same allocation succeeded)

10 Job hang

7 Transient processor failure (floating-point exception or segmentation fault)

31%

Level 2: Partner / XOR checkpoint sufficient

104 Node failure (bad power supply, failed network card, or unexplained reboot)

54%

Level 3: PFS checkpoint sufficient

23 Permanent parallel file system write failure (no job in same allocation succeeded)

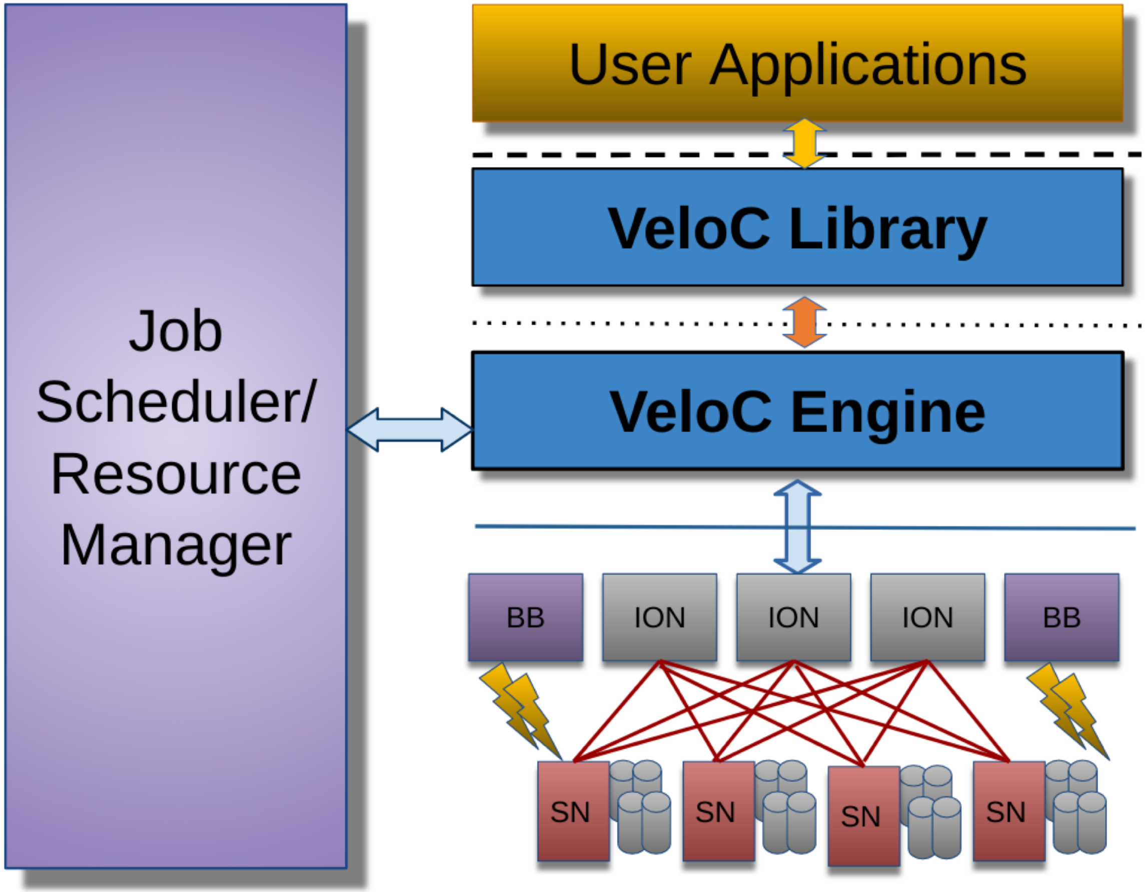
3 Permanent hardware failure (bad CPU or memory DIMM)

2 Power breaker shut off

15%

Observed 191 failures spanning 5.6 million node hours from 871 runs of PF3d on 3 different clusters (Coastal, Hera, and Atlas).

Hidden Complexity of Heterogeneous Storage



One simple VeloC API

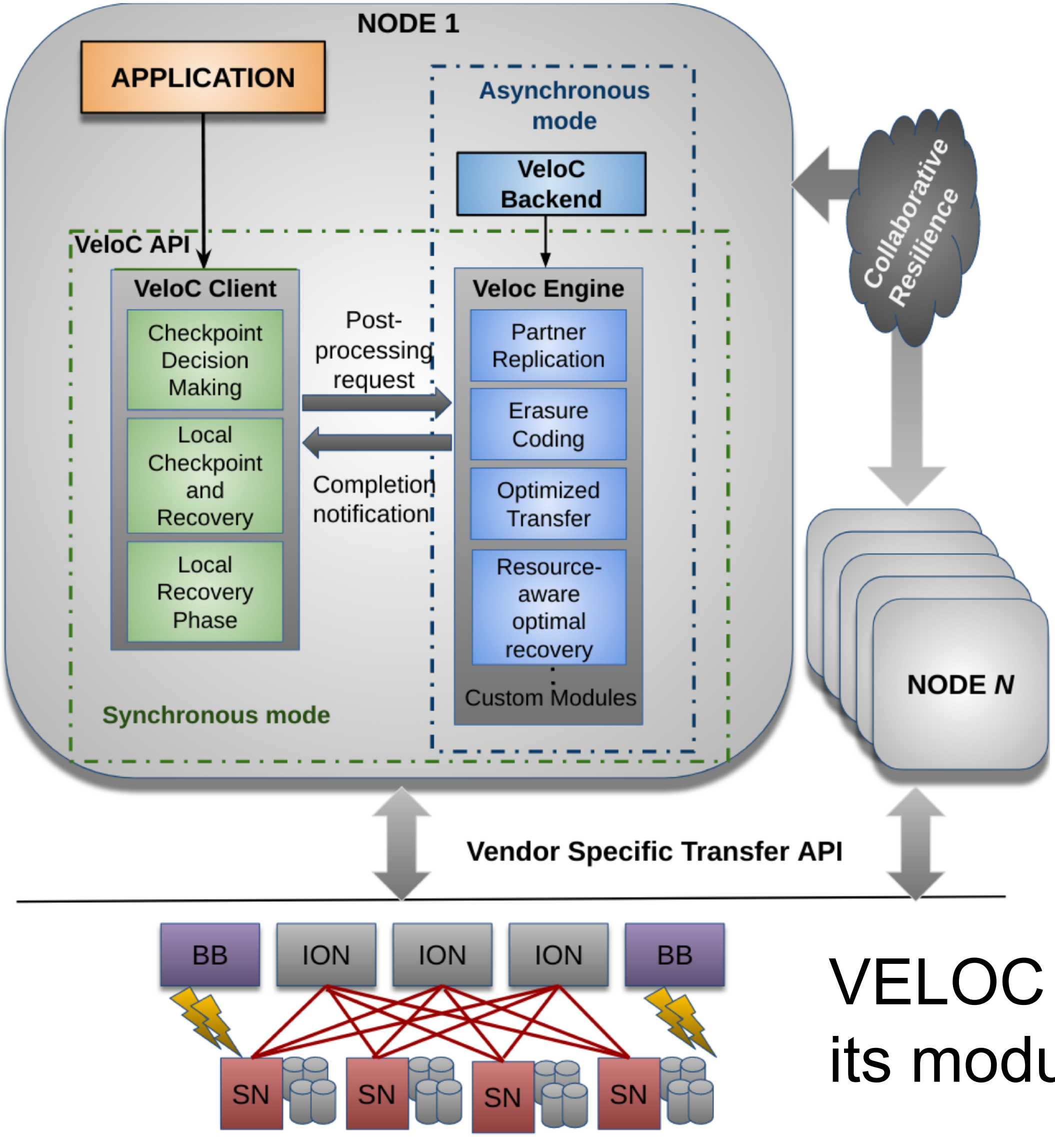
Many complex vendor APIs:

- Cray DataWarp
- DDN IME
- EMC 2 Tiers
- IBM CORAL burst buffer

Complex Heterogeneous Storage Hierarchy (Burst Buffers, Parallel File Systems, Object Stores, etc.)

VELOC facilitates ease of use by transparent interaction with the heterogeneous storage hierarchy

Modular Architecture

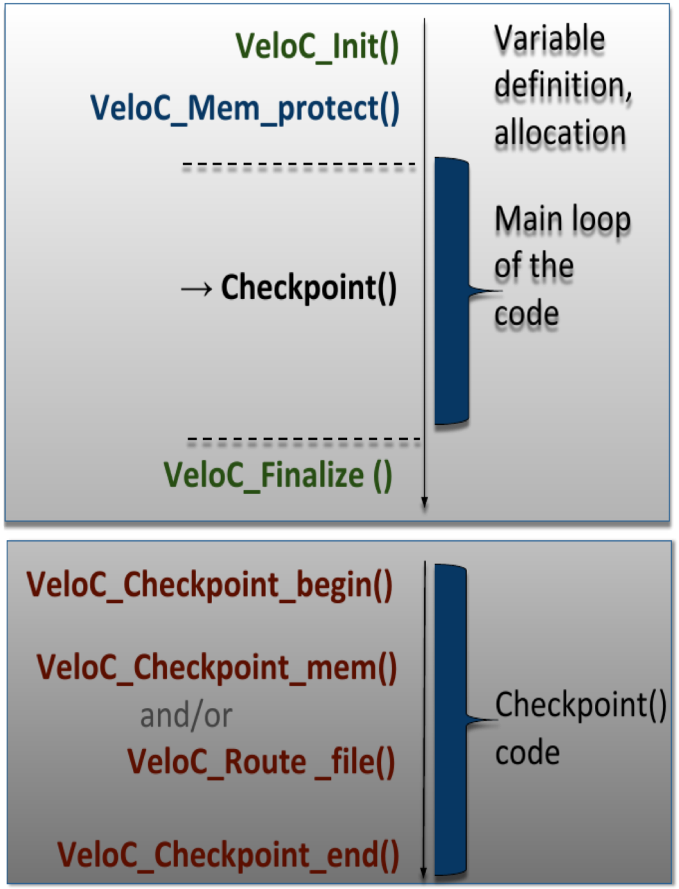


- Configurable resilience strategy:
  - L1: Local write
  - L2: Partner replication, XOR encoding, RS encoding
  - L3: Optimized transfer to external storage
- Configurable mode of operation:
  - Synchronous mode: resilience engine runs in application process
  - Asynchronous mode: resilience engine in separate backend process (backend survives software failures in apps)
- Easily extensible:
  - Custom modules can be added for additional post-processing in the engine (e.g. compression)

