Altair - Mistral User Manual

Version 2025.2.0

Contents

1 Introduction

Mistral is a tool used to report on and resolve I/O performance issues when running complex Linux applications on high performance compute clusters. Mistral was originally developed by Ellexus, now an Altair company.

Mistral allows you to monitor application I/O patterns in real time, and log undesirable behaviour.

This document is a reference manual. If you would like step-by-step instructions for installing and configuring Mistral with default settings then please consult the Mistral Quick Start Guide.

2 Overview

See the Mistral Quick Start Guide for tips on getting started with Mistral, and the `docs/samples` directory for sample Mistral launch scripts and scripts for integrating with various schedulers.

2.1 Download

Go to the Altair One page and register for an account. You can then log in and download Mistral.

We have 3 formats of package, ".tar.gz", ".deb" and ".rpm". Select whichever is easiest for you to work with. Mistral needs to be installed in the same location across all the machines that you wish monitor. This is most easily achieved by installing it to a shared file system, if this is selected then there is no need to install Mistral on each individual host.

If Mistral is not installed on a shared file system, then it is important to ensure that `etc/mistral_config.yaml` is kept consistent across the cluster.

Please make sure that you use the appropriate version of Mistral (`i386`, `aarch64` or `x86_64`) for the machines you want to run it on.

2.1.1 Tar file

To install, just untar the product on any filesystem which is accessible from all the hosts that will run workloads.

2.1.2 Debian package

Install using your package manager. This will install Mistral to `/opt/altair/mistral_2025.2.0_x86_64` by default.

e.g.

```
$ sudo dpkg -i mistral_2025.2.0-0_amd64.deb
```

To install to a different location use the `--instdir=dir` argument to `dpkg`.

2.1.3 RPM package

Install using your package manager. This will install Mistral to `/opt/altair/mistral_2025.2.0_x86_64` by default.

e.g.

```
$ sudo rpm --install mistral-2025.2.0-0.x86_64.rpm
```

To install in a different location use the `--prefix=<path>` argument to `rpm`.

2.2 Licensing

Mistral requires a license. Please contact Altair if you do not already have a valid Mistral license. Most customers will use an Altair floating or node-locked license. The environment variable `ALTAIR_LICENSE_PATH` must indicate your Altair license file or license servers (as `<port1>@<server1>:<port2>@<server2>` and so on).

```
export ALTAIR_LICENSE_PATH="/opt/licenses/mistral_lic.dat:6200@licserver"
```

If using a legacy Ellexus license file, the environment variable `MISTRAL_LICENSE` must be set to the path of your license - either a specific license file or a directory which contains one or more license files.

```
export MISTRAL_LICENSE=<path to license file>
```

By default Mistral writes errors to `MISTRAL_ERR_LOG` or `syslog` if this is not set. When setting up Mistral it can be useful to get license error messages output to `stderr` as well as the normal error log location. To do this export the `MISTRAL_CHECK=1` environment variable before starting Mistral (or running `start-mistral-monitor`). This behaviour is not normally desirable when run in production - as it can change the job output.

```
    $ export MISTRAL_CHECK=1
    $ export ALTAIR_LICENSE_PATH=/opt/altair/mistral/invalid
    $ ./bin/mistral.sh ls /dev/null

/dev/null
Altair licensing failure: Couldn't get license: 9
(Altair License Manager: License error
Feature(s): MISTRAL-CORES
Error Code: 9
Error Description:
Feature not found

License Path: /opt/altair/mistral/invalid
)
License is invalid. ALTAIR_LICENSE_PATH="/opt/altair/mistral/invalid",
MISTRAL_LICENSE="(null)".
```

Set the environment variable `MISTRAL_LOG`, `MISTRAL_ERR_LOG` and `ELLEXUS_LOG_LEVEL` in order to get debugging information in `MISTRAL_ERR_LOG`.

```
export MISTRAL_LOG=<path to mistral log file>
export MISTRAL_ERR_LOG=<path to mistral error log file>
export ELLEXUS_LOG_LEVEL=license:MAJOR
```

Following is the list of warning and errors in `MISTRAL_ERR_LOG` file for different scenarios::

1. If using both `MISTRAL_LICENSE` and `ALTAIR_LICENSE_PATH`, then the warning message will be added in the mistral error log file as **Don't set both ALTAIR_LICENSE_PATH and MISTRAL_LICENSE**.

2. If the value is empty in both `MISTRAL_LICENSE` and `ALTAIR_LICENSE_PATH`, then the warning message will be added in the mistral error log file as **The value is empty in ALTAIR_LICENSE_PATH and MISTRAL_LICENSE**.

3. If not set both `MISTRAL_LICENSE` and `ALTAIR_LICENSE_PATH`, then the warning message will be added in the mistral error log file as **Set either ALTAIR_LICENSE_PATH or MISTRAL_LICENSE**.

4. If using `MISTRAL_LICENSE` but there is a problem with it, then the either of the following warning message to provide more details about it.

