REMORA v2

REsource MOnitoring for Remote Applications
Document Revision 1.0
August 28, 2023

Antonio Gómez-Iglesias, Carlos Rosales Fernández (Version 1)
`agomez@tacc.utexas.edu`, `carlos@tacc.utexas.edu`

Chun-Yaung, Kent Milfeld (Version 2)
`cylu@tacc.utexas.edu`, `milfeld@tacc.utexas.edu`

High Performance Computing
Texas Advanced Computing Center
The University of Texas at Austin

Abstract

REMORA is a modular tool to monitor a job runtime resource utilization in HPC environments. As of version 2.0 the suite contains code designed to test:

- Memory utilization in CPUs, Xeon Phi co-processors, and NVIDIA GPUs
- CPU utilization
- IO utilization for the Lustre and DVS file systems
- NUMA properties
- Network topology
- MPI communication statistics
- Socket Power consumption
- CPU temperature

Throughout the text we have used text boxes to highlight important information. These boxes look like this:

These gray boxes contain highlighted material for each section and chapter.

Remora was designed to provide a runtime resource monitoring tool that is both simple to use and presents easy to understand high level information of resource usage, as well as detailed statistics for in-depth analysis. We welcome all feedback, particulary suggestions that includes improvement in the usability, reliability, and accuracy.

Version 2.0 is a feature release that has been optimized to reduced collection overhead, programmed for attachment to running jobs, power collection for all intel platforms (with rapl), provide easier control over resource module selection, and support unit testing.

REMORA is an open-source project. Please help support continued development/improvement by citing Remora:

C. Rosales, A. Gómez-Iglesias, A. Predoehl. "REMORA: a Resource Monitoring Tool for Everyone". HUST2015 November 15-20, 2015, Austin, TX, USA. DOI: 10.1145/2834996.2834999 [1]

Chun-Yaung Lu and Kent Milfeld. "REMORA Resource Monitor: Usability, Performance and User Interface Improvements". HUST2023 November 15-20, 2023,, USA. DOI: 10.1145/2834996.2834999 [1]

Contents

1 Installation

REMORA is simple to install. In order to use all of its features you will need GNU Make and a C compiler. Support for MPI statistics requires an MPI installation with an MPI compiler wrappers and runtime.

Download the latest tar version from the following URL in your browser: [1] and extract the contents in your system:

```
tar xvf *remora*.tar
```

Alternatively clone the git repository with the latest development version (not recommended):

```
git clone https://github.com/TACC/remora
```

This will create a top level directory called `remora`, with subdirectories `/docs`, `/extra` `/src`. Change directory to remora and edit `install.sh` to reflect your choice of installation directory, build type, and modules configuration. The variables to modify are:

Variable	**Description**	**Default**
REMORA_DIR	Absolute path to installation directory	.

The resource data collectors, modules, are configured in the `src\config\modules` file. Simply remove any modules that you don't want to be available in your system. Also, you can create your own modules and add them into this file. (See Section 4 for more information about how to expand REMORA's functionality).

By default Remora is installed in the present working directory (`remora`). Change compiler variables in the install script, install.sh; also the install directory can be set by directly assigning a path to `REMORA_DIR`. (`REMORA_INSTALL_PREFIX`, PWD is the default) in install.sh. Then, execute install.sh to build and install remora.

```
./install.sh |& tee install.log
```

An installation path can also be specified on the command line:

```
REMORA_INSTALL_PREFIX=<install_directory> ./install.sh |& install.log
```

1.1 More on MPI Support

If your systems supports multiple MPI stacks you should build a version for each of stack, since the collection of MPI statistics is performed differently for each stack. The installation script will

[1] https://github.com/tacc/remora/tarball/master

detect and set the mpiexec and mpicc / mpif90 compilers of the present environment. Simply use a different installation prefix for each of the stacks so the installations do not overwrite each other. When no MPI compiler is present during the installation, remora will be installed with full functionality except for the missing MPI statistics support.

