

REMORA v1.5.0

REsource MOnitoring for Remote Applications
Document Revision 1.0
February 10, 2016

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

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

Copyright 2016 The University of Texas at Austin.

Abstract

REMORA is a modular tool to monitor runtime resource utilization in HPC environments. As of version 1.5.0 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 file system
- NUMA properties
- Network topology

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.

Our intention is to create a simple runtime resource monitoring tool that provides both simple to understand high level information to the user, as well as detailed statistics for in-depth analysis. We welcome all feedback. Feedback that includes suggestions for improvement in the usability, reliability, and accuracy of this tool is particularly welcome.

Version 1.5.0 is a new release that adds a modular design that allows to easily configure the functionality available on each site as well as a simple addition of new functionality to the tool. In this release REMORA has also become independent of XLTOP, so Lustre IO data may be captured without the need to install and run third party tools.

Remora is an open-source project. Funding to keep researchers working on Remora depends on the value of this tool to the scientific community. We would appreciate if you could include the following citation in your scientific articles:

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]

Contents

1	Installation	2
2	Design and Implementation	3
2.1	Statistics Collected	3
2.2	Modular Design	5
2.3	Code Structure	7
3	Using Remora	8
3.1	Collecting Data	8
3.2	Post-Processing	8
3.3	Execution Customization	8
3.4	Expanding Remora's Functionality	9
3.4.1	Structure of Modules	9
3.4.2	Creating a New Module	10
	References	10

1 Installation

Remora is simple to install. In order to use all of its features you will need GNU Make and a C compiler. If testing the Xeon Phi you will also need a compiler with support for compilation of native code for the Xeon Phi - currently limited to Intel 13+ compilers.

First of all obtain the latest version tarball, currently `remora-1.4.tar.gz` ¹, and expand it in a convenient location in your system:

```
tar xzvf ./remora-1.5.0.tar.gz
```

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` and `extra`. Change directory to the top level of 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	.
PHI_BUILD	Enable (1) or disable (0) Xeon Phi build	0

The 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 3.4 for more information about how to expand Remora's functionality).

Once these variables are set the tool can be installed by using:

```
./install.sh
```

The executable tests will be placed in the `remora/bin` directory unless the `REMORA_DIR=/install/dir/location/` has been modified at the top of the installation script.

¹<https://github.com/TACC/remora/archive/v1.5.0.tar.gz>

2 Design and Implementation

Remora is designed with ease of use in mind. Our goal is to create a usage model that users can take advantage of with minimal effort. 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 our 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 scripts in the same job. In that case, the user will have to put all the commands regarding the execution in a shell script (i.e. a shell script called `myjob`). Then, all the user has to do is to call the script as follows `remora ./myjob`. The tool can be used in a batch script or interactively in the command prompt.

Although we mention modules in the previous paragraph, modules are only used as a way to make easier for users to use Remora. It is not required to install Remora as a module.

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.

As previously stated, REMORA allows users to monitor the statistics as they are being generated. All the collected data is written to files accessible by the users in real time.

2.1 Statistics Collected

REMORA collects a set of statistics that are useful in many different scenarios when profiling an application. The data collected by REMORA consists of:

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

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 3.3](#)):

- 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 1.5.0. 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.

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 sections can be revised for possible optimization.

2.2 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`
- `finalize_module_cpu`