    1. **Failed to read license: MISTRAL_LICENSE="<license path>".**,
    2. **No license found, please check MISTRAL_LICENSE="<license path>".**
    3. **Not a valid Ellexus legacy license, event <event number>**

5. If using `ALTAIR_LICENSE_PATH` but there is a problem with it, then the failure message will provide more details as **Altair licensing failure: <failure message>**.

6. If the license is invalid, then Mistral will not monitor the appilcation and log the error message as **License is invalid. ALTAIR_LICENSE_PATH="<license path>", MISTRAL_LICENSE="<license path>"**.

2.3 Data pipeline

Mistral can be run on a single application, but to visualize data for multiple applications you will need to set up a database, to collect the data from each application, and a dashboard, to present visualisations of the data in the database. Mistral can push data to various databases such as Elasticsearch and Splunk.

See the Plug-ins section for more information on connecting Mistral to a database.

For customers who do not already have a log-aggregation infrastructure in place we recommend starting with Elasticsearch, and the Grafana dashboard. See the Mistral Quick Start Guide for detailed instructions on setting up Elasticsearch and Grafana.

2.4 Configuration

Mistral behaviour is controlled by a number of environment variables and a configuration file. See the Configuring Mistral section for more details, and the `docs/samples/mistral_start_reference.sh` for an example script setting many of these variables.

2.5 Scheduler integration

In order to deploy Mistral across your cluster, you may wish to integrate it with a scheduler, workflow manager, or other orchestration tool. The `docs/samples` directory contains scripts and configuration files for many such integrations. See the Scheduler integration section for detailed instructions.

3 Monitoring an application

Once Mistral has been configured it can be run using the `mistral.sh` script available in the bin directory of the installation. To monitor an application you just type `mistral.sh` followed by your command and arguments. For example:

```
$ ./bin/mistral.sh ls -l $HOME
```

By default any error messages produced by Mistral will be written to the system log (`syslog`). Any errors that prevent the job running as expected, such as a malformed command line, will be output to `stderr`.

This behaviour can be changed by the following command line options.

`--errlog=<filename>`
`-l=<filename>`

Record Mistral error messages in the specified file. If this option is not set, errors will be written to the system log (`syslog`).

`-q`

Quiet mode. Send all error messages, regardless of severity, to the error log. Command line options are processed in order, therefore this option must be specified first to ensure any errors parsing command line options are sent to the error log.

In addition to monitoring single applications in this way, you may wish to integrate Mistral with your general scheduling or job management systems, to monitor performance across your cluster. See Scheduler integration.

4 Configuring Mistral

Mistral is configured using environment variables and a configuration file. This section describes the principal variables you may wish to set.

4.1 License

As described above, the `ALTAIR_LICENSE_PATH` environment variable must indicate your Altair license file or license servers (as `<port1>@<server1>:<port2>@<server2>` and so on). If you have a legacy Ellexus license, `MISTRAL_LICENSE` should give its path instead.

If using Mistral with other Altair products, such as HyperWorks, you can specify multiple sources of licenses e.g.

```
export ALTAIR_LICENSE_PATH="/tools/altair/licenses/mistral_lic.dat:6200@licserver"
```

4.2 Log locations

The following environment variable configures the location Mistral uses for its log file.

`MISTRAL_LOG`	The path of the file in which Mistral will log I/O activity.

This may also be set by the relavent configuration option, with the environment variable taking precedence. See Configuration file.

If the log path contains %h at any point, the hostname is substituted for it. If a log path *doesn't* contain %h, the hostname is inserted, either at the end of the path or before a final `.log` if it exists. This helps to avoid confusion or corruption if several Mistral jobs are running at the same time, by ensuring that log filenames always contain the hostname.

4.3 Plug-in configuration

To use a plug-in to communicate Mistral data to your database, simply add a plugin configuration into your mistral configuration file. See the `docs/samples/` folder for example configurations to include in your mistral configuration file. See Configuration file.

Plug-ins are available in the `plugins` directory, each plug-in has it's own documentation in this folder.

4.4 Working directory location

Mistral requires a directory, to store its working data during operation. This should be on high-performance storage, local to each host if possible, and writable by user processes. Mistral will create per-job directories within this top-level directory. To specify a suitable location, set the variable `MISTRAL_OUTPUT_DIRECTORY`. If you do not specify this, Mistral will create a directory in `/tmp`.

4.5 Error log

If Mistral encounters any internal errors, it will write messages to a log file. You can specify the location of this log file with the `MISTRAL_ERR_LOG` environment variable or in the configuration file. If this is not set then Mistral will write any error messages to `syslog`.

4.6 Configuration file

The `MISTRAL_CONFIG` variable contains the path to a configuration file. If this variable isn't set, the file `mistral_config.yaml` in the `etc` directory is used instead. That file as distributed sets the default configuration as described below.

A configuration file is a YAML file with keys and values controlling various aspects of Mistral behaviour including what will be written to the log file or plugin. The following keys and values are supported.