After installing remora, make sure the remora `bin` directory is prepended to your `PATH` and the remora `lib` directory is in your `LD_LIBRARY_PATH`. (Remora determines the `bin` path from your PATH variable, sets `REMORA_BIN` to its value and uses the variable in remote processes to specify the location of remora execution scripts.)

For SH/BASH/ZSH users, add the follow to an appropriate startup file to ensure REMORA is available every time you login.

```
export PATH=$REMORA_DIR/bin:$PATH
export LD_LIBRARY_PATH=$REMORA_DIR/bin/lib:$LD_LIBRARY_PATH
```

If the mkmod utility is installed on your system, after installation, cd to the install directory (where bin exists), and execute the follow to create a my_remora user-space module:

```
module load mkmod
export NAME=remora VER=2.0 TOPDIR=‘pwd‘
        mkmod
```

After this, the command module load my_remora will create an execution environment for you (with the correct PATH, LD_LIBRARY_PATH, and module help information).

2 Using REMORA

2.1 Collecting Data with remora Wrapper

If you are on a system that manages application environments with modules, load remora:

```
module load remora
```

Otherwise, it is assumed that the remora command and library are in your PATH and LD_LIBRARY_PATH variables, respectively.

For a serial job:

```
    export REMORA_PERIOD=2     #change default period from 10 sec. to 2
remora app [app_args]
```

For a parallel job (assuming `mpirun` is your parallel launcher, change for mpiexec.hydra, ibrun or other as needed):

```
remora mpirun [mpirun_options] mpi_app [mpi_app_args]
```

The code will run as normal, and contain remora summary printed to stdout at the end. Also, a directory named `remora_<jobid>`, where `jobid` is the job number, will contain remora data files (*.txt) and an html index file (remora_summary.html) linking to the Google Charts (*.html), that you can open on your laptop to view the resource usage. Currently SGE and SLURM are supported.

The Remora output directory will contain a set of subdirectories with information about different resources used by your code – See table 2.1 – as well as a summary output in your terminal

Directory	**Contents**
CPU	CPU utilization
INFO	Runtime environment and hostlist
IO	File system utilization
MEMORY	Memory utilization
MONITOR	Continuous monitoring data (MONITOR mode only)
NETWORK	IB utilization data
NUMA	Non-Uniform Memory Access data
POWER	Power consumption
TEMPERATURE	CPU temperature

Table 2.1: Contents of output directory

```
=============================== REMORA SUMMARY ===============================
 Max Virtual  Memory Per Node :     6.08 GB
 Max Physical Memory Per Node :     1.23 GB
 Available Memory at time 0.0 :   177.56 GB
 MPI Communication Time       :    13.95 %
 Total Elapsed Time           :  0d 0h  0m 55s 276ms
------------------------------------------------------------------------------
 Max IO Load / home1          :      68 IOPS      42 RD(MB/S)       0 WR(MB/S)
 Max IO Load / scratch        :      70 IOPS       1 RD(MB/S)       0 WR(MB/S)
 Max IO Load / work           :       1 IOPS       0 RD(MB/S)       0 WR(MB/S)
==============================================================================
 Sampling Period              : 2 seconds
 Report Directory             : R=/scratch/00770/milfeld/amber/remora_5599286
 Google Plots HTML Index Page : $R/remora_summary.html
==============================================================================
```

Figure 2.1: Summary output for typical job execution using REMORA

screen (interactive use) or stdout file (batch use). An example of the summary output is shown in Figure 2.1.

The summary shown in Figure 2.1 shows that the maximum virtual and physical memory used per node in this particular execution was 6.08 GB and the executable ran for just over 55 seconds, with 14% of the total execution time spent in MPI routines (inclusive of communication and MPI IO). From the output we can also see that the job was run out of the `scratch` file system, and that very small amounts of IO occurred throughout the execution. As indicated by the `Sampling Period` entry data was collected using a `REMORA_PERIOD` of 2 seconds, and the complete collected data set can be found in the `remora_5599286` directory. The maximum IO was 42 MB/s with only 68 IOPS maximum. This summary information is saved in `remora_5599286/INFO` directory.