VELOC facilitates flexibility thanks to its modular design

VELOC API

- Application-level checkpoint and restart API
- Minimizes code changes in applications
- Two possible modes:
  - File-oriented API: Manually write files and tell VeloC about them
  - Memory-oriented API: Declare and capture memory regions automatically
- Fire-and-forget: VeloC operates in the background
- Waiting for checkpoints is optional; a primitive is used to check progress



Initializing VELOC:

- `VELOC_Init()`
- `VELOC_Finalize()`

Memory registration:

- `VELOC_Mem_protect()`
- `VELOC_Mem_unprotect()`

File registration:

- `VELOC_Route_file()`

Checkpoint functions:

- `VELOC_Checkpoint_wait()`
- `VELOC_Checkpoint_begin()`
- `VELOC_Checkpoint_mem()`
- `VELOC_Checkpoint_end()`

Restart functions:

- `VELOC_Restart_test()`
- `VELOC_Restart_begin()`
- `VELOC_Recover_mem()`
- `VELOC_Restart_end()`

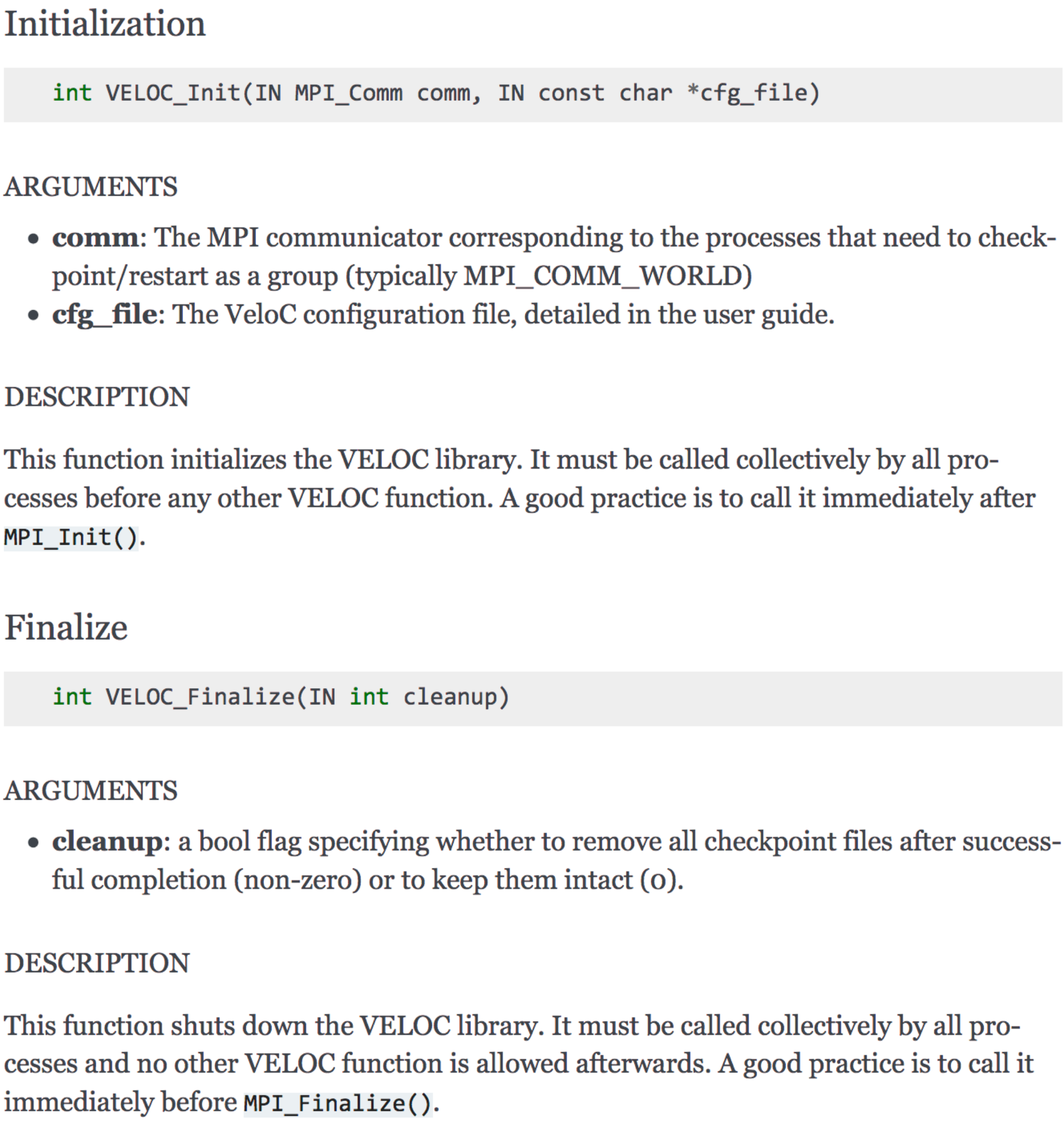
Environmental functions:

- `VELOC_Get_version()`

Convenience functions (Mem. only):

- `VELOC_Checkpoint()`
- `VELOC_Restart()`

VeloC Initialization and Finalize



Initialization

```
int VELOC_Init(IN MPI_Comm comm, IN const char *cfg_file)
```

ARGUMENTS

- **comm**: The MPI communicator corresponding to the processes that need to checkpoint/restart as a group (typically MPI_COMM_WORLD)
- **cfg_file**: The VeloC configuration file, detailed in the user guide.

DESCRIPTION

This function initializes the VELOC library. It must be called collectively by all processes before any other VELOC function. A good practice is to call it immediately after `MPI_Init()`.

Finalize

```
int VELOC_Finalize(IN int cleanup)
```

ARGUMENTS

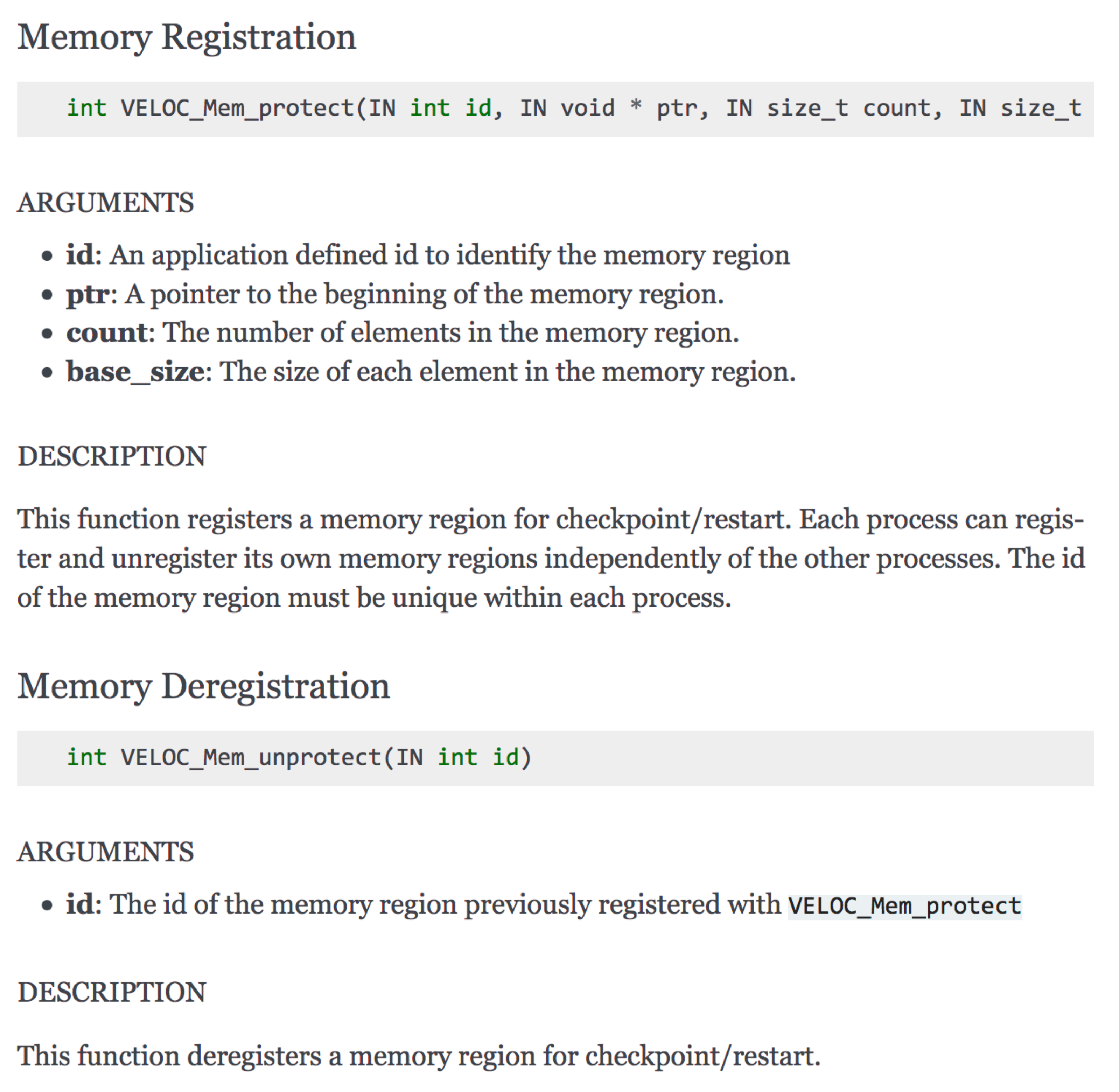
- **cleanup**: a bool flag specifying whether to remove all checkpoint files after successful completion (non-zero) or to keep them intact (0).

DESCRIPTION

This function shuts down the VELOC library. It must be called collectively by all processes and no other VELOC function is allowed afterwards. A good practice is to call it immediately before `MPI_Finalize()`.

VELOC Memory-Based Mode

In memory-based mode, applications need to register any critical memory regions needed for restart.
Registration is allowed at any moment before initiating a checkpoint or restart.
Memory regions can also be unregistered if they become non-critical at any moment during runtime.



Memory Registration

```
int VELOC_Mem_protect(IN int id, IN void * ptr, IN size_t count, IN size_t
```

ARGUMENTS

- **id**: An application defined id to identify the memory region
- **ptr**: A pointer to the beginning of the memory region.
- **count**: The number of elements in the memory region.
- **base_size**: The size of each element in the memory region.

DESCRIPTION

This function registers a memory region for checkpoint/restart. Each process can register and unregister its own memory regions independently of the other processes. The id of the memory region must be unique within each process.