Key	Values	Default	Description
`output`	/path/to/file	/tmp/mistral-%h.log	Where to output Mistral reports if configured to. See `plugin` option as well. Can be overriden with `MISTRAL_LOG`
`error`	/path/to/file	syslog	Where to output Mistral errors. Special value `syslog` goes to the system log. Can be overriden with `MISTRAL_ERR_LOG`
`license`			
`path`	string		Sets `ALTAIR_LICENSE_PATH` if it isn't set, or prepends to it if it is
`units`	yes no	no	Whether license is an Altair Units license or not (in which case it will be a feature license)
`timeframe`	number (s|m|h)	10s	Time interval between reports, or "no" for no periodic reports
`totals`	yes no	yes	Report total statistics at the end of the job. This includes duration reports
`duration-sampling`			
`sample`	number	0	If set to a positive number *n*, Mistral will measure the duration of about 1/*n* I/O operations
`limit`	number	0	If set to a positive number *n*, Mistral will limit the number of duration measurements to *n*/second
`wait_pid`	mistral first parent group parent-group	mistral	Which process Mistral should wait for before exiting (see below)
`vars`	["ENV_VAR"]		A sequence of environment variables to report on.
`read:`			
`enabled`	yes no	yes	Report read statistics
`duration`	yes no	no	Report the duration of read operations
`sized`	small-medium-large combined binary	small-medium-large	Break down read statistics by size (see below)
`write:`			

Key	Values	Default	Description
enabled	yes no	yes	Report write statistics
duration	yes no	no	Report the duration of write operations
sized	small-medium-large combined binary	small-medium-large	Break down write statistics by size (see below)
seek:			
enabled	yes no	yes	Report seek statistics
metadata:			
enabled	combined separate both no	combined	Report statistics for metadata operations (open, access, create, delete, fschange, connect and accept). combined: aggregated statistics; separate: separate statistics for each call type; both: both aggregated and separate statistics. no: do not report metadata statistics.
duration	yes no	no	Report the duration of metadata operations
network:			
enabled	yes no	no	Report network I/O in addition to file I/O
cpu-memory:			
enabled	all job host no	all	Report CPU/memory usage. job: report usage per-job; host: report usage per-host; all: report both per-host and per-job usage; no: do not report CPU/memory usage.
gpu:			
enabled	yes no	no	Report GPU usage
nvidia-library-path	/path/to/libnvidia-ml.so	default	Path to NVidia Management Library. Special value “default” uses system library paths.
plugin:			
path	/path/to/file		Path to plug-in executable, either absolute or relative to <mistral_install>
interval	number (s|m|h)	5s	Time interval between calls to plug-in
file-output	yes no	no	Output to the log file specified by MISTRAL_LOG as well.
options:	option: value		A list of option: value pairs representing options to pass to the plug-in executable. See sample etc/mistral_config.yaml for format
switches:	[“switch”]		A sequence of switches to pass to the plug-in executable.

4.6.1 Wait Process

This specifies which process Mistral should watch before exiting. Mistral will, by default, attempt to use information about your environment or job-scheduler and use this to determine when it should stop monitoring your job. However, if you are using a non-supported environment or job-scheduler, then this may cause Mistral to continue monitoring even after the job you wish to be traced has exited.

The values for the option `wait_pid` override the default behaviour as follows:

- `first` will cause Mistral to wait for the first process to run in a job, useful for if you have a daemonised process that starts jobs and you do not wish to wait for this.
- `parent` will cause Mistral to wait for the parent of the first process in a job, useful for when jobs are started by a short-lived scheduling process, such as a terminal.
- `group` will cause Mistral to wait for the process group leader of the first process to run in a job, useful for data transfer jobs where data is continuously piped through various processes.
- `parent-group` as above but the parent processes' process group leader
- `mistral` uses default behaviour

Since Mistral monitors child processes, if the specified process has exited, Mistral will not exit until all child processes have exited.

4.6.2 Size bins

The `read.sized` and `write.sized` configuration keys are used to control the reporting of read/write operations broken down into size categories. By default, such operations are divided into "small", "medium", and "large" operations, where "small" operations are up to 32 kibibytes (32 * 1024 bytes), "large" operations are over 128 mebibytes (128 * 1024 * 1024 bytes), and "medium" operations are between 32 KiB and 128 MiB. If `read.sized` or `write.sized` is set to `combined`, all read or write operations are aggregated together. If it is set to `binary` then such operations are divided into power-of-two "bins": 0 bytes, 1 byte, 2-3 bytes, 4-7 bytes, 8-15 bytes, and so on.

Naturally, size bins without any operations are omitted from the report.

4.7 Miscellaneous configuration

4.7.1 Volumes

Mistral can be configured to treat an arbitrary directory as a mount point by creating a text file with a series of absolute paths, one per line, and setting the environment variable `MISTRAL_VOLUMES` to point to this file.

4.7.2 Duration sampling

Duration sampling is controlled by two entries in the configuration file.