2.2 Execution Customization

REMORA is configurable in terms of the amount and type of data collected, but sensible defaults are provided to simplify its use. By default the resource data are collected every ten seconds.

REMORA provides two different running modes and it also allows the users to specify how frequently the data is collected. A verbose mode is provided mostly for troubleshooting and should not be used by default. The behavior of the application is controlled via four environment variables:

- `REMORA_MODE`: this variable accepts three possible values (BASIC, FULL, and MONITOR). The FULL mode runs all the tests that the tool allows. The BASIC mode only reports memory and cpu usage. MONITOR mode is equivalent to FULL, with the added advantage that data is post-processed inline and a summary file is generated in real time for application monitoring. BASIC is the recommended mode when the users know that the application of interest does not create problems in the distributed file system. The default is FULL.
- `REMORA_PERIOD`: Time in seconds between consecutive data records. This is the time from the end of a collection event until the start of the next collection event. Depending on the platform where the tool is running, the overhead introduced by the application can make the duration of the collection event to vary, in which case there will be less data points in the collected results than expected. However, in the systems that we have tested the overhead of the application is small enough that the total number of collection points (CP) is almost equal to $CP = ET/RP$ where ET is the execution time (in seconds) and RP is the period (in seconds).
- `REMORA_MONITOR_PERIOD`: Similar to `REMORA_PERIOD`. It corresponds to the number of seconds between updates to the monitor file which contains the real time summary of resource utilization. This is provided in case regular monitoring data is required at a different rate than the full data collection. It must always be larger than `REMORA_PERIOD`.
- `REMORA_TMPDIR`: Full path for intermediate file storage. It is recommended that this is a local disk. Default value is the location of the REMORA output directory (which must be on a shared file system). When specified by the user REMORA will collect data in this location, and only copy the files to the output directory once data collection is complete. Using a local file system for the temporary files reduces overhead.
- `REMORA_PLOT_RESULTS`: When this variable is unset, by default, REMORA automatically generates a set of HTML files with plots for the collected data. However, for very large use cases, this automatic generation can take a long time. Users can skip the generation of these files by setting this variable to 0. A value of 1 designates automatic generation.
- `REMORA_VERBOSE`: Enable (1) or disable (0) verbose mode. Default is disabled.
- `REMORA_WARNING`: Verbosity level for REMORA issued warnings. Acceptable values are 0, 1, 2 in increasing level of verbosity. Critical errors will always be reported independently of the chosen value.

In addition to these variables a list of file systems that should be ignored during data collection can be provided to REMORA in the file `config/fs_blacklist`. This is a simple text file with a names of mounts that should be ignored during runtime.

2.3 Collecting Data with snapshot, after attachment

The `snapshot` command is used to gather remora data from a running job. After accessing (attaching to) a compute node of a running job (with ssh), load the remora module and execute `snapshot` in its simplest form, with the number of data points (required) to be collected. To collect 100 snapshots (data values) for the default period, execute: The number of snapshots is required.

```
snapshot 100
```

The command has options that allow a user to set the period (an integer) on the command line (or set the REMORA_PERIOD env. var.). Also, The arguments can be a list of the collection modules to use. No order is required, except for the number of snapshots, which must be first.

```
snapshot num_snapshots [period] [space-separated list of modules]
  e.g.
snapshot 100  10 memory mpi   #100 snapshots 1_snapshot/10sec
                                      #collect only memory and mpi info
```

By default, data is collected for all nodes in a job. Use the argument `node` to just collect data for the node you are presently accessing. Also, by default plots are not created for a snapshot execution.