Memory Deregistration

```
int VELOC_Mem_unprotect(IN int id)
```

ARGUMENTS

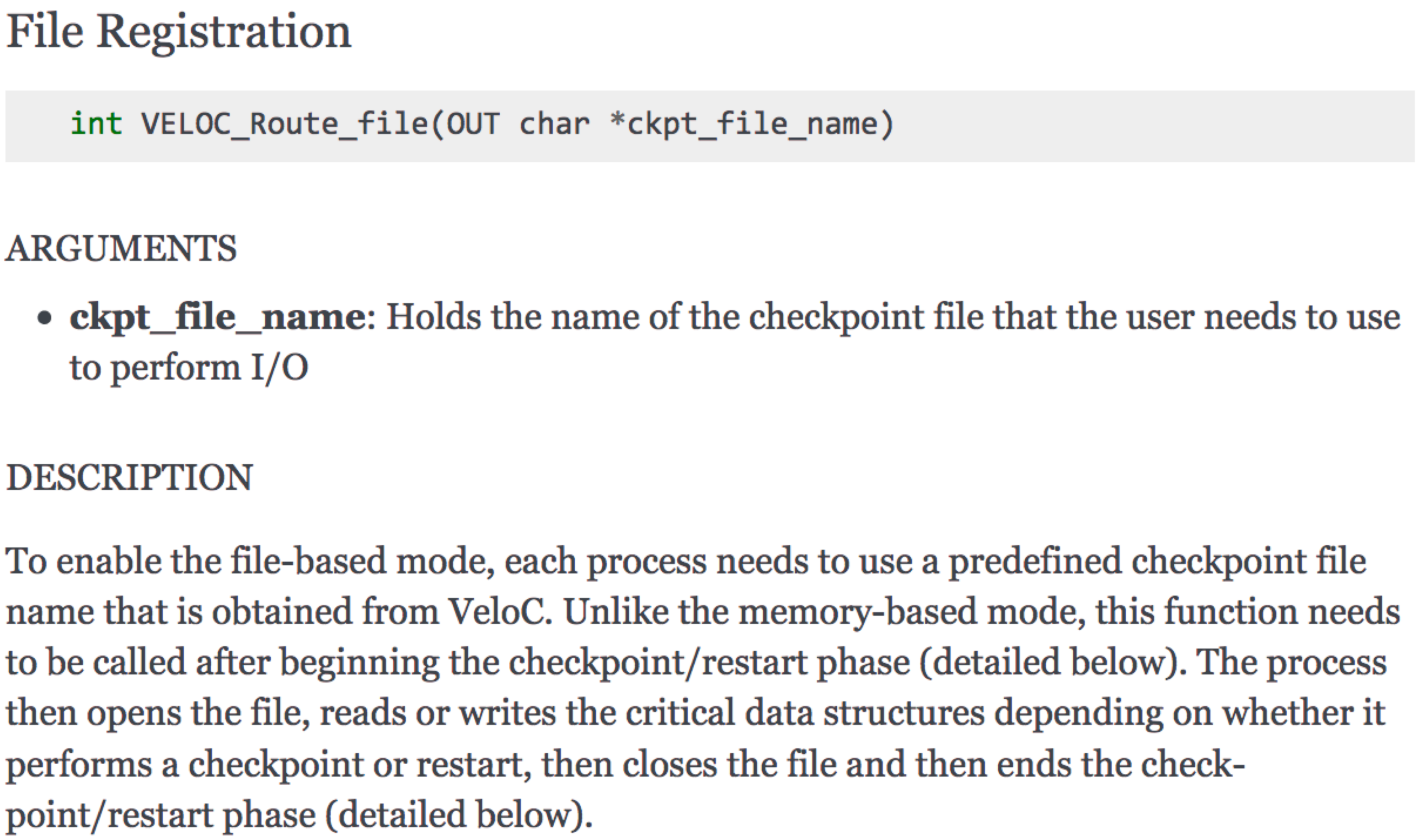
- **id**: The id of the memory region previously registered with `VELOC_Mem_protect`

DESCRIPTION

This function deregisters a memory region for checkpoint/restart.

VELOC File-Based Mode

In the file-based mode, applications need to manually serialize/recover the critical data structures to/from checkpoint files. This mode provides fine-grain control over the serialization process and is especially useful when the application uses non-contiguous memory regions for which the memory-based API is not convenient to use.



File Registration

```
int VELOC_Route_file(OUT char *ckpt_file_name)
```

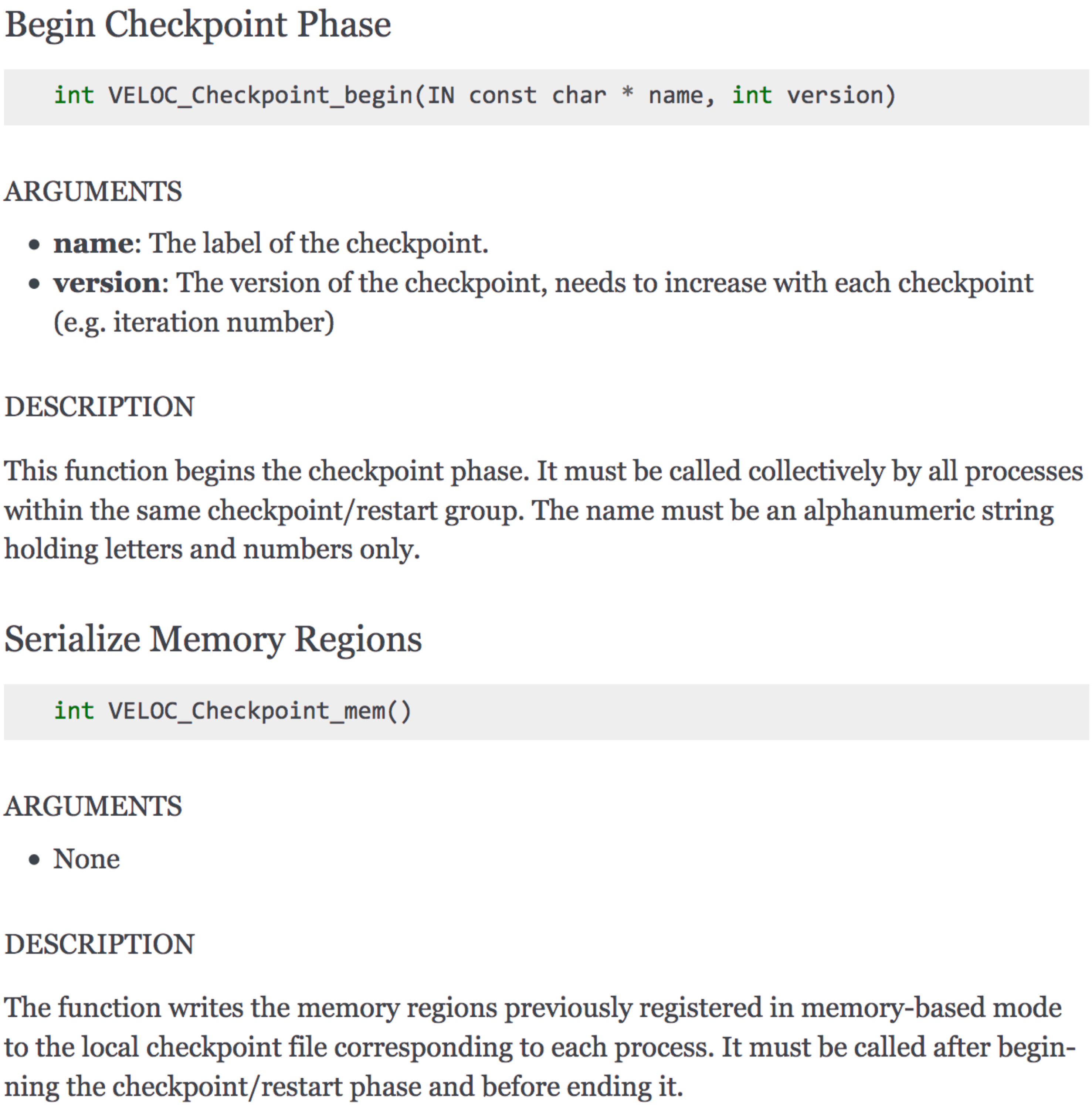
ARGUMENTS

- **ckpt_file_name**: Holds the name of the checkpoint file that the user needs to use to perform I/O

DESCRIPTION

To enable the file-based mode, each process needs to use a predefined checkpoint file name that is obtained from VeloC. Unlike the memory-based mode, this function needs to be called after beginning the checkpoint/restart phase (detailed below). The process then opens the file, reads or writes the critical data structures depending on whether it performs a checkpoint or restart, then closes the file and then ends the checkpoint/restart phase (detailed below).

VELOC Checkpoint Functions



Begin Checkpoint Phase

```
int VELOC_Checkpoint_begin(IN const char * name, int version)
```

ARGUMENTS

- **name**: The label of the checkpoint.
- **version**: The version of the checkpoint, needs to increase with each checkpoint (e.g. iteration number)

DESCRIPTION

This function begins the checkpoint phase. It must be called collectively by all processes within the same checkpoint/restart group. The name must be an alphanumeric string holding letters and numbers only.