If `duration-sampling.sample-factor` is set to a positive number, *n*, Mistral will measure the duration of an I/O operation with probabilty 1/*n*. So a value of value of 1 means that Mistral will measure the duration of all I/O calls, while a value of 10 means that about 1 in 10 I/O calls will be measured. If set to zero Mistral will choose a sampling factor based on how long it takes to get the current time. On many systems this overhead is low, so Mistral can measure all durations. But on other systems it makes more sense to only measure a small fraction of the I/O calls.

If `duration-sampling.sample-limit` is set to a positive number then Mistral will measure the duration of no more than that number of I/O calls per second. If set to zero then in most cases Mistral won't impose a limit. If however both `sample-factor` and `sample-limit` are set to zero, then when running on a system where it takes more than 1 microsecond to get the current time, a limit of 1000 duration measurements per second will be used.

On most modern systems it only takes a few tens of nanoseconds to get the current time and Mistral can easily measure the duration of all I/O calls. Some older systems, especially some older virtual machines, can take as much as 1 microsecond to get the current time. Measuring all durations on those machines could noticeably slow down the job. If you set both `duration-sampling.sample-factor` and `duration-sampling.sample-limit` to zero then Mistral will measure all the durations on a fast machine, but will reduce the number of measurements on slower machines.

If you prefer to define your own parameters then setting `duration-sampling.sample-factor` to 10 and `duration-sampling.sample-limit` to 1000 will often give you usable data. However, setting an upper limit on the number of measurements per second can easily distort the timing statistics, especially if the operating system varies the CPU frequency. If you see inconsistent results, such as the I/O time for a single threaded program being greater than it's run time, then you should increase the sampling factor and remove any upper limit. For example, you might set `duration-sampling.sample-factor` to 20 and set `duration-sampling.sample-limit` to zero.

Note that Mistral estimates total duration for a call type based on the duration of the sampled operations. So if there were 20000 read operations in a single time frame of which 1000 were sampled, Mistral would report a value which is twenty times the sum of the measured latencies.

Duration sampling reduces run-time overhead, but some things which seem intuitively obvious will no longer be true. Suppose you configured Mistral to report the total duration of reads, with small/medium/large size bins. If you measure all durations the value reported for "all reads" will be equal to the sum of the duration of reads for each of the size bins. But if duration sampling is used these numbers may not add up exactly. Each individual value reported will be the best estimate of that duration, but the best estimate of the sum is not necessarily the sum of the best estimates of its parts.

In older versions of Mistral duration sampling was controlled by environment variables. Those variables will still be honoured, but non-zero values set in the config file will take priority over them.

4.7.3 Other environment variables

The following other environment variables for configuring Mistral are described elsewhere in this manual:

- `MISTRAL_PLUGIN_EXIT_TIMEOUT`, see Plug-ins;
- `MISTRAL_CONTAINER_BIND_PATH`, see Singularity and Apptainer;
- `MISTRAL_NOTRACE_PROGRAMS`, see Turning Mistral on and off;
- `MISTRAL_BYPASS_PROGRAMS`, see Bypassing specific programs with Mistral;
- `MISTRAL_BYPASS_DISABLE`, see Bypassing specific programs with Mistral;
- `MISTRAL_PROFILE_SMALL_IO`, see Summary Record;
- `MISTRAL_DISABLED_SUBMITTERS` see Disabling specific function wrappers with Mistral.

In addition to these, there are additional environment variable settings which may be recommended to you by Altair Support.

4.7.4 LD_PRELOAD

This is not a configuration setting. Mistral uses the `LD_PRELOAD` variable to add itself to user processes. If you are writing your own Mistral starter script, it must set this variable in addition to those listed above. Please consult the `docs/samples/mistral_start_reference.sh` script for details.

5 Plug-ins

A plug-in is used to process log entries generated by the Mistral application. All log entries are sent to the output plug-in.

On start up Mistral will check the configuration file for a plug-in configuration. If a plug-in configuration is not defined Mistral will default to recording data to disk as described above. In addition if a plug-in performs an unclean exit during a job Mistral will revert to recording data to a log file.

The `docs/samples` directory contains sample plug-in configuration files for several different databases.

When using a plug-in, at the end of a job Mistral will wait for a short time, by default 30 seconds, for the plug-in to exit in order to help prevent data loss. If any plug-in process is still active at the end of this timeout, it will be killed. The timeout can be altered by setting the environment variable `MISTRAL_PLUGIN_EXIT_TIMEOUT` to an integer value between 0 and 86400 that specifies the required time in seconds.

5.1 Data rate

When setting up a plug-in it makes sense to consider the rate at which Mistral can be configured to output data. The amount of data output is dependent on your configuration and job behaviour: jobs which access many mountpoints will always produce more log data.

For the default Mistral configuration (small/medium/large size bins, duration metering off, metadata reported together, no network metering), we recommend allowing for around 20 kilobytes of data per job per timeframe. For more detailed logging (power-of-two size bins, duration metering on, separate metadata metering), we recommend allowing 200 kilobytes of data per job per timeframe.