The plots are not automatically generated in `snapshot` because plots are created on each node, and will compete with the execution of your job.

If this is of no concern, add the `plot` to the snapshot command to create plots immediately after data collection.

2.4 Plotting snapshot data

After collecting data from `snapshot`, execute `snapshot` with `plot` as the first argument on a login node in the directory where the snapshots were taken:

```
snapshot plot         #execute on login node, remora_<job_id> in PWD
or
snapshot plot remora_<job>      #supply remora_<job_id> path
```

When plotting the snapshot data, the data in the latest remora directory (remora_¡job_id¿) in the present working (top level) directory is found and used for the plots. A different remora_¡job_id¿ may be suppliede, or a full path to a remora_¡job_id¿ directory can be added as an argument if the present working directory is not a top level directory.

2.5 Post-Processing

All the data is collected in a set of files with the metrics organized in columns. Since remora data consists of ASCII files, with space separated column data, with each row pre-pending with a time stamp, it is easy to extract remora data with your own postprocessing tools. However, for simplicity, REMORA already provides a plotting script called `remora_post` that takes all the metrics generated during collection and generates plots. These plots show the most relevant information previously collected and represent a visual alternative while analyzing the results. The script can be called from the batch script or from the login nodes after the job has finished. In this second scenario, the script requires an argument with the job ID to be analyzed.

The post-processing script can be invoked simply as:

```
remora_post -j <jobID>
```

2.6 Post-Crash Summary

In a typical situation, when an application crashes, the remora collection is interrupted and no final summary is produced (or an incomplete summary is shown). We now provide an independent script `remora_post_crash` that attempts to produce a final summary of the collected data. This script can be executed as:

```
remora_post_crash <JobID>
```

2.7 Automated Memory Protection

Following user requests we have developed a simplified version of remora that does not produce any data collection records but simply monitors application memory usage and kills the user application if available memory on each node reaches a minimum threshold.

For a serial job:

```
remora_mem_safe ./myexe [myexe_arguments]
```

For a parallel job (assuming `mpirun` is your parallel launcher, change for mpiexec.hydra, ibrun or other as needed):

```
remora_mem_safe mpirun [mpirun_options] ./myexe [myexe_arguments]
```

The minimum memory available is configured by default to be 0.5GB, and the sampling period is 0.01 seconds. The sampling period is not configurable, but the memory threshold can be modified.

- `REMORA_FREEMEM`: this variable sets the minimum available memory (in KB) for `remora_mem_free`. If available memory falls below this threshold all user processes under monitoring will be killed.

3 Design and Implementation

REMORA is designed with ease of use in mind. Use with minimal effort is fundamental throughout the model. This is included in access, operation, employment of features, understanding collection tables and reports and visual presentations. While there are models where the tool could run transparently, just by loading a module that would change the environment to collect all the information at runtime, we decided to opt for a model where the user loads the module, and then changes the submission script to invoke the remora tool by prepending its name to the actual command that has to be analyzed. This has the advantage of preventing unnecessary overhead when the module is loaded and runs that do not require instrumentation are submitted. It also simplifies the data collection for serial jobs and scripts, which do not use an MPI launcher that can be easily hijacked or modified. For example, if the original command is *mpirun ./myparallelapp*, the new command will be *remora mpirun ./myparallelapp*. A more complicated scenario is when the user wants to run a set of different commands or scripts in the same job. In that case, it is necessary to put all the commands to be executed in a shell script (i.e. a shell script called *mycommands*). Then, executed the script with the remora wrapper: *remora ./myjob*. Note, `remora` can be used in a batch script or interactively in the command prompt.

As a system-installed tool, the remora environment will be accessible by loading a Lua or Tcl resource module (module load remora). Users can install their own version Remora. Users can create their own module (with mktau), or simply add environment setup commands to their startup shells (the exact commands are listed at the end of the install).

