VELOC Documentation

Feb 11, 2022

Contents

Very-Low Overhead Checkpointing System

CHAPTER 1

Project Details

1.1 Quick Start Guide

This guide is for the impatient who wants to run and test a minimal VeloC environment on a single node to get a first impression of how it works. It can be achieved in three-steps:

1.1.1 Download and Install

```
$git clone -b 'veloc-x.y' --depth 1 https://github.com/ECP-VeloC/veloc.git <source_
↪dir>
$cd <source_dir>
$./bootstrap.sh
$./auto-install.py <install_dir>
```

Note: replace `x.y` with the latest stable version available on github.

1.1.2 Configure

Create and swich to the temporary working directory `/tmp/work`. Open `test.cfg` and add the following contents:

```
scratch = /tmp/scratch
persistent = /tmp/persistent
mode = async
```

1.1.3 Run VeloC

Open a terminal and run the heatdis example application using two MPI ranks:

```
$export LD_LIBRARY_PATH=<install_dir>/lib
$export PATH=<install_dir>/bin:$PATH
$mpirun -np 2 <source_dir>/build/test/heatdis_mem 256 test.cfg
```

If the run was successful, the scratch and persistent directories will be populated with checkpoint files of the form heatdis-x-y.dat, where is is the rank and y is the iteration number. Also, a log file should have been created where the active backend (the VELOC service running on each node which responsible for checkpoint management) displays important information: `/dev/shm/veloc-backend-<hostname>-<uid>.log`.

Now, delete one of the checkpoints with the highest version from both directories, then run the same command again. The application should restart from the checkpoint version immediately preceding the highest version and continue running to completion. After the second run, the directories should be populated with the same checkpoint files as in the case of the previous run.

Congratulations, this concludes the quick start guide!

1.2 VeloC API

This document is intended for application developers that need to integrate VeloC into their application code. It focuses on the API that VeloC exposes for this purpose.

1.2.1 API Specifications

VeloC supports two modes of operation: memory-based and file-based checkpoints. With memory-based checkpoints, an application registers regions of its memory that should be saved with each checkpoint and restored upon a restart. In this mode, the serialization of the memory regions happens automatically. With file-based checkpoints, the application has full control over how to serialize the critical data structures needed for restart into checkpoint files.

Note: the most up-to-date API specification is found in the VELOC header file: `<install_dir>/include/veloc.h`

Return Codes

All functions use the following return codes, defined as an `int` type.

- `VELOC_SUCCESS`: The function completed successfully.
- `VELOC_FAILURE`: Indicates a failure. VeloC prints a corresponding error message when this error code is returned. In the future, more error codes may be added to indicate common error scenarios that can be used by the application to take further action.

Initializing and Finalizing VeloC

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()`.

Non-collective initialization

```
int VELOC_Init_single(unsigned int unique_id, IN const char *cfg_file)
```

ARGUMENTS

- **unique_id**: A unique identifier of the process that needs to checkpoint individually
- **cfg_file**: The VeloC configuration file, detailed in the user guide.

DESCRIPTION

This function initializes the VELOC library in non-collective mode, which enables each process to checkpoint and restart independently of the other processes. Notably, this will impact the behavior of `VELOC_Restart_test` (detailed below), which will return the latest version available for the calling process, rather than the whole group.

Finalize

```
int VELOC_Finalize(IN int drain)
```

ARGUMENTS

- **drain**: a bool flag specifying whether to wait for the active backend to flush any pending checkpoints to persistent storage (non-zero) or to finalize immediately (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()`.

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 base_size)
```

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.

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(IN char *original_name, OUT char *ckpt_file_name)
```

ARGUMENTS

- **ckpt_file_name**: The name of the checkpoint file that the user needs to use to perform I/O
- **original_name**: The original name of the checkpoint file. VELOC will use **ckpt_file_name** internally

but will stick to the original name when persisting the checkpoint on the parallel file system. This enables users to customize the checkpoint namespace to facilitate their use for other purposes than restart (e.g. analytics).

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).

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.