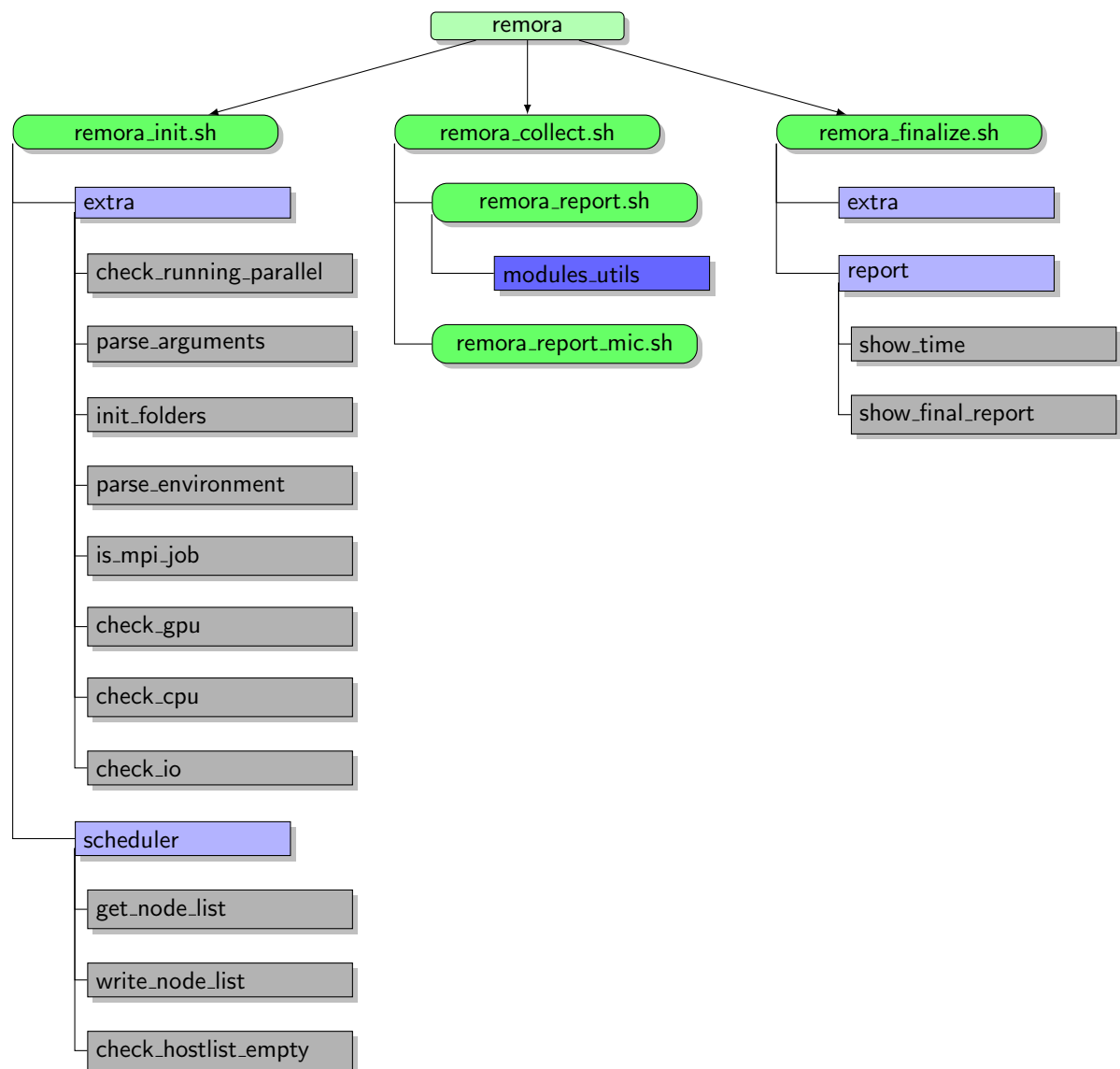
Developing a new module following this model is a task that can be performed without understanding the infrastructure that calls the different modules. This makes it simple to extend REMORAS's functionality to support additional data collection and processing. If a new module is being developed with name *new_module*, a file named *new_module* will need to be created with at least the following functions:

- `init_module_new_module`
- `collect_data_new_module`
- `process_data_new_module`
- `finalize_module_new_module`

In order to activate *new_module*, the configuration file will need to include a line with the text *new_module*. Functions may be left empty. 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 not including a module name in the configuration file will de-activate the module at runtime

2.3 Code Structure



3 Using Remora

3.1 Collecting Data

After making sure the `remora bin` directory is in your `PATH` and the `remora lib` directory is in your `LD_LIBRARY_PATH`

3.2 Post-Processing

All the data is collected in a set of files with the statistics organized in columns. Users can take those files and run any postprocessing tool that they develop. However, for simplicity, REMORA already provides a plotting script called `remora_post` that takes all the statistics generated during collection and generates a number of plots. These plots show the most relevant information previously collected and represent a visual alternative while analyzing the results. This is a Python script that requires Numpy, matplotlib and blockdiag. 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 the id of the job to be analyzed as argument.

3.3 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 statistics 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 three environment variables:

- **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_MODE**: this variable accepts two possible values (FULL and BASIC). The FULL mode runs all the tests that the tool allows. The BASIC mode only reports memory and cpu usage. This is the recommended mode when the users now that the application of interest does not create problems in the distributed file system.
- **REMORA_VERBOSE**: Enable (1) or disable (0) verbose mode. Default is disabled.

3.4 Expanding Remora's Functionality

Since version 1.5.0, 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`
- `ib`
- `gpu`
- `network`

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

If you are installing Remora system-wide, you can modify the configuration file before installing it. If you have Remora installed in your user account, you can modify this file at any time.

3.4.1 Structure of Modules

All the modules are located in the `src/modules/` folder. They are all bash scripts that have to 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. Depending on the module, different statistics will be collected in this function. This function can include some postprocessing.
- `process_data_modulename()`: currently unused.
- `finalize_module_modulename()`: this function is also called only once, when the application of interest has finished. Any heavy postprocessing method can go in this function.

All these functions take two, and only two, arguments:

- Name of the compute node where the function is going to run.
- Full path to the output folder.

The `module_name` 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()`
- `finalize_module_cpu()`

3.4.2 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`.
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()`
 - `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 different 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.

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](https://doi.org/10.1145/2834996.2834999)