In its current implementation the tool generates a flat ssh tree with a single connection from the master node in the execution to all other nodes. This connection initiates a background process that collects the statistics for each node with a frequency specified by the user. The remote background tasks execute a loop over all processes owned by the user, and aggregate the data before committing it to file.

For runs involving a Xeon Phi co-processor the background task is pinned to the last core (assumed to own hardware threads 0, 241, 242 and 243) since in most execution modes this will avoid interference with the application during runtime and minimize impact on the execution time. The source file `mic_affinity.c` can be modified in order to change this setting.

In monitor mode, a subset of the metrics is appended to a file that the user tails in real time.

3.1 Data Collection

REMORA collects a set of metrics that is useful in many different scenarios when profiling and evaluating resource usage of an application. The data collected by REMORA consist of:

- Detailed timing of the application.
- CPU loads.
- Memory utilization.
- NUMA information.
- I/O information (file system load and Lustre traffic).

- Network information (topology and InfiniBand traffic).
- MPI Statistics (time, most used MPI calls, message sizes)
- Temperature (of each core)
- Power (of each socket)

Dynamic information is collected every `REMORA_PERIOD` seconds. The following describes the data collected in more detail.

CPU The application reports the average CPU usage of the last second (independently of the value specified for `REMORA_PERIOD`). This information is very important for applications that use OpenMP, where it is possible to easily analyze how the cores are being used. It also allows to check for a correct pinning of threads to the cores: not pinning processes could lead to threads floating between cores, which will be show up in this report. MPI applications can also benefit from this information.

Memory One of the most recurring questions for HPC users is "how can I know how much memory my job is using?". Trying to answer that question, REMORA collects the most relevant statistics regarding memory usage every `REMORA_PERIOD` seconds (more information in Section 2.2):

- Virtual Memory (and Max Virtual Memory): this is a very important value as the OOM (out of memory) killer will use it to kill the application if needed.
- Resident Memory (and Max Resident Memory): physical memory used by the application.
- Shared memory: applications have access to shared memory by means of /dev/shm. Any file that is put there counts towards the memory used by the application, so the application reports this usage.
- Total free memory: this will take into account the memory not being used by the application, the libraries needed by the application, and the OS.

Data is collected from /proc/<pid>/status for all of these except shared memory, which can be obtained from /dev/shm. Memory usage for all user processes is aggregated and written to a single file per node involved in the execution. At the end of the run the maximum values for memory utilization (and minimum value of total free memory) are aggregated into a single file.

When Xeon Phi co-processors are used as part of a symmetric execution model, each Xeon Phi is treated as a separate node and the same memory information is collected from Phi and from host CPU. Individual files are maintained for each node and Phi and the per node aggregated summary is provided individually for nodes and Xeon Phi devices, since their available memory tends to be different.

NUMA As it is well known, NUMA (Non-Uniform Memory Access) can have a large impact on the overall performance of an application. Sometimes small changes in the code can lead to large improvements once it has been discovered that NUMA was imposing a penalty over the application. Our tool reports how memory is being used in each socket and it also collects the number of NUMA hits and misses. The information is extracted from the `numastat` tool: `numastat` is called only once on each collecting period; the output of `numastat` is then analyzed and several different fields are used for the statistics:

- Number of hits: total number of memory access hits.
- Number of misses: total number of memory access misses.
- Number of hits in the current node: if the data that the application was looking for is in the same of node where the core requesting that data is located, it produces a hit in the current NUMA node.
- Number of hits in the other node: when the data required is in cache, but in the other NUMA node.
- Total memory free/used on each node.

Lustre A new Lustre module was included in REMORA 1.4.0. This module looks at the content of the files located in `/proc/fs/lustre/{mdc,osc}/*/` . In particular, it looks for the content of the stats file. In order to generate a more user-friendly data, and it extracts the name of the filesystems and the different lustre mounts from the `df` command.