Serialize Memory Regions

```
int VELOC_Checkpoint_mem()
```

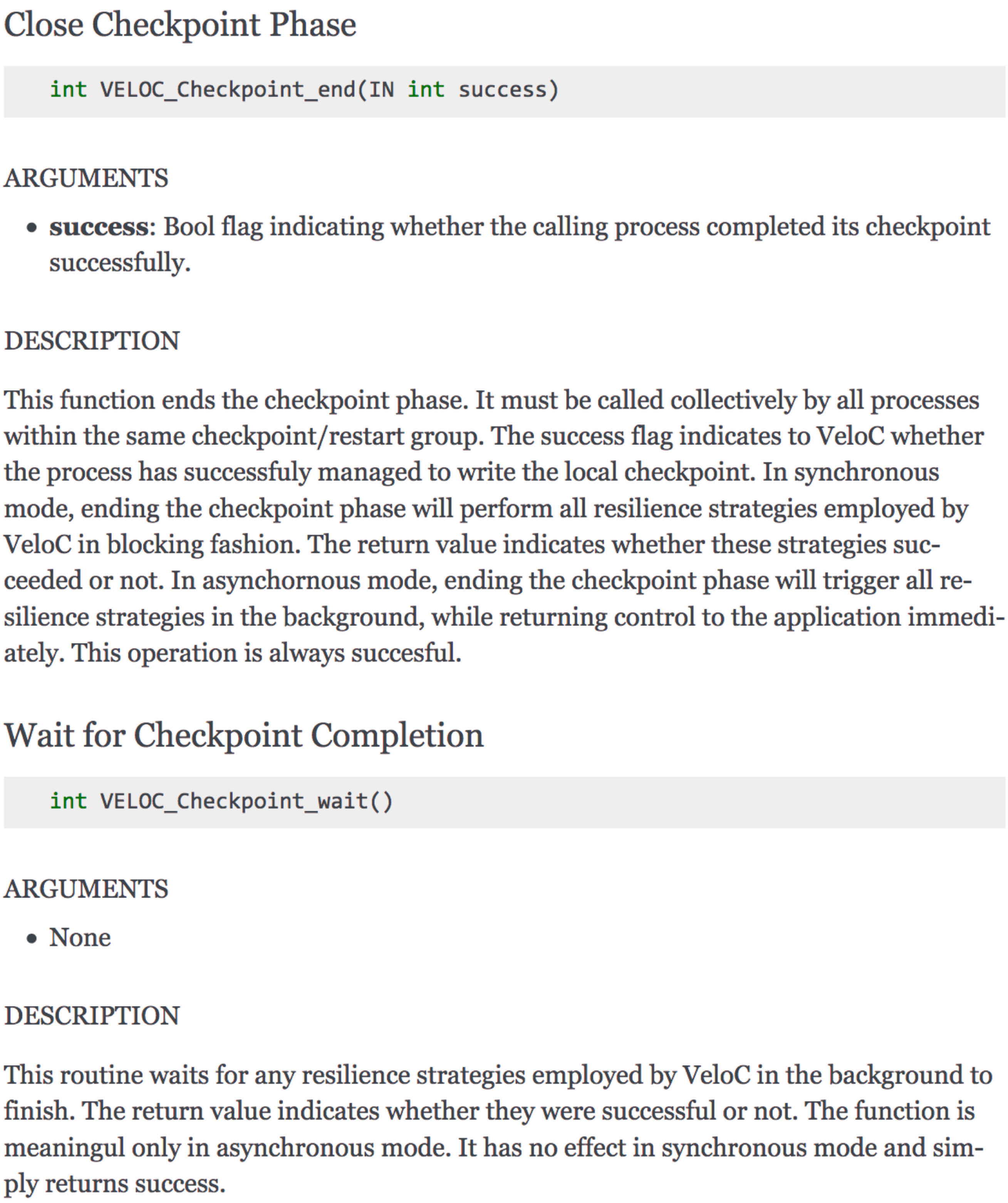
ARGUMENTS

- None

DESCRIPTION

The function writes the memory regions previously registered in memory-based mode to the local checkpoint file corresponding to each process. It must be called after beginning the checkpoint/restart phase and before ending it.

VELOC Checkpointing Functions (cont.)



Close Checkpoint Phase

```
int VELOC_Checkpoint_end(IN int success)
```

ARGUMENTS

- **success**: Bool flag indicating whether the calling process completed its checkpoint successfully.

DESCRIPTION

This function ends the checkpoint phase. It must be called collectively by all processes within the same checkpoint/restart group. The success flag indicates to VeloC whether the process has successfuly managed to write the local checkpoint. In synchronous mode, ending the checkpoint phase will perform all resilience strategies employed by VeloC in blocking fashion. The return value indicates whether these strategies succeeded or not. In asynchornous mode, ending the checkpoint phase will trigger all resilience strategies in the background, while returning control to the application immediately. This operation is always succesful.

Wait for Checkpoint Completion

```
int VELOC_Checkpoint_wait()
```

ARGUMENTS

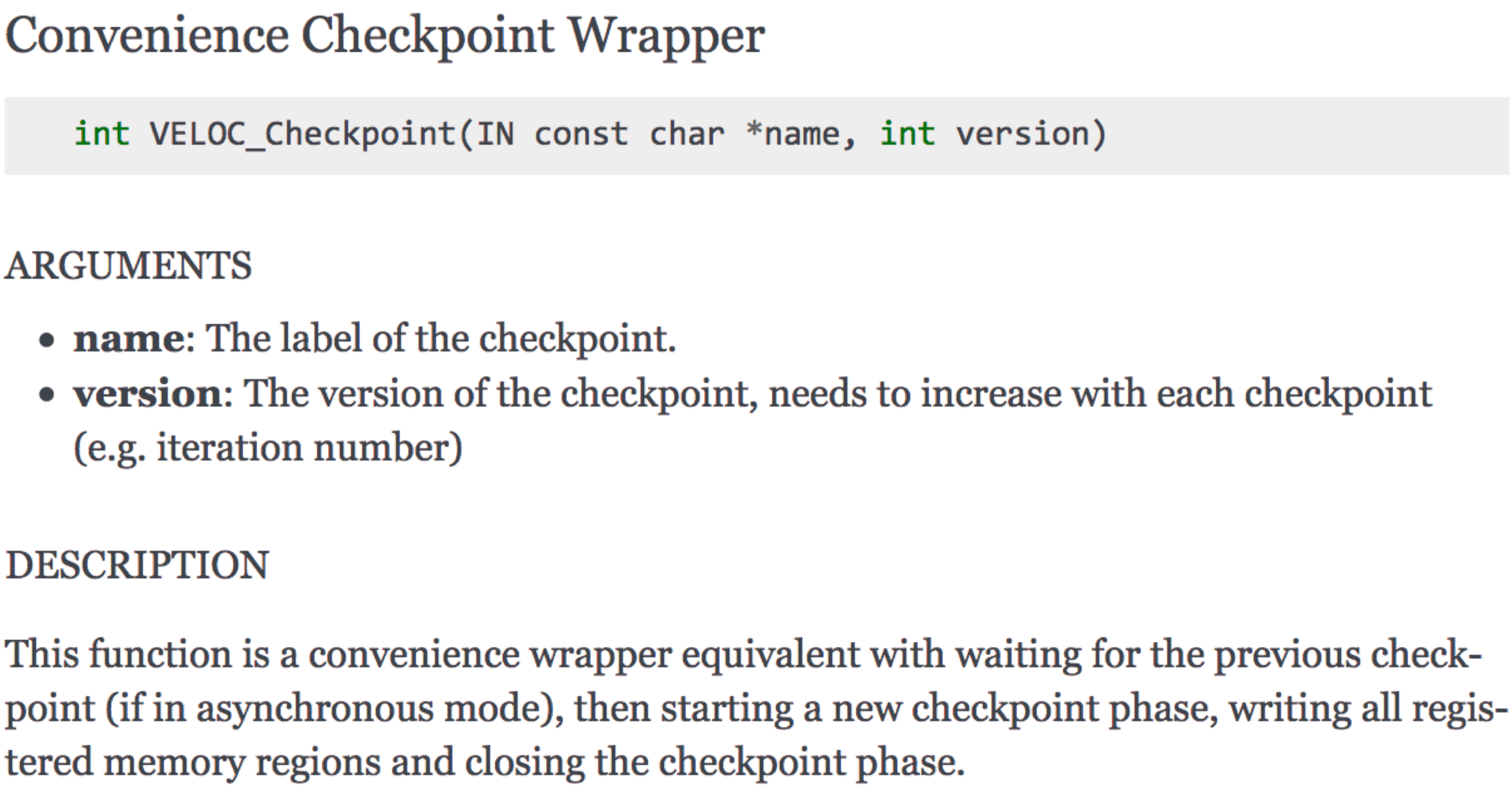
- None

DESCRIPTION

This routine waits for any resilience strategies employed by VeloC in the background to finish. The return value indicates whether they were successful or not. The function is meaningul only in asynchronous mode. It has no effect in synchronous mode and simply returns success.

Needed in the file mode:
VeloC needs to know when
writing on the checkpoint file
Is done to start the next steps
(synchronous or asynchronous)
of multi-level checkpointing.

VELOC Checkpointing Functions (cont.)



Convenience Checkpoint Wrapper

```
int VELOC_Checkpoint(IN const char *name, int version)
```

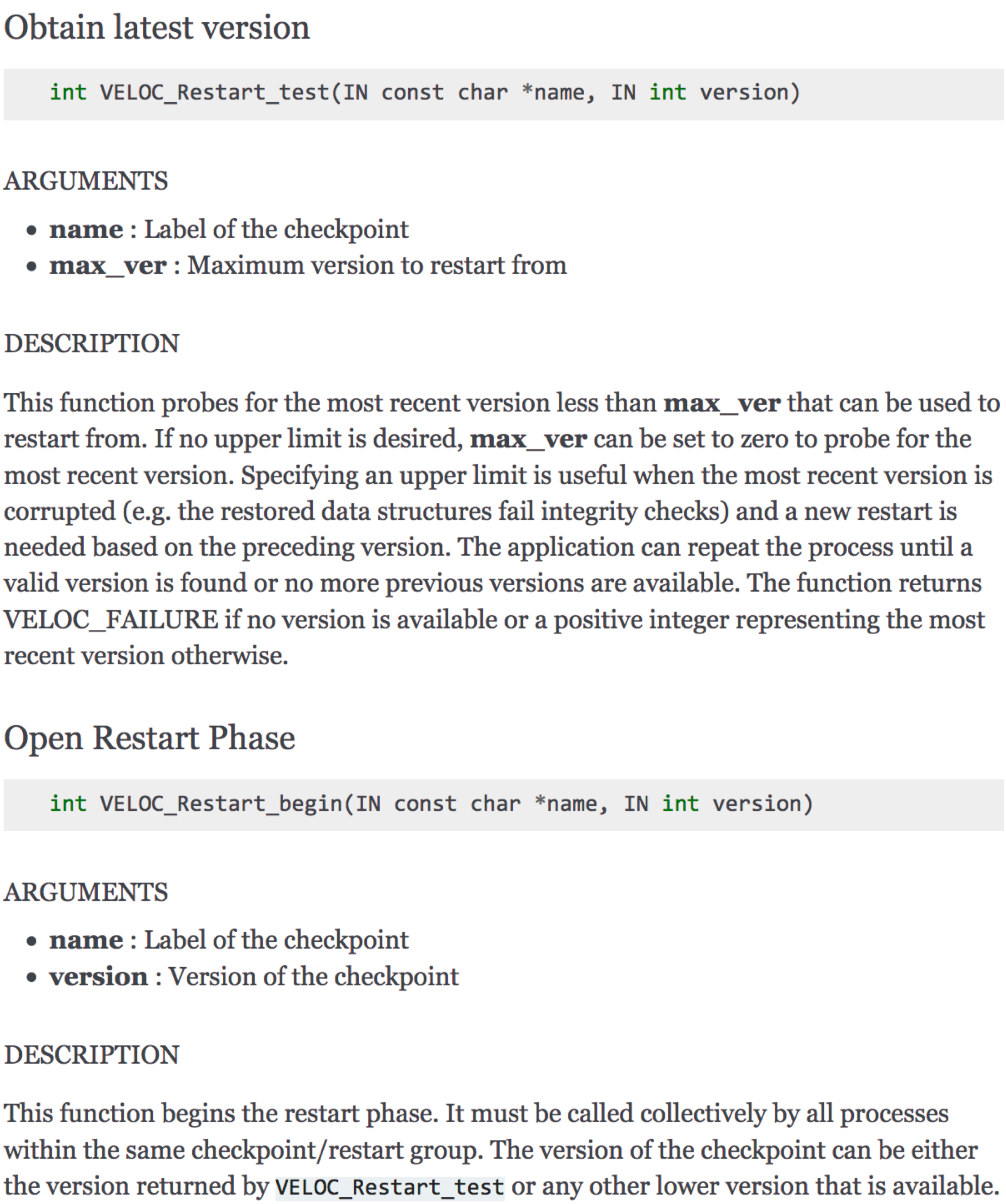
ARGUMENTS

- **name**: The label of the checkpoint.
- **version**: The version of the checkpoint, needs to increase with each checkpoint (e.g. iteration number)

DESCRIPTION

This function is a convenience wrapper equivalent with waiting for the previous checkpoint (if in asynchronous mode), then starting a new checkpoint phase, writing all registered memory regions and closing the checkpoint phase.