6 Scheduler Integration

This section describes how to integrate Mistral with various schedulers and workflow managers. It assumes some familiarity with configuration and administration of your scheduler environment.

6.1 Altair PBS Professional

You should create a PBS hook script that sets the required environment variables and starts/stops Mistral. A sample hook is provided for you to copy and edit:

`<mistral_install>/docs/samples/pbs/mistral_hook.py`

Note that hooks run for jobs on every queue. The sample hook is written so that it only has any effect on queues with names beginning “mistral”. You may want to modify this condition depending on your queue names.

Run these commands to create a hook named “mistral” and import the hook script:

```
$ qmgr -c 'create hook mistral event="execjob_launch,execjob_end,execjob_epilogue"'
$ qmgr -c 'import hook mistral application/x-python default
 ↪  /path/to/mistral_hook.py'
```

Note: Every time the hook script is modified, it needs to be “imported” again using the same `import hook` command.

6.2 Altair Grid Engine version 2022.1 and above

Version 2022.1 and above of Altair Grid Engine includes Mistral integration. For earlier releases of Altair Grid Engine, see the next section.

To enable Mistral for a queue, use `qconf` to change the `execd_params` configuration item. See the manual page for `sge_conf(5)`. The following four parameters are mandatory:

- `AGE_MISTRAL_MODE`:
  - Set to `ALWAYS` to turn Mistral on for all jobs on this queue.
  - Set to `DEFAULT_ON` for Mistral to be enabled for all jobs on the queue unless the `AGE_MISTRAL` environment variable is set to `0` (with `qsub -v`).
  - Set to `IF_REQUESTED` for Mistral to be enabled for a job on the queue only when the `AGE_MISTRAL` environment variable is set to `1` (with `qsub -v`).
  - Set to `NEVER` (the default value) to prevent Mistral being enabled on the queue.
- `AGE_MISTRAL_INSTALL_PATH`: set to the full path of the directory where Mistral has been installed.
- `AGE_MISTRAL_LICENSE_PATH`: set to the full path of the license file for Mistral.
- `AGE_MISTRAL_ENV`: set to the full path for an environment file to be used for Mistral jobs. The file should contain `name=value` settings for Mistral environment variables such as `MISTRAL_CONFIG`, `MISTRAL_LOG`, and so on (see Configuring Mistral above for guidance on these environment variables). For a sample environment file, see `docs/samples/age-2022.1/mistral.env`.

This additional `execd_params` parameter is optional:

- `AGE_MISTRAL_LD_PRELOAD`: set to the full value for Grid Engine to use as `LD_PRELOAD` for the Mistral library. If not set when profiling is requested, Altair Grid Engine will use a value relative to the Mistral installation path: `AGE_MISTRAL_INSTALL_PATH/dryrun/$LIB/libdryrun.so` **Note that this path is the old path Mistral used to be packaged as. In order for Grid Engine to find this, please run the `compatibility-symlinks.sh` script found in the sbin folder of your Mistral install with the `create` parameter. This will create symlinks to simulate the old Mistral directory structure.**

Note: if Altair Breeze is also enabled for a job, then only Breeze profiling will happen.

Breeze can run upon the mistral node:

From Breeze and Mistral 2024.2.0, Breeze tracing and profiling can run along with Mistral as follows.

```
# Directly with mistral.sh script
~/mistral_latest/bin/mistral.sh ~/breeze_latest/bin/trace-program.sh     \
    -f ~/trace/trace.out/ --relocate ~/trace/relocate/                  \
    /bin/ls

# In AGE cluster
qsub -q all.q -v AGE_MISTRAL=1 -cwd -t 1-2 -b y         \
    ~/breeze_latest/bin/trace-program.sh                \
    -f ~/trace/trace.out/ --relocate ~/trace/relocate/  \
    /bin/ls
```

Note: If the job run Breeze and Mistral together, then the durations recorded by Mistral will include the overhead of Breeze due to the simultaneous execution of them.

6.3 Altair Grid Engine version 8.x

Older versions of Altair Grid Engine do not include Mistral integration. You should copy the example prolog, starter, and epilog scripts for Altair Grid Engine:

```
<mistral_install>/docs/samples/age-8.x/mistral_prolog.sh
<mistral_install>/docs/samples/age-8.x/mistral_starter.sh
<mistral_install>/docs/samples/age-8.x/mistral_epilog.sh
```

Copy and edit the environment script, setting environment variables appropriately for your systems.

`<mistral_install>/docs/samples/age-8.x/mistral_env.sh`

These scripts should all be saved together in a directory accessible to all execution nodes.

Then modify your queue configurations to add `prolog`, `starter_method` and `epilog` settings pointing to the scripts created above. For example if the scripts above has been saved in `/apps/altair/mistral/`, in order to add it to an existing queue “`mistral.q`”, type the command:

```
$ qconf -mq mistral.q
```

This will launch the default editor. Find the settings for `prolog`, `starter_method` and `epilog`, which are typically set to `NONE`, and update them with paths to the new scripts. Save the configuration and exit the editor. For example the following snippet of queue configuration shows the appropriate setting to use the files described above.

```
prolog                /apps/altair/mistral/mistral_prolog.sh
epilog                /apps/altair/mistral/mistral_epilog.sh
starter_method        /apps/altair/mistral/mistral_starter.sh
```

It is important to note that the `starter_method` will not be invoked for `qsh`, `qlogin`, or `qrsh` acting as `rlogin` and as a result these jobs will not be wrapped by Mistral.