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.

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.

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.

Memory-based Restart

```
int VELOC_Recover_selective(IN int mode, INT int *ids, IN int length)
```

ARGUMENTS

- **mode** : One of VELOC_RECOVER_ALL (all regions from the checkpoint, ignores rest of arguments), VELOC_RECOVER_SOME (regions explicitly specified in ids), VELOC_RECOVER_REST (all regions except those specified in ids)
- **ids** : Array of ids corresponding to the memory regions previously saved in the checkpoint
- **length**: Numer of elements in array of ids

DESCRIPTION

This function restores the memory regions from the checkpoint speficied when calling `VELOC_Restart_begin()`. Must be called between `VELOC_Restart_begin()` and `VELOC_Restart_end()`. For all ids that will be restored, a previous call to `VELOC_Mem_protect()` must have been issued. The size of the registered memory region must be large enough to fit the data from the checkpoint. A typical use of this function relies on VELOC_RECOVER_SOME to figure out the size of data structures (assumed to be saved into the checkpoint), allocate and protect memory regions large enough to hold them, the use VELOC_RECOVER_REST to restore the content.

```
int VELOC_Recover_mem()
```

ARGUMENTS

- None

DESCRIPTION

This is a convenience wrapper equivalent to calling `VELOC_Recover_selective(VELOC_RECOVER_ALL, NULL, 0)`

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()`.

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.

1.2.2 Example

To illustrate the API, we have included with VeloC a sample MPI application that simulates the propagation of heat in a medium. This application can be found in the `test` sub-directory and includes both the original and two modified versions that use VeloC: one using the memory-based API (`heatdis_mem`) and the other using the file-based API (`headis_file`).

Original Code

In a nutshell, the original heatdis application has the following basic structure:

```
MPI_Init(&argc, &argv);
// further initialization code
// allocate two critical double arrays of size M
h = (double *) malloc(sizeof(double *) * M * nbLines);
g = (double *) malloc(sizeof(double *) * M * nbLines);
// set the number of iterations to 0
i = 0;
while (i < n) {
    // iteratively compute the heat distribution
    // increment the number of iterations
    i++;
}
MPI_Finalize();
```

Memory-based API

To add checkpoint/restart functionality using VeloC in memory-based mode, several modifications are necessary: (1) initialize VeloC (immediately after `MPI_Init`); (2) register the memory regions corresponding to the critical arrays; (3) check if there is a previous checkpoint to restart from using `VeloC_Restart_test`; (4) if yes, restore the memory regions to their initial state; (5) every K iterations initiate a checkpoint; (6) finalize VeloC before calling `MPI_Finalize`. This is illustrated below:

```
MPI_Init(&argc, &argv);
VELOC_Init(MPI_COMM_WORLD, argv[2]); // (1): init
// further initialization code
// allocate two critical double arrays of size M
h = (double *) malloc(sizeof(double *) * M * nbLines);
g = (double *) malloc(sizeof(double *) * M * nbLines);
// (2): protect
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));
// (3): check for previous checkpoint version
int v = VELOC_Restart_test("heatdis", 0);
// (4): restore memory content if previous version found
if (v > 0) {
    printf("Previous checkpoint found at iteration %d, initiating restart...\n", v);
    // v can be any version, independent of what VELOC_Restart_test is returning
    assert(VELOC_Restart("heatdis", v) == VELOC_SUCCESS);
 } else
     i = 0;
 while (i < n) {
     // iteratively compute the heat distribution
     // (5): checkpoint every K iterations
     if (i % K == 0)
         assert(VELOC_Checkpoint("heatdis", i) == VELOC_SUCCESS);
      // increment the number of iterations
      i++;
 }
 VELOC_Finalize(0); // (6): finalize
 MPI_Finalize();
```

File-based API

To add checkpoint/restart functionality using VeloC in file-based mode, the same modifications are needed as in the case of memory-based API mode, except for the checkpoint and restart, which need to be manually implemented:

Checkpoint

```
if (i % K == 0) {
    assert(VELOC_Checkpoint_wait() == VELOC_SUCCESS);
    assert(VELOC_Checkpoint_begin("heatdis", i) == VELOC_SUCCESS);
    char veloc_file[VELOC_MAX_NAME];
    assert(VELOC_Route_file(veloc_file) == VELOC_SUCCESS);
    int valid = 1;
    FILE* fd = fopen(veloc_file, "wb");
    if (fd != NULL) {
        if (fwrite(&i, sizeof(int),            1, fd) != 1)         { valid = 0; }
        if (fwrite( h, sizeof(double), M*nbLines, fd) != M*nbLines) { valid = 0; }
        if (fwrite( g, sizeof(double), M*nbLines, fd) != M*nbLines) { valid = 0; }
        fclose(fd);
    } else
        // failed to open file
        valid = 0;
    assert(VELOC_Checkpoint_end(valid) == VELOC_SUCCESS);
}
```

Restart

```
assert(VELOC_Restart_begin("heatdis", v) == VELOC_SUCCESS);
char veloc_file[VELOC_MAX_NAME];
assert(VELOC_Route_file(veloc_file) == VELOC_SUCCESS);
int valid = 1;
FILE* fd = fopen(veloc_file, "rb");
if (fd != NULL) {
    if (fread(&i, sizeof(int),            1, fd) != 1)          { valid = 0; }
    if (fread( h, sizeof(double), M*nbLines, fd) != M*nbLines) { valid = 0; }
    if (fread( g, sizeof(double), M*nbLines, fd) != M*nbLines) { valid = 0; }
    fclose(fd);
} else
    // failed to open file
    valid = 0;
assert(VELOC_Restart_end(valid) == VELOC_SUCCESS);
```

1.3 User Guide

This documentation is intended for users who need to run applications that make use of VeloC for checkpoint/restart.

1.3.1 Setup

Due to the large number of software and hardware configurations where VeloC can run, it must be built from source. Once built and installed, VeloC needs to be configured using a configuration file. These aspects are detailed below:

Download VeloC

The source code of VeloC is publicly available on `github`. To download it, look for the latest stable version x.y, which should appear under the 'Releases/Tags' tab as 'veloc-x.y'. Then, use the following command:

```
git clone -b 'veloc-x.y' --single-branch --depth 1 https://github.com/ECP-VeloC/veloc.
↪git
```

If you want to experiment with the latest development version, you can directly check out the main branch. This is helpful to stay up-to-date with the latest features. The main branch is extensively tested and can be considered stable for most practical purposes.

```
git clone --single-branch --depth 1 https://github.com/ECP-VeloC/veloc.git
```

Install VeloC

VeloC has an automated installation process based on Python, which depends on several standard libraries. These standard libraries may not be present on your system. If that is the case, you need to bootstrap the installation process first as follows:

```
$./bootstrap.sh
```

Once the bootstrapping has finished, the `auto-install.py` script will build and install VeloC and all it dependencies. The installation can be fine-tuned with several options, which can be listed by supplying the "–help" switch. Notably, you can control the installation directory, communication protocol between the client and the backend, whether

to use pre-installed libraries, etc. The script can be edited to modify certain compiler options if needed. Common compiler options needed for some machines (e.g. Cray) are included as comments. After editing the script, run it as follows:

```
$./auto-install.py <install_dir>
```

Note that it may be possible that your Python installation will not detect the libraries installed by the bootstrapping automatically. In this case, the `auto-install.pu` script will fail. Locate the installed libraries and tell Python about them as follows:

```
$setenv PYTHONPATH ~/.local/lib/python3.6/site-packages
```

If the installation process was successful, the VeloC client library (and its dependencies) are installed under `<install_dir>/lib`. The `veloc.h` header needed by the application developers to call the VeloC API is installed under `<install_dir>/include`. The active backend needed to run VeloC in asynchronous mode can be found in `<install_dir/bin/veloc-backend`. The examples can be found in `<source_dir>/src/test`, while the corresponding compiled executables are here: `<source_dir>/build/test`.