VELOC Restart Functions



Obtain latest version

```
int VELOC_Restart_test(IN const char *name, IN int version)
```

ARGUMENTS

- **name** : Label of the checkpoint
- **max_ver** : Maximum version to restart from

DESCRIPTION

This function probes for the most recent version less than **max_ver** that can be used to restart from. If no upper limit is desired, **max_ver** can be set to zero to probe for the most recent version. Specifying an upper limit is useful when the most recent version is corrupted (e.g. the restored data structures fail integrity checks) and a new restart is needed based on the preceding version. The application can repeat the process until a valid version is found or no more previous versions are available. The function returns VELOC_FAILURE if no version is available or a positive integer representing the most recent version otherwise.

Open Restart Phase

```
int VELOC_Restart_begin(IN const char *name, IN int version)
```

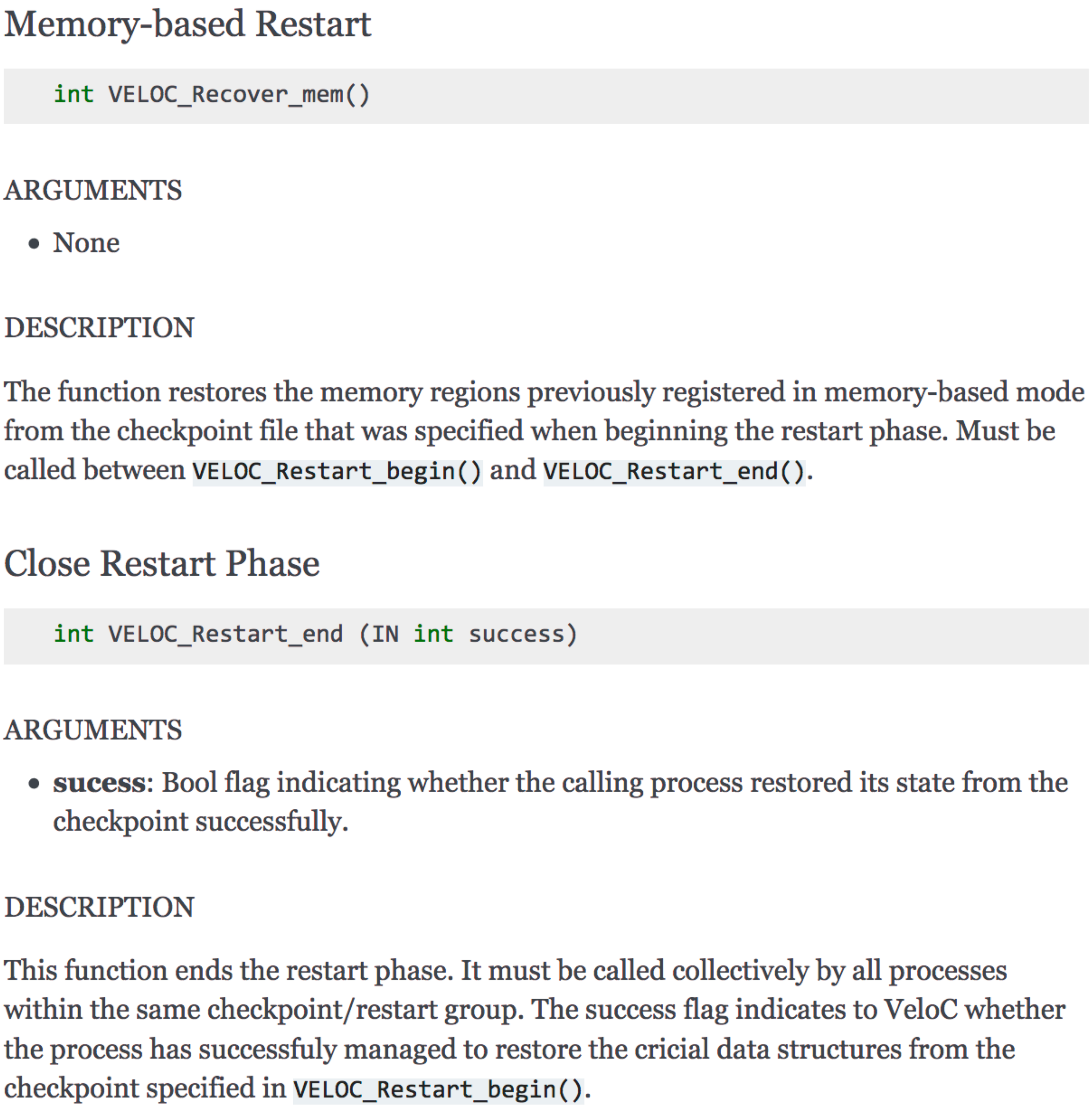
ARGUMENTS

- **name** : Label of the checkpoint
- **version** : Version of the checkpoint

DESCRIPTION

This function begins the restart phase. It must be called collectively by all processes within the same checkpoint/restart group. The version of the checkpoint can be either the version returned by `VELOC_Restart_test` or any other lower version that is available.

VELOC Restart Functions (cont.)



Memory-based Restart

```
int VELOC_Recover_mem()
```

ARGUMENTS

- None

DESCRIPTION

The function restores the memory regions previously registered in memory-based mode from the checkpoint file that was specified when beginning the restart phase. Must be called between `VELOC_Restart_begin()` and `VELOC_Restart_end()`.

Close Restart Phase

```
int VELOC_Restart_end (IN int success)
```

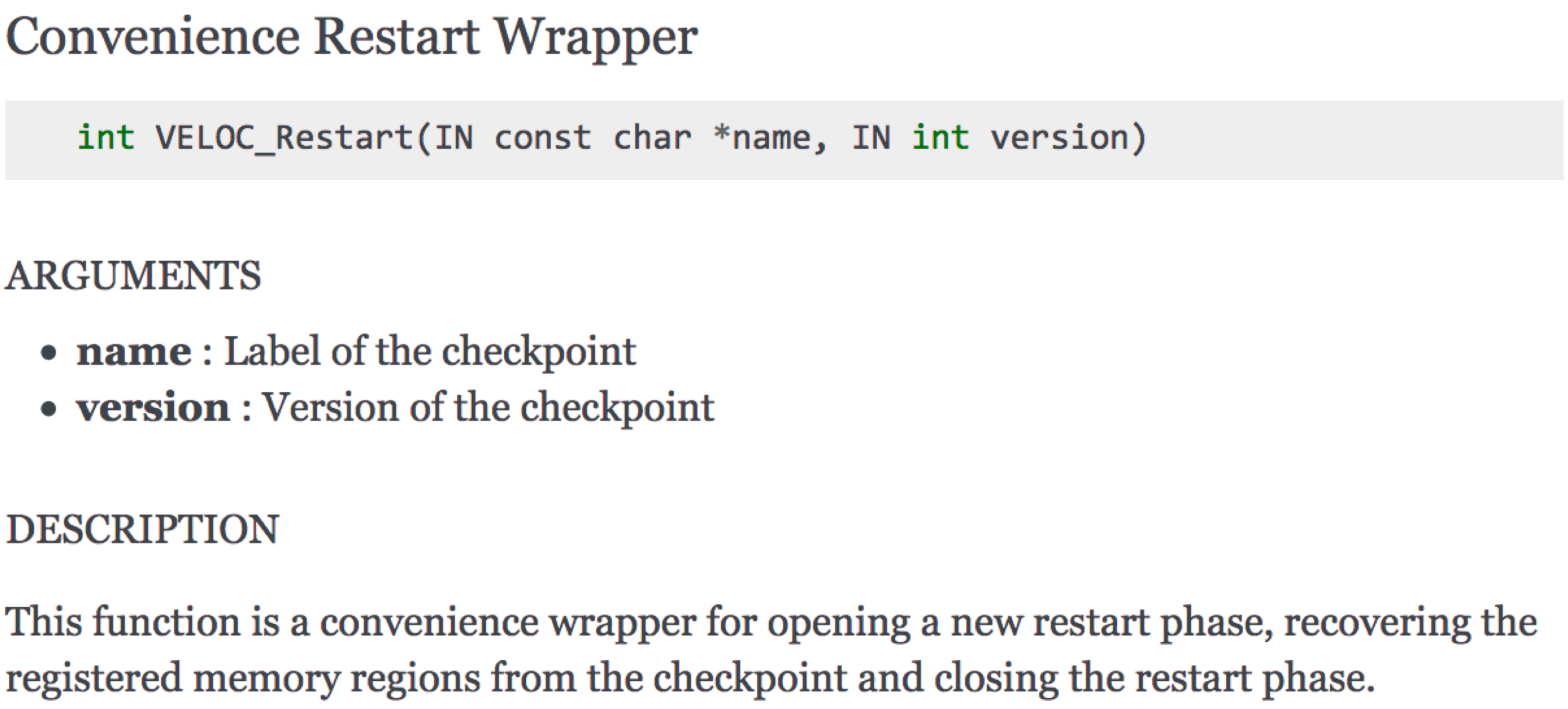
ARGUMENTS

- **sucess**: Bool flag indicating whether the calling process restored its state from the checkpoint successfully.

DESCRIPTION

This function ends the restart phase. It must be called collectively by all processes within the same checkpoint/restart group. The success flag indicates to VeloC whether the process has successfuly managed to restore the cricial data structures from the checkpoint specified in `VELOC_Restart_begin()`.

VELOC Restart Functions (cont.)