DVS A new Data Virtualization Service (DVS) module was added in REMORA 1.5.0. This module captures information in the `/proc/fs/dvs/mounts/*/` files to provide the number of requests per second for every DVS served file system that is not in the blacklist under `/config/fs\_blacklist`.

LNET Stats Our tool collects information regarding the data transferred by Lustre on each node used by the job while running. Normally, these statistics do not provide much information to the users. However, they are very useful if there was a problem in the file system while the job was being analyzed, as the number of messages dropped will significantly increase. The following Lustre information is collected:

- Number of currently active Lustre messages. It also includes a highwater mark of this value.
- Messages sent/received: total number of Lustre messages sent/received by the current node.
- Messages dropped: number of Lustre messages that failed to be delivered to the destination.
- Bytes sent/received: total number of bytes sent/received on Lustre messages.

InfiniBand Packets Number of packets transmitted using InfiniBand. This data can be used to get extra information regarding how the communication in parallel applications takes place. In particular, the time series can be used to correlate high network activity levels with sections of the code, and those ections can be revised for possible optimization.

MPI Statistics MPI statistics are collected using mpiP for Mvapich2 and the internal mechanisms for Intel MPI. The percentage of execution time spent in MPI calls is included in the summary report, and the raw data stored in the MPI/mpi_data.txt file. A simple pie chart shows the percentage of time spent in MPI calls – including communication and IO – and a bar chart is provided showing the five MPI cals where the application spent most of its communication time.

3.2 Modular Design

REMORA presents a modular design that makes it easy to modify and extend the functionality provided by the tool. For example, currently REMORA supports the Lustre file system, but there are HPC systems that use other types of file systems. Data collection for those file systems can be incorporated into REMORA in a straightforward manner due to its modular design.

This design allows the automatic discovery of new functionality by means of a configuration file that is read during REMORA startup. This configuration file contains a list of module names to activate. Typically, each of these modules collects a different type of statistic. The tool will read each line and will load a script file with the same name specified in the configuration file. The script file needs to implement at least four functions: initialization, data collection, post-processing, and finalization.

Developers of new modules may define other functions, but they will have to be called from one of the four required functions. The initialization, post-processing and finalization functions are called only once during the execution of the tool, while the data collection method will be called by REMORA on each time step.

If the default configuration file includes, for example, a line with the text *cpu*. This line indicates there will be a script with the file name *cpu*. This script is responsible for collecting the data regarding CPU utilization on each one of the nodes used by the application. The script defines the following functions:

- init_module_cpu
- collect_data_cpu
- process_data_cpu
- plot_data_cpu
- monitor_data_cpu
- finalize_module_cpu

Each of these functions takes three arguments:

- Name of the node where the function is running (`$REMORA_NODE`)
- Full path where the output will be stored (`$REMORA_OUTDIR`)
- Full path to an optional temporary storage location (`$REMORA_TMPDIR`)

```
$REMORA_NODE $REMORA_OUTDIR $REMORA_TMPDIR
```

Because it isn't necessary to understand the program architeture and calling sequence to create a module which initializes, collects and plots data from a resource, it is simple to extend REMORAS's functionality to to include additional resources. All that is necessary is to create a to create a new module file (named appropriately after the resource, here named *newres_module*) with the following functions:

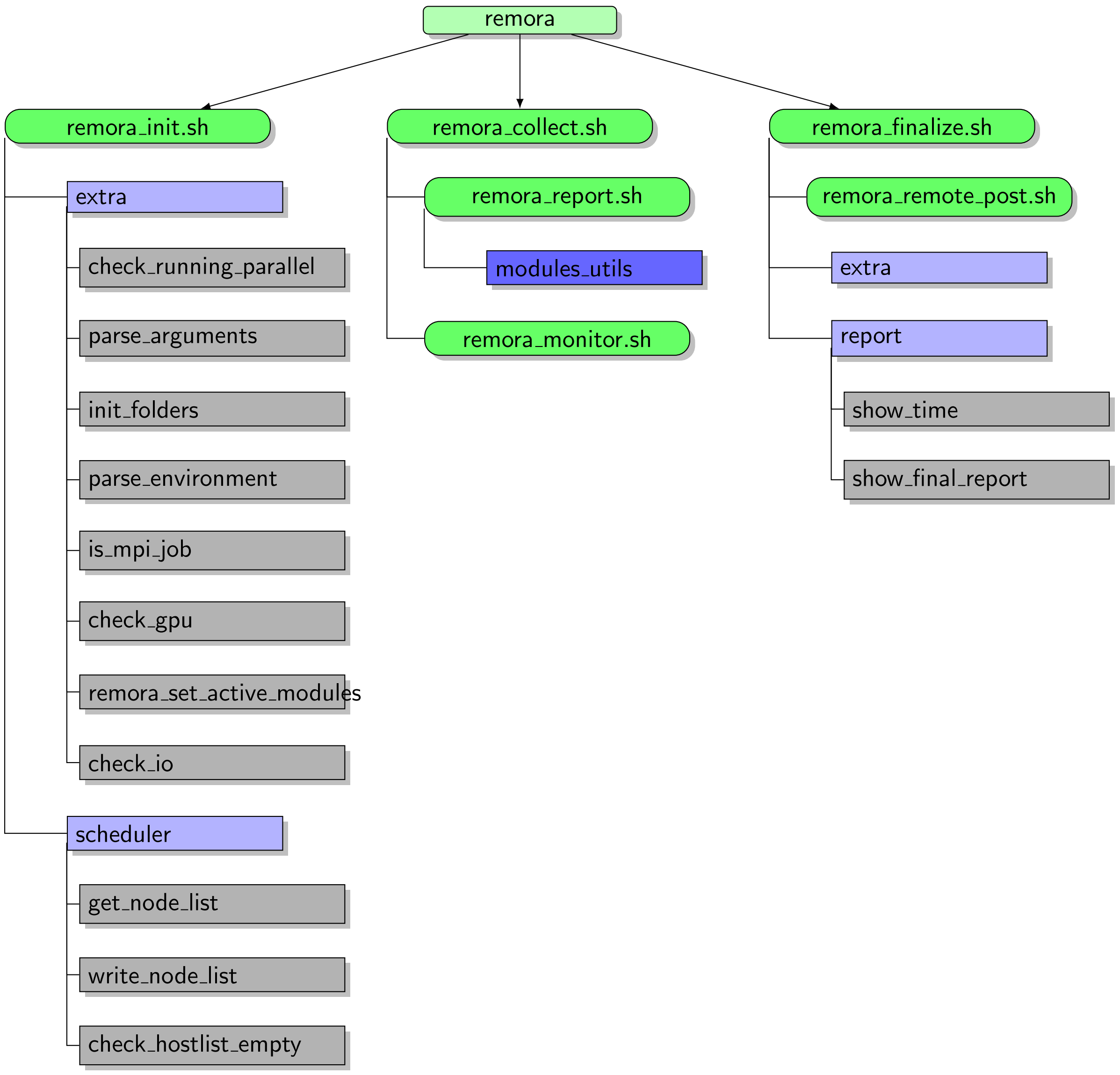
- init_module_newres_module
- collect_data_newres_module

- plot_data_newres_module
- monitor_data_newres_module
- finalize_module_newres_module

Use any one of the other modules as a template. In order to activate *newres_module*, include a line in the configuration file with the text *newres_module*. Functions may be left empty (e.g. monitor_data_newres_module). Typically, the majority of a module's functionality will be implemented in the data collection module.

This design also makes it straightforward to deactivate existing modules. Simply by removing a module name in the configuration file, the module becomes disabled during a run.