Configure VeloC

VeloC uses a INI-style configuration with the following mandatory fields:

```
scratch = <path> (node-local path where VELOC can save temporary checkpoints that␣
↪live for the duration of the reservation)
persistent = <path> (persistent path where VELOC can save durable checkpoints that␣
↪live indefinitely)
```

In addition, the following optional fields are available:

```
persistent_interval = <int> (seconds between consecutive persistent checkpoints,␣
↪default: 0 - perform all)
ec_interval = <int> (seconds between consecutive EC checkpoints, default: 0 - perform␣
↪all)
watchdog_interval = <int> (seconds between consecutive checks of client processes:␣
↪default: 0 - don't check)
max_versions = <int> (number of previous checkpoints to keep on persistent, default:␣
↪0 - keep all)
scratch_versions = <int> (number of previous checkpoints to keep on scratch, default:␣
↪0 - keep all)
failure_domain = <string> (failure domain used for smart distribution of erasure␣
↪codes, default: <hostname>)
axl_type = <string> (AXL read/write strategy to/from the persistent path, default:
↪<empty> - deactivate AXL)
chksum = <boolean> (activates checksum calculationa and verification for checkpoints,␣
↪default: false)
meta = <path> (persistent path where VELOC will save checksumming information)
```

Both the persisten and ec interval can be set to -1, which fully deactivates that feature. This is preferred to setting a high number (which also works but is less readable and has slightly higher overhead because VeloC will need to do extra checks). If you leave scratch_versions to the default value, you must ensure the scratch mount point will run out of space. VELOC will not automatically delete checkpoints when space is low. If space is a concern, set scratch_versions accordingly. Similar observations apply for the persistent mount point, for which the corresponding option is max_versions. Finally, you can specify whether to use a built-in POSIX file transfer routine to flush the files to a parallel file system or to use the AXL library for optimized flushes that can take advantage of additional hardware to accelerate I/O (such as burst buffers). If the use of AXL is desired, you need to specify as `axl_type` as per the AXL documentation (which is part of VELOC). Note that VELOC uses a separate `meta` path for checksumming

information, instead of writing checksumming information directly into the checkpoints. Thus, it is perfectly valid to save checksumming information during checkpointing but then delete or ignore it later on restart (in which case the `meta` option must be omitted).

1.3.2 Execution

VeloC can be run in either synchronous mode (all resilience strategies are embedded in the client library and run directly in the application processes in blocking fashion) or asynchronous mode (the resilience strategies run in a separate process called the active backend in asynchronous mode in the background).

To use VeloC in synchronous mode, the application simply needs to be run as any normal MPI job. To run VeloC in asynchronous mode, you need to make sure the `veloc-backend` executable can either be found in the `$PATH` or `$VELOC_BIN` environment variable. This is true for every node running the MPI ranks of your application (thus, `veloc-backend` should be accessible through a shared mount point). By default, `veloc-backend` will create the following log file on each node: `/dev/shm/veloc-backend-<host_name>-<uid>.log`. The log file contains important information (error messages, time to flush to PFS, etc.) that you may want to collect and inspect while/after running your application. In this case, you can control where the log files are saved using `$VELOC_LOG` environment variable (e.g., a shared directory).

Examples

VeloC comes with a series of examples in the `test` subdirectory that can be used to test the setup. To run these examples (in either synchronous or asynchronous mode), edit the sample configuration file `heatdis.cfg` and then run the application as follows (run the active backend first as mentioned above if in async mode):

```
mpirun -np N <source_dir>/build/test/heatdis_mem <mem_per_process> <config_file>
```

1.3.3 Batch Jobs

HPC machines are typically configured to run the user applications as batch jobs. Therefore, the user needs to make sure that the job scheduler is not configured to kill the entire job when a node fails. Assuming the job scheduler is configured correctly, the user needs to write a script as follows:

```
reserve N+K nodes (to survive a maximum of K total failures over the entire␣
↪application runtime)
do
    run the application (on the surviving nodes)
while (failure detected) // e.g, exit code of the application
```

- search