To check if the configuration has been successfully applied, the `qconf` command can be used with the `-sq` option to show the full queue configuration which will list the configured scripts:

```
$ qconf -sq mistral.q
qname                 mistral.q
hostlist              @allhosts
...
prolog                /apps/altair/mistral/mistral_prolog.sh
epilog                /apps/altair/mistral/mistral_epilog.sh
...
starter_method        /apps/altair/mistral/mistral_starter.sh
...
```

6.4 Altair Accelerator

You should create a special environment file called MISTRAL.start.tcl which sets all the environment variables for the job, by copying the example file:

`<mistral_install>/docs/samples/accelerator/MISTRAL.start.tcl`

Edit the settings in the file to suit your installation, and then save the resulting file as `MISTRAL.start.tcl` in an Accelerator environment directory.

e.g. `$VOV_ENV_DIR` or `$VOVDIR_LOCAL/environments`.

To use Mistral, submit a job with a command such as:

```
nc run -e SNAPSHOT+MISTRAL -- myJob
```

See also the Bypassing scheduler internal activity section for an example of using Mistral's "bypass" feature to disregard I/O caused by Accelerator's internal programs.

6.5 Altair Flowtracer

You should create a special environment file called MISTRAL.start.tcl which sets all the environment variables for the job, by copying the example file:

`<mistral_install>/docs/samples/flowtracer/MISTRAL.start.tcl`

Edit the settings in the file to suit your installation, and then save the resulting file as `MISTRAL.start.tcl` in a Flowtracer environment directory.

e.g. `$VOV_ENV_DIR` or `$VOVDIR_LOCAL/environments`.

To use Mistral, add `+MISTRAL` to the environment used for a job.

See also the Bypassing scheduler internal activity section for an example of using Mistral's "bypass" feature to disregard I/O caused by Flowtracer's internal programs.

6.6 IBM Spectrum LSF

You should copy and edit the sample LSF Mistral starter script:

`<mistral_install>/docs/samples/lsf/mistral_starter.sh`

This script should be saved in an area accessible to all execution nodes.

For each queue that is required to automatically wrap jobs with Mistral, add a `JOB_STARTER` setting that re-writes the command to launch the submitted job using the script created above. For example if the script above has been saved in `/apps/altair/mistral_starter.sh` the following code defines a simple queue that will use it to wrap all jobs with Mistral:

```
# Mistral job starter queue
Begin Queue
QUEUE_NAME = mistral
PRIORITY = 30
INTERACTIVE = NO
TASKLIMIT = 5
JOB_STARTER = . /apps/altair/mistral_starter.sh; %USRCMD
DESCRIPTION = For mistral demo
End Queue
```

Once the job starter configuration has been added the queues must be reconfigured by running the command:

```
$ badmin reconfig
```

To check if the configuration has been successfully applied to the queue the `bqueues` command can be used with the "`-l`" long format option which will list any job starter configured, e.g.

```
$ bqueues -l mistral

QUEUE: mistral
-- For mistral demo
PARAMETERS/STATISTICS
PRIO NICE STATUS MAX JL/U JL/P JL/H NJOBS PEND RUN SSUSP USUSP
RSV
30 0 Open:Active - - - - 0 0 0 0 0 0
Interval for a host to accept two jobs is 0 seconds
TASKLIMIT
5
SCHEDULING PARAMETERS
r15s r1m r15m ut pg io ls it tmp swp mem
loadSched - - - - - - - - - - -
```

```
loadStop - - - - - - - - - -
SCHEDULING POLICIES: NO_INTERACTIVE
USERS: all
HOSTS: all
JOB_STARTER: . /apps/altair/mistral_starter.sh; %USRCMD
```

6.7 Slurm

You should create a Slurm TaskProlog script that prints out the required environment variables and any default settings by copying and editing the sample file:

```
<mistral_install>/docs/samples/slurm/mistral_prolog.sh
```

This script should be saved in an area accessible to all execution nodes.

If slurm is set to use cgroups, it is also necessary to create a Slurm TaskEpilog script that signals to Mistral that the job is finished before the cgroup kills the task. Copy and edit the sample script:

```
<mistral_install>/docs/samples/slurm/mistral_epilog.sh
```

Configure Slurm to use the resulting TaskProlog and TaskEpilog scripts by adding the following lines in your slurm.conf file:

```
TaskProlog=/path/to/mistral/taskprolog.sh
```

```
TaskEpilog=/path/to/mistral/taskepilog.sh
```

Each execution host requires the same settings.