3.3 Code Structure



4 Expanding REMORA's Functionality

Since version 1.6.0, REMORA allows system administrators to easily change the functionality provided by REMORA and extend its capabilities. The configuration file `src\config\modules` defines the modules used by REMORA. Each module provides a specific capability in terms of data collection. By default, these are the modules available in REMORA:

- cpu
- memory
- numa
- lustre
- lnet
- dvs
- ib
- gpu
- network
- impi
- temperature
- power

These modules are currently specified in the configuration file `config/modules`. At runtime, REMORA will read this file and execute all the modules listed on it.

The remora distribution uses modules that require no elevated privileges in the system. If you feel safe in using setuid on your own module, then you can easily extend `remora` to support to access root protected data or utilities.

When installing REMORA system-wide, modify the configuration file src/config/modules to use only modules which can access resources on the system, before installing it. For a user-space installation the bin/config/modules file can be modified at any time.

4.1 Structure of Modules

All the modules are located in the `src/modules/` folder (or the `bin/modules/` in the install directory). They are bash scripts that implement, at least, the following functions:

- `init_module_modulename()`: this function is called only once to initialize the environment or files that are required by each specific module.

- `collect_data_modulename()`: main function of the module. This is the function called each `REMORA_PERIOD` seconds. The metrics collected will depend upon the resource usage being evaluated by the function. This function can include some data manipulation.
- `process_data_modulename()`: currently unused.
- `plot_data_modulename()`: generate interactive Google Charts plots in html file.
- `monitor_data_modulename()`: real time post-processing of captured data
- `finalize_module_modulename()`: this function is also called only once, when the application wrapped by `remora` has finished, or `snaphot` with plotting is executed. Any heavy postprocessing method can go in this function.

Each of these functions takes exactly three arguments:

- Name of the node where the function is running (`$REMORA_NODE`)
- Full path where the output will be stored (`$REMORA_OUTDIR`)
- Full path to an optional temporary storage location (`$REMORA_TMPDIR`)

```
$REMORA_NODE $REMORA_OUTDIR $REMORA_TMPDIR
```

The `modulename` is exactly the same name specified in the configuration file and also the filename of the bash script. This means that, if we have a module called `cpu` in the configuration file, there will be a file named `cpu` in the modules folder. And, inside that file, the required functions will be called:

- `init_module_cpu()`
- `collect_data_cpu()`
- `process_data_cpu()`
- `plot_data_cpu()`
- `monitor_data_cpu()`
- `finalize_module_cpu()`

4.2 Creating a New Module

REMORA has been designed so that expanding its functionality is very simple. If you want to create a new module, let's call it `newmodule`, you will need to follow these steps:

1. Add a new line to the configuration line with the string `newmodule`, followed by one of the RESOURCE directory names.
2. Create a new file in the `src/modules` folder called `newmodule`.
3. Inside this new file, define the following functions:
   - `init_module_newmodule()`
   - `collect_data_newmodule()`

- `process_data_newmodule()`
- `plot_data_newmodule()`
- `monitor_data_newmodule()`
- `finalize_module_newmodule()`

4. Implement the functionality that `newmodule` requires in those functions.

At runtime, REMORA will find the new module specified in the configuration file and call the appropriate functions at runtime.

It is very important to make sure that all the functions contain exactly the same name of the module included in the configuration file and used as filename for the module, since the remora framework will use that name to automate several of the processing stages.

It is possible to use an existing system wide installation to check a new module. Two environment variables can be used:

- `REMORA_CONFIG_PATH`: path to the configuration file (named config). If defined, REMORA will only use the modules specified on that file, not the previously existing ones.
- `REMORA_MODULE_PATH`: path to the modules. By default, REMORA looks in `REMORA_BIN/modules`, but users can create their own modules and store them somewhere else. If defined, system modules will not be used.

Bibliography

[1] C. Rosales, A. Gómez-Iglesias, A. Predoehl. "REMORA: a Resource Monitoring Tool for Everyone". HUST2015 November 15-20, 2015, Austin, TX, USA. DOI: 10.1145/2834996.2834999