Convenience Restart Wrapper

```
int VELOC_Restart(IN const char *name, IN int version)
```

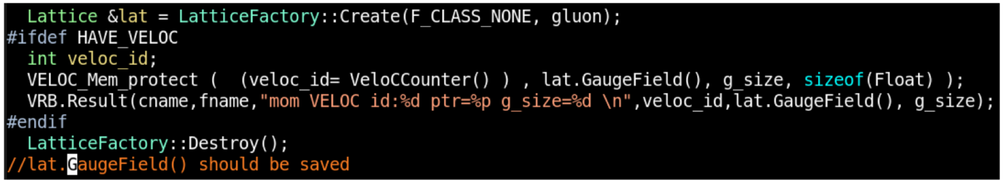
ARGUMENTS

- **name** : Label of the checkpoint
- **version** : Version of the checkpoint

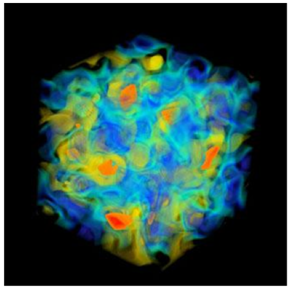
DESCRIPTION

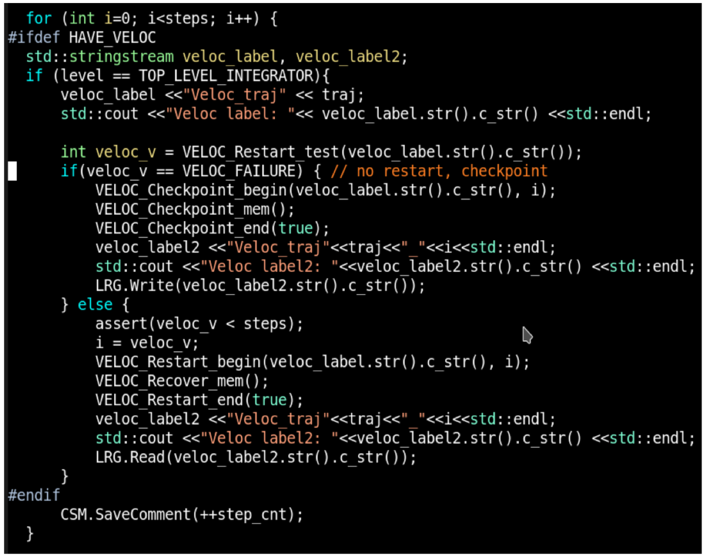
This function is a convenience wrapper for opening a new restart phase, recovering the registered memory regions from the checkpoint and closing the restart phase.

Examples of ECP apps using VELOC

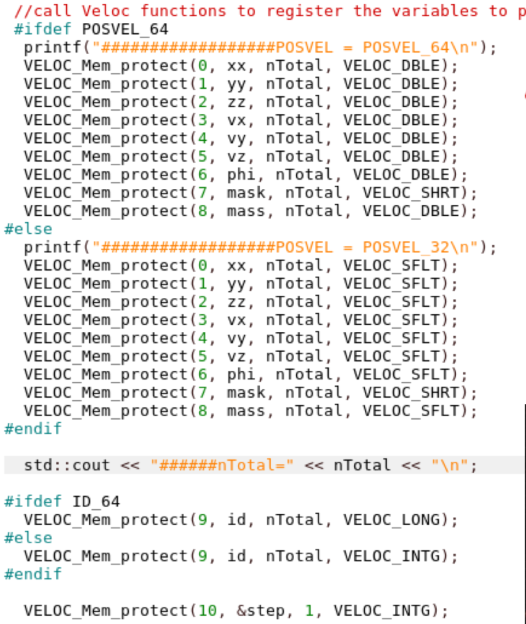


```
  Lattice &lat = LatticeFactory::Create(F_CLASS_NONE, gluon);
#ifdef HAVE_VELOC
  int veloc_id;
  VELOC_Mem_protect (  (veloc_id= VeloCCounter() ) , lat.GaugeField(), g_size, sizeof(Float) );
  VRB.Result(cname,fname,"mom VELOC id:%d ptr=%p g_size=%d \n",veloc_id,lat.GaugeField(), g_size);
#endif
  LatticeFactory::Destroy();
//lat.GaugeField() should be saved
```

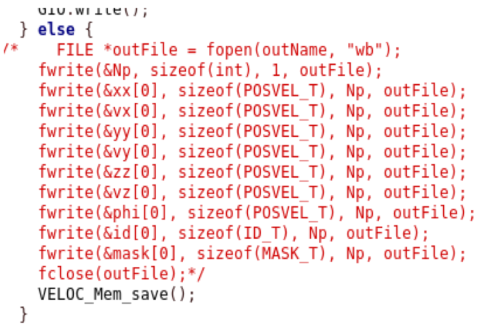




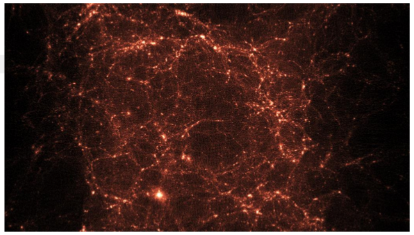
```
  for (int i=0; i<steps; i++) {
#ifdef HAVE_VELOC
  std::stringstream veloc_label, veloc_label2;
  if (level == TOP_LEVEL_INTEGRATOR){
      veloc_label <<"Veloc_traj" << traj;
      std::cout <<"Veloc label: "<< veloc_label.str().c_str() <<std::endl;

      int veloc_v = VELOC_Restart_test(veloc_label.str().c_str());
      if(veloc_v == VELOC_FAILURE) { // no restart, checkpoint
         VELOC_Checkpoint_begin(veloc_label.str().c_str(), i);
         VELOC_Checkpoint_mem();
         VELOC_Checkpoint_end(true);
         veloc_label2 <<"Veloc_traj"<<traj<<"_"<<i<<std::endl;
         std::cout <<"Veloc label2: "<<veloc_label2.str().c_str() <<std::endl;
         LRG.Write(veloc_label2.str().c_str());
      } else {
         assert(veloc_v < steps);
         i = veloc_v;
         VELOC_Restart_begin(veloc_label.str().c_str(), i);
         VELOC_Recover_mem();
         VELOC_Restart_end(true);
         veloc_label2 <<"Veloc_traj"<<traj<<"_"<<i<<std::endl;
         std::cout <<"Veloc label2: "<<veloc_label2.str().c_str() <<std::endl;
         LRG.Read(veloc_label2.str().c_str());
      }
#endif
      CSM.SaveComment(++step_cnt);
  }
```



```
 //call Veloc functions to register the variables to protect
 #ifdef POSVEL_64
  printf("##################POSVEL = POSVEL_64\n");
  VELOC_Mem_protect(0, xx, nTotal, VELOC_DBLE);
  VELOC_Mem_protect(1, yy, nTotal, VELOC_DBLE);
  VELOC_Mem_protect(2, zz, nTotal, VELOC_DBLE);
  VELOC_Mem_protect(3, vx, nTotal, VELOC_DBLE);
  VELOC_Mem_protect(4, vy, nTotal, VELOC_DBLE);
  VELOC_Mem_protect(5, vz, nTotal, VELOC_DBLE);
  VELOC_Mem_protect(6, phi, nTotal, VELOC_DBLE);
  VELOC_Mem_protect(7, mask, nTotal, VELOC_SHRT);
  VELOC_Mem_protect(8, mass, nTotal, VELOC_DBLE);
#else
  printf("##################POSVEL = POSVEL_32\n");
  VELOC_Mem_protect(0, xx, nTotal, VELOC_SFLT);
  VELOC_Mem_protect(1, yy, nTotal, VELOC_SFLT);
  VELOC_Mem_protect(2, zz, nTotal, VELOC_SFLT);
  VELOC_Mem_protect(3, vx, nTotal, VELOC_SFLT);
  VELOC_Mem_protect(4, vy, nTotal, VELOC_SFLT);
  VELOC_Mem_protect(5, vz, nTotal, VELOC_SFLT);
  VELOC_Mem_protect(6, phi, nTotal, VELOC_SFLT);
  VELOC_Mem_protect(7, mask, nTotal, VELOC_SHRT);
  VELOC_Mem_protect(8, mass, nTotal, VELOC_SFLT);
#endif

  std::cout << "######nTotal=" << nTotal << "\n";

#ifdef ID_64
  VELOC_Mem_protect(9, id, nTotal, VELOC_LONG);
#else
  VELOC_Mem_protect(9, id, nTotal, VELOC_INTG);
#endif

  VELOC_Mem_protect(10, &step, 1, VELOC_INTG);
```



```
        GIO.write();
      } else {
  /*      FILE *outFile = fopen(outName, "wb");
        fwrite(&Np, sizeof(int), 1, outFile);
        fwrite(&xx[0], sizeof(POSVEL_T), Np, outFile);
        fwrite(&vx[0], sizeof(POSVEL_T), Np, outFile);
        fwrite(&yy[0], sizeof(POSVEL_T), Np, outFile);
        fwrite(&vy[0], sizeof(POSVEL_T), Np, outFile);
        fwrite(&zz[0], sizeof(POSVEL_T), Np, outFile);
        fwrite(&vz[0], sizeof(POSVEL_T), Np, outFile);
        fwrite(&phi[0], sizeof(POSVEL_T), Np, outFile);
        fwrite(&id[0], sizeof(ID_T), Np, outFile);
        fwrite(&mask[0], sizeof(MASK_T), Np, outFile);
        fclose(outFile);*/
        VELOC_Mem_save();
      }
```



LatticeQCD

- Helps understand particle dynamics (quarks, gluons)
- Based on CPS (Columbia Physics System)
- Needs to checkpoint a 1D array

HACC

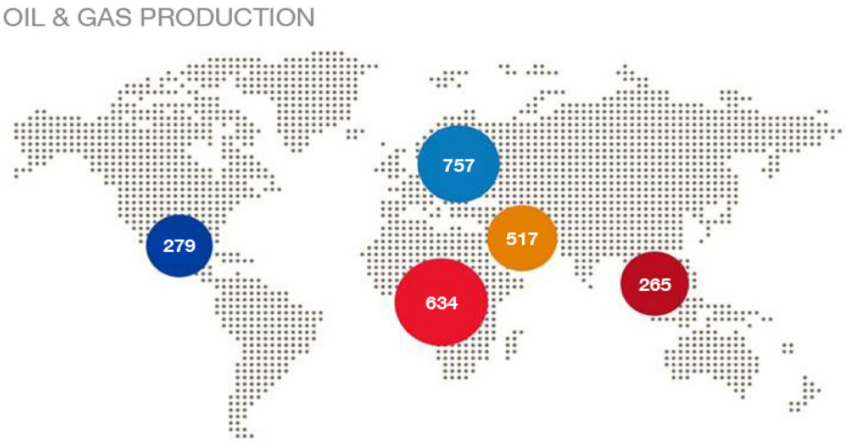
- Helps understand structure formation of universe
- Needs to checkpoint 6 x 1D arrays