Finally, instruct all Slurm daemons to re-read the configuration file:

```
$ scontrol reconfigure
```

Now all jobs submitted with sbatch, srun and salloc commands use Mistral.

Note that the sample TaskProlog and TaskEpilog scripts only run Mistral on jobs in the “mistral” partition. Modify the scripts as necessary to suit your partition choices.

7 Container Support

7.1 Singularity and Apptainer

Mistral will monitor workloads in Singularity and Apptainer containers by default. This will add a number of bind paths to each singularity container, so that Mistral is able to read the configuration files and run the executables that it normally would. If these files are all in one area of your filesystem you can minimise the number of paths that are bound by setting the following environment variable to that path:

`MISTRAL_CONTAINER_BIND_PATH`

If you are writing your own Mistral starter script you may need the stand-alone program called `mistral-singularity-bind`. This program examines the Mistral configuration and outputs a value which can be used as the `SINGULARITY_BIND` or `APPTAINER_BIND` environment variable. Singularity and Apptainer mount the directories listed in these environment variables in the container. If you use a standard starter script then setting `SINGULARITY_BIND` or `APPTAINER_BIND` is handled automatically.

Note that Singularity by default attempts to mount both `/var/tmp` and `/tmp` to `/tmp` in the container. As such, if your `MISTRAL_LOG` variable points to somewhere in `/tmp`, then any data inside the container may be output to a different log file if you do not have a plug-in enabled. Likewise for `MISTRAL_ERR_LOG` should you have error logging enabled.

In order to work around changes to Singularity from v3.6.0 onwards we assume that the first application will be a 64-bit executable. If this is not the case then an error will be output and the executable will not be monitored. If this causes any problems please contact support.

7.1.1 PBS

If using Singularity with the PBS_hpc_container hook, you can ensure that Mistral will work by adding the Mistral installation directory to the `mount_paths` option in the configuration file.

To add this setting export the configuration file.

```
#qmgr -c "export pbshook PBS_hpc_container application/x-config default" \
      > container_config.json
```

You can then modify the `container_config.json` file setting e.g.

```
"mount_paths": ["/etc/passwd", "/etc/group", "/opt/altair/mistral"],
```

Then re-import the modified file:

```
#qmgr -c "import pbshook PBS_hpc_container application/x-config default" \
         "container_config.json"
```

For more information about container support in PBS please see the PBS Admin Guide.

7.2 Docker

Mistral does not currently monitor workloads in Docker containers by default – this feature is planned for a future release.

8 Mistral Healthcheck

If you are running Mistral on a small scale, for instance to test the functionality, it can sometimes be useful to log data to disk and then process the log file(s) that it produces.

There is a master script for doing this in the `bin` directory called `mistral_report.sh`; this creates separate CSV files for the different data series, GNUplot graphs and a HTML report.

8.1 mistral_report.sh

This script expects the path (or paths) to Mistral log files. Optionally you can also specify an output directory with the `-o` argument.

e.g.

```
$ bin/mistral_report.sh -o /tmp/mistral.out /tmp/job1.mistral.log
```

This will generate the HTML report, CSV files and GNUPlot graphs. To omit the CSV files and GNUPlot graphs supply the `-n` option.

8.2 Mistral Healthcheck reports

When the `bin/mistral_report.sh` script is run it will create the Healthcheck HTML file `mistral_report.html` and output the location of the file. This is the main report file and has links to all the other data. The other data is split by data series into different HTML files.

9 Turning Mistral on and off

9.1 Disabling Mistral for specific binaries

It is possible to disable Mistral monitoring for specific binaries, this is done by setting an environment variable `MISTRAL_NOTRACE_PROGRAMS`. This should contain a comma separated list of the full path, directory or binary name. If specifying a directory, ensure you use the absolute path.

When a matching binary is executed within the job, Mistral will be disabled for that binary and for any of its children or subprocesses. Mistral will not be loaded for that program so any undesired interaction will go away.

e.g.

```
export MISTRAL_NOTRACE_PROGRAMS="ssh,rsh"
```

Will match ssh and rsh, for example `/usr/bin/ssh`.

```
export MISTRAL_NOTRACE_PROGRAMS="/bin/,/usr/bin/"
```

Will match locally installed binaries such as `/bin/sh`.

```
export MISTRAL_NOTRACE_PROGRAMS="/home/user/dev/project/bin/failing_binary"
```

Will only match the binary that is causing you problems.

By default, Mistral won't monitor programs with any of these names: `blaunch, bsub, lsgrun, lsrun, pbsdsh, pbs_tmrsh, qsub, qrsh, sbatch, srun.` This is to prevent it from monitoring cluster scheduling infrastructure. You can remove a program from this list by including it in `MISTRAL_NOTRACE_PROGRAMS` preceded by a `-` sign:

```
export MISTRAL_NOTRACE_PROGRAMS="-srun"
```

These removal entries can be combined with additions in the obvious way:

```
export MISTRAL_NOTRACE_PROGRAMS="-srun,failing_binary,/usr/bin/ssh,-qsub"
```

The setting above will turn off profiling for failing_binary,/usr/bin/ssh and any subprocesses, and it will turn on profiling for srun and qsub.