Industry Interest for VELOC

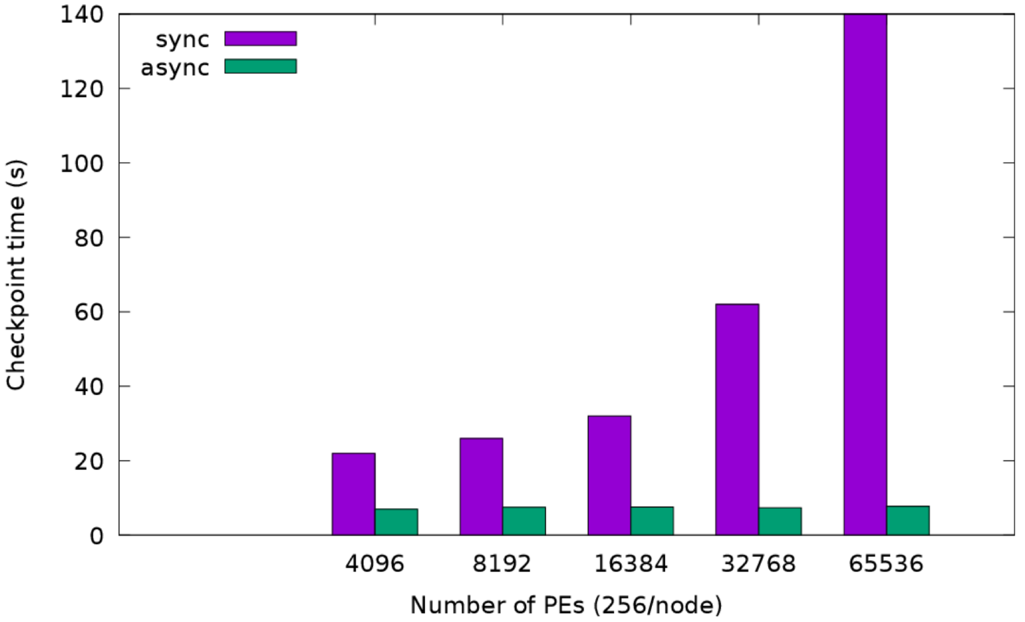
- Total SA
  - Major French oil and gas multi-national
  - Needs HPC to accelerate studies
  - Largest industrial supercomputer (6 PFlop)

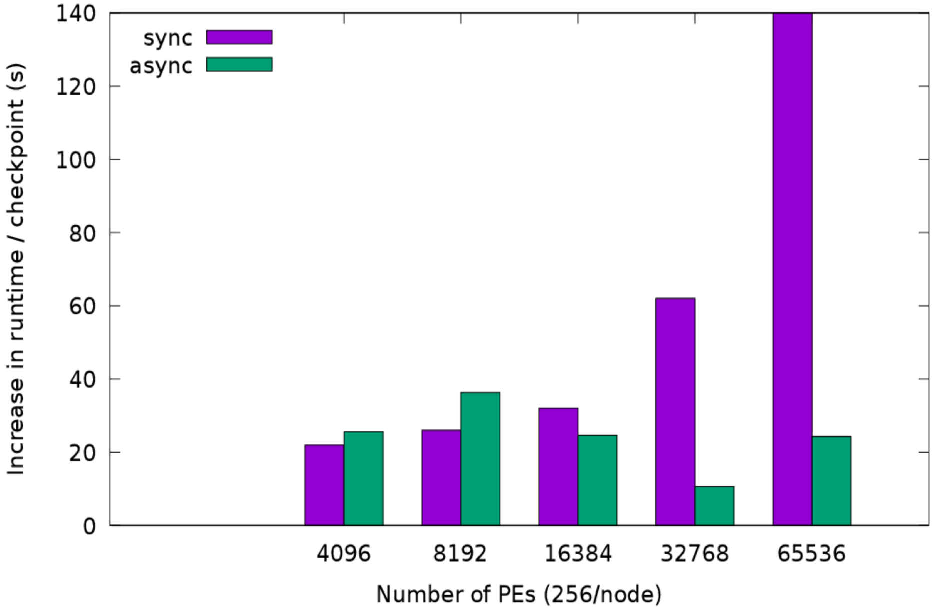
- Application: PoroDG
  - Simulations of porous media
  - Discontinuous Galerkin method
  - Written in Fortran
  - Needs efficient checkpoint-restart

- Collaborative project
  - Fortran bindings for VELOC
  - Evaluations of VELOC in progress



Results: Sync vs. Async Mode



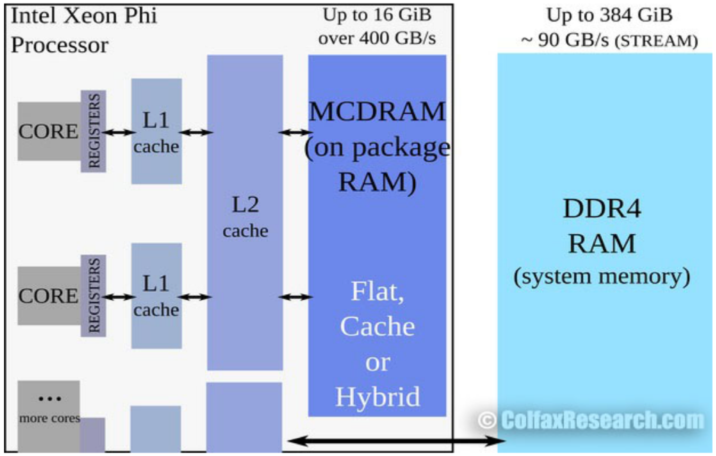


- Experimental platform: Theta (thousands of KNL nodes, Lustre PFS)
- What people did so far: blocking writes to PFS (purple)
  - The result: poor scalability
- What VeloC can do: async writes to PFS (green)
  - Apps are blocked only during local writes (on DRAM)
  - Much better scalability

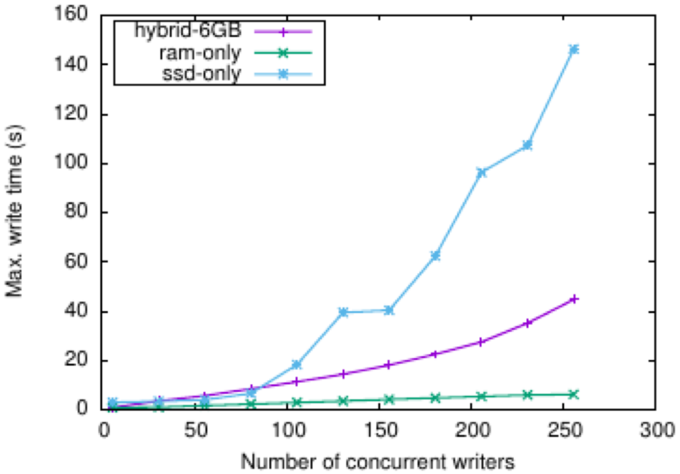
- The cost for doing async flushes to PFS:
  - They generate noticeable interference but it does not grow at scale
- Overall:
  - Rapid growing gap between sync and async with increasing #PEs

Heterogeneity of Local Storage

- Local storage is increasingly complex
- Example: KNL Node (ANL Theta)
  - MCDRAM
  - DDR4 RAM
  - Flash Storage (SSD)

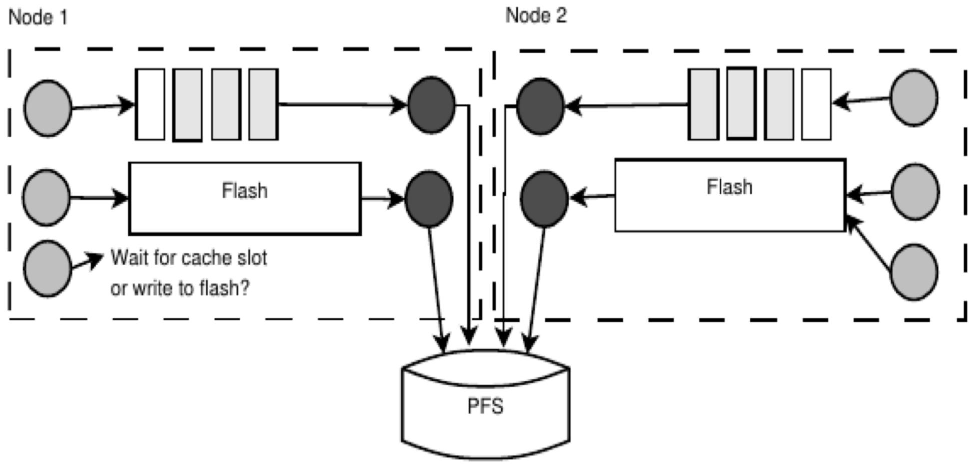


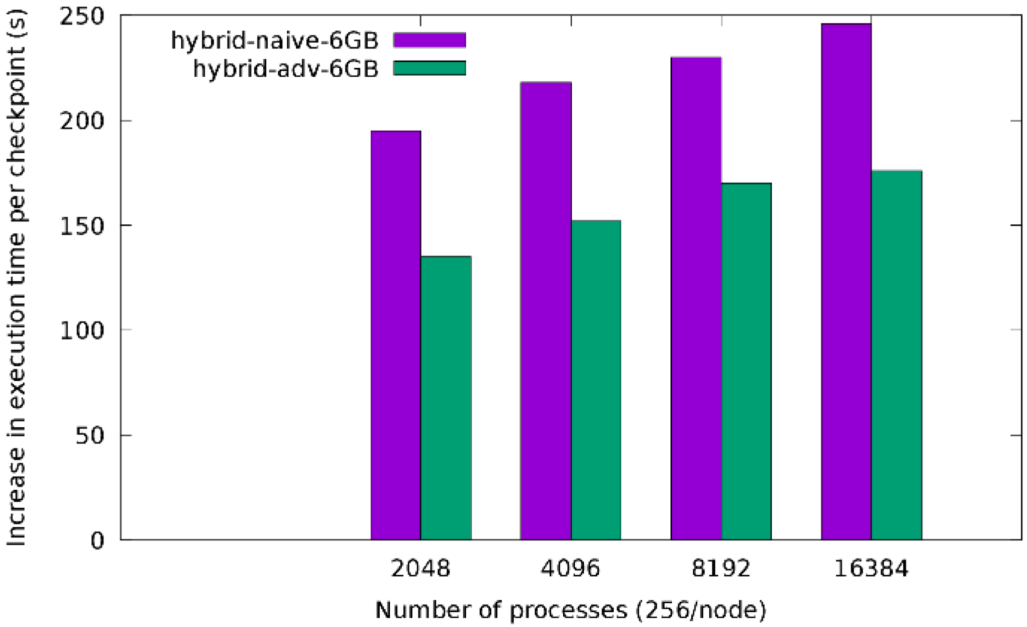
- VELOC can leverage heterogeneous local storage to improve performance
- Example:
  - Scenario: 256 concurrent writers, each writing 256 MB
  - Hybrid local storage: 6 GB DDR4 + 128 GB SSD
  - Hybrid local storage much faster than SSD only despite small DDR4 size



(b) Total time for all processes to write to local storage under increasing concurrency.

Zoom on Hybrid Local Storage

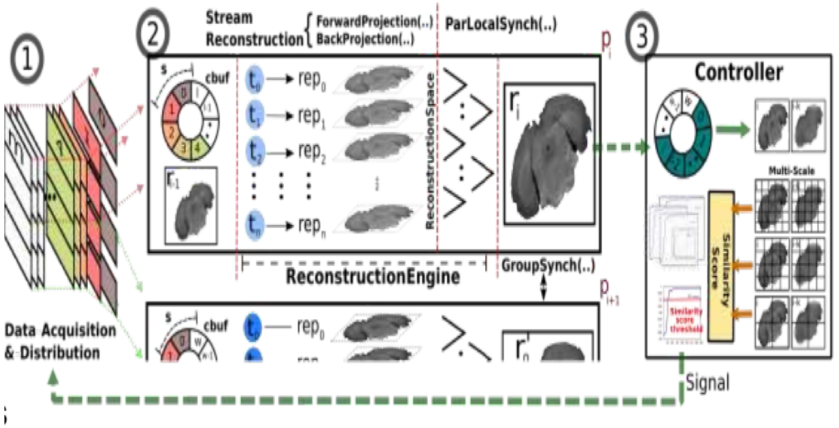




- Problem: Naive strategies that write to fastest available local storage are not enough for multi-level checkpointing
- Example:
  - Nodes equipped with small RAM cache (6 GB) and flash storage (128 GB)
  - Two resilience levels: local and parallel file system (async flush from local)
  - When RAM cache is full, if PFS is faster than flash storage, it is better to wait for RAM cache instead of writing to flash
- VELOC has a multi-level aware strategy to manage local storage
- Experiments on ANL Theta (KNL): better performance for strategy employed by VELOC vs. naive strategy

Use of CR Beyond Resilience (1)

- "Administrative" checkpointing:
  - Suspend-Resume
    - Reservations too short
    - Make room for real-time jobs
  - Migration
  - Debugging



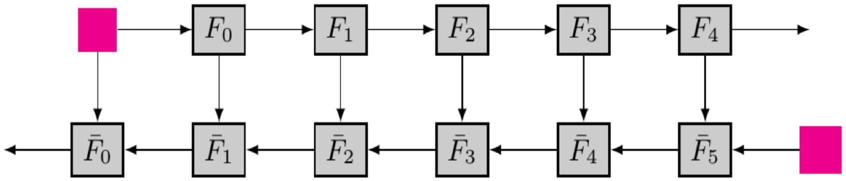
- Example: Real Time Analysis and Experimental Steering
  - Classic HPC: process and validate data only after experiment has finished
  - Issues:
    - Errors detected too late or not at all
    - Cannot act early on results
  - Solution:
    - Mix real-time stream processing (on-demand jobs) with batch jobs
    - Apply suspend-resume to batch jobs make room for on-demand jobs

Use of CR Beyond Resilience (2)

- “Productive” checkpointing:
  - Large state space that needs to be constantly revisited
  - Ensemble searches with shared states

$$F_i(x_i) = x_{i+1} \qquad i < l$$

$$\bar{F}_i(x_i, \bar{x}_{i+1}) = \bar{x}_i \qquad i \leq l$$



- Example: Adjoint Computations
  - Modelling of fluid dynamic code (e.g. atmospheric simulation)
  - Initial parameters x0, x’l + 1
  - Two phases:
    - Forward simulation (F0, F1, ...): model system using intermediate states
    - Inverse problem (F’l+1, F’l+2, ...): how well intermediate states fit goal
  - Need all intermediate states from forward simulation
  - However, there is not enough room to save them all in DRAM
  - Solution: use CR to save and restore intermediate states optimally

Conclusions

- Checkpoint-Restart at Exascale is challenging
  - High I/O contention but limited I/O bandwidth per processing unit
  - Heterogeneous storage with different performance characteristics and vendor APIs
- VELOC: Very Low Overhead Checkpointing System
  - Multi-level checkpointing delivers high performance and scalability
  - Hidden complexity of heterogeneous storage facilitates ease of use
  - Modular architecture facilitates high flexibility and extensibility
- Supports
  - Synchronous, asynchronous mode
  - Memory-based, file based API
- Results
  - Survives up to 85% of failures without need to checkpoint to parallel file system
  - Up to an order of magnitude improvement in async mode over blocking checkpointing to parallel file system

Part 2:
Hands-on Session

Installation

VeloC is available on Spack, the ECP package manager:

```
$ git clone https://github.com/spack/spack.git
$ . spack/share/spack/setup-env.sh
$ spack install veloc
```

VeloC also has its own automated installation tools:

```
$ git clone https://github.com/ECP-VeloC/VELOC.git
$ ./bootstrap.sh
$ ./auto-install.py <install_directory>
```

Installation is not covered in this tutorial

First Step: Setup

For the purpose of this tutorial, we will use a Docker image that has both ULFM and VeloC pre-installed:

```
$ apt-get install docker.io # install if needed (Ubuntu)
$ sudo usermod -aG docker $USER #log out to refresh
$ docker run hello-world #test docker installation
$ docker pull bnicolae/veloc-tutorial
```

For MAC users, follow the instructions here:
https://store.docker.com/editions/community/docker-ce-desktop-mac
You will have to create an account on DockerHub to be able to download.

The tutorial uses a sample application and some helper scripts available here:
https://goo.gl/nDtDPa

Second Step: Run Original Application

Set up aliases for **make** and **mpirun** so that they run in a Docker container based on the image previously downloaded:

```
$ . create-aliases.sh
$ alias # check the aliases
```

Compile the sample application (modeling of heat distribution):

```
$ make
```

Run the application (4 ranks per node, 256 MB per rank):

```
$ mpirun -np 4 heatdis 256 heatdis.cfg
```

Successful Output

```
Local data size is 8192 x 2051 = 256.000000 MB (256).
Target precision : 0.000010
Maximum number of iterations : 600
Step : 0, error = 1.000000
Step : 50, error = 0.484743
Step : 100, error = 0.242139
Step : 150, error = 0.161172
Step : 200, error = 0.121036
Step : 250, error = 0.096793
Step : 300, error = 0.080644
Step : 350, error = 0.069129
Step : 400, error = 0.060499
Step : 450, error = 0.053781
Step : 500, error = 0.048396
Step : 550, error = 0.043974
Execution finished in 162.528864 seconds
```

Third Step: Add VELOC Checkpointing

- Follow the comments in the source code of the application (heatdis.c)
- Replace the VELOC code comments with the missing Veloc API calls.

- Consult the documentation: **http://veloc.rtfd.io**
- Check out in particular the API section: **https://veloc.readthedocs.io/en/latest/api.html**

Third Step: Solution Part 1

Example application: Heat Distribution (included with VeloC)

Initialize VeloC:

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nbProcs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ...
if (VELOC_Init(rank, argv[2]) != VELOC_SUCCESS) {
    printf("Error initializing VELOC! Aborting...\n");
    exit(2);
}
```

Protect essential data structures:

```
nbLines = (M / nbProcs) + 3;
h = (double *) malloc(sizeof(double *) * M * nbLines);
g = (double *) malloc(sizeof(double *) * M * nbLines);
initData(nbLines, M, rank, g);
...
VELOC_Mem_protect(0, &i, 1, sizeof(int));
VELOC_Mem_protect(1, h, M * nbLines, sizeof(double));
VELOC_Mem_protect(2, g, M * nbLines, sizeof(double));
```

Third Step: Solution Part 2

Check if a previous checkpoint exists & restore essential data structures:

```
int v = VELOC_Restart_test("heatdis", 0);
if (v > 0) {
    printf("Previous checkpoint at iteration %d, initiating restart...\n", v);
             assert(VELOC_Restart("heatdis", v) == VELOC_SUCCESS);
} else // no previous checkpoint found
    i = 0;
```

Third Step: Solution Part 3

Inside the main loop, checkpoint each CKPT_FREQ iterations:

```
while(i < ITER_TIMES) {
    err = doWork(nbProcs, rank, M, nbLines, g, h);
    if (((i % ITER_OUT) == 0) && (rank == 0))
        printf("Step : %d, error = %f\n", i, globalerr);
    if ((i % REDUCE) == 0)
            MPI_Allreduce(&err, &globalerr, 1, MPI_DOUBLE, MPI_MAX, 
MPI_COMM_WORLD);
    if (globalerr < PRECISION)
            break;
        i++;
        if (i % CKPT_FREQ == 0) {
        // wait for previous checkpoint to finish (only in async mode)
            assert(VELOC_Checkpoint_wait() == VELOC_SUCCESS);
        // capture the protected data structures
            assert(VELOC_Checkpoint("heatdis", i) == VELOC_SUCCESS);
        }
}
...
VELOC_Finalize();
MPI_Finalize();
```

Fourth Step: Configure VELOC & Run

Create **veloc.cfg**, then specify the path to the local scratch directory (L0), persistent PFS directory (L3) and mode of operation (minimum mandatory parameters). L2 is disabled for a single node. The directories will be created automatically by VELOC if they don’t exist.

```
scratch = ./scratch
persistent = ./persistent
mode = sync
```

Run the application with VELOC up to iteration 250. Confirm VELOC created checkpoints:

```
$ mpirun -np 4 heatdis 256 veloc.cfg
$ ls -Al ./scratch
```

Kill the application (Ctrl+C), then run again. The application will pick up from where it left. Check the final result to confirm correctness.

Consult the documentation to learn about more configuration parameters: https://veloc.readthedocs.io/en/latest/userguide.html

Bonus: Asynchronous Mode

Edit **veloc.cfg** to activate the asynchronous mode:

```
scratch = ./scratch
persistent = ./persistent
mode = async
```

Remove all previous checkpoints and start the active backend:

```
$ rm -rf scratch persistent
$ veloc-backend veloc.cfg
```

Run the application in a different terminal, same as in sync mode:

```
$ . create-aliases.sh
$ mpirun -np 4 heatdis 256 veloc.cfg
```

Feel free to visit our web site:

http://veloc.rtfd.io

Thank you!