9.2 Bypassing specific programs with Mistral

The `MISTRAL_BYPASS_PROGRAMS` environment variable can be used to disregard I/O which originates in a program with the possibility or re-enabling I/O monitoring for sub-proceses.

If the `MISTRAL_BYPASS_PROGRAMS` environment variable is set to a list of programs and directories, then any program which matches an entry in the list will be run in bypass mode. For example, `MISTRAL_BYPASS_PROGRAMS="emacs,/usr/local/"`, would ensure that `emacs` and any program or script in `/usr/local/`, or a sub-directory such as `/usr/local/bin`, would run in bypass mode. For directories, ensure that the entry is an absolute path, whereas binaries should be listed by name as shown in the example.

The `MISTRAL_BYPASS_DISABLE` variable can be used to turn tracing back on. It should contain the name of another environment variable, and turns tracing on when that specified variable is set.

For example, consider the following script:

```
#!/bin/bash
# hello_world.sh script
echo "hello world"
export TURN_TRACING_BACK_ON_PLEASE=1
ls
```

If Mistral is run with the following settings then I/O monitoring will be turned off for the script called `hello_world.sh` so the I/O generated from the echo call will not be captured. However `ls` is called with the bypass disabling env var set `TURN_TRACING_BACK_ON_PLEASE` so the I/O from `ls` will be captured.

```
export MISTRAL_BYPASS_PROGRAMS=hello_world.sh
export MISTRAL_BYPASS_DISABLE=TURN_TRACING_BACK_ON_PLEASE
```

9.2.1 Bypassing scheduler internal activity

This combination of `MISTRAL_BYPASS_PROGAMS` and `MISTRAL_BYPASS_DISABLE` is particularly useful for programs which are part of a scheduler or workflow management framework. It allows one to disregard I/O performed directly by the framework, while still monitoring I/O from user programs run by the framework. For example, Altair Flowtracer or Accelerator programs can be disregarded, while still monitoring user jobs, with a configuration such as this:

```
export MISTRAL_BYPASS_PROGRAMS=$(dirname $VOVDIR)
export MISTRAL_BYPASS_DISABLE=VOV_JOBID
```

With this configuration, Mistral will turn off monitoring when a program or script is called from the directory above `$VOVDIR` - including Flowtracer or Accelerator programs - and it will turn back on once `VOV_JOBID` is set.

If setting `MISTRAL_BYPASS_PROGRAMS` to a hard-wired value for VOVDIR, then please ensure it is an absolute path to the directory as outlined in the above section.

9.3 Disabling specific function wrappers with mistral

Sometimes it may be desirable to disable the Mistral wrapper of a C API, in particular those which are part of a schedular's or workflow manager's external API for submitting jobs/tasks. For example, PBS exposes the `tm_spawn` function which can be used by a submitted job to spawn child tasks. Mistral wraps this function to trace these tasks however this can be disabled with a configuration like the below:

```
export MISTRAL_DISABLED_SUBMITTERS=tm_spawn
```

The current list of functions which can be disabled is:

`tm_spawn`, `lsb_launch`, `lsb_submit`, `lsb_submitPack`.

10 Technical reference

10.1 Host Memory Metrics

Mistral monitors host-level memory usage metrics and records them in the `Resources` record. The table below provides descriptions of these metrics, all measured in bytes.

Memory Metrics	Description
memfree	Free memory excluding Buffer, Cache, and SReclaimable. This is the memory that is completely free in the system.
memavail	Available memory including Buffer, Cache, and SReclaimable. This is the memory available for new applications, because the system can reclaim cached memory.
memused	Used memory, calculated as Total memory - Available memory.

These metrics are similar to that provided by the `free` command. However, note that different versions of `free` may calculate used memory differently. Some versions use `total - free - buffers - cache`, while others use `total - available`. Mistral follows the latter approach as it offers a more comprehensive view of memory usage.

10.2 GPU Profiling

Mistral tracks certain metrics for NVidia, Intel data center and AMD GPU devices and will output these if configured to do so. It does this by calling the relevant functions from the vendor's libraries to query GPU state. Note that not every GPU device supports every type of query exposed by the libraries, and so in Mistral these numbers will always be output as zero (or missing from the report.) For the complete reference, with supported features, see the offical guides for each vendor:

Vendor	C Library	Official guide
NVidia	NVidia Management Library "NVML"	NVML Reference Guide
Intel	XPU Management "XPUM"	XPUM Reference Guide
AMD	AMD System Management Interface Library "AMD-SMI"	AMD SMI Reference Guide
AMD	ROCm System Management Interface Library "ROCm SMI"	ROCm SMI Reference Guide

Note that both AMD SMI and ROCm SMI are supported by Mistral, and if both are installed, then AMD SMI will be used.

The below table details the metrics queried by Mistral and the name each vendor gives to each, as well as a brief description.

Mistral	NVidia	Intel	AMD	Description of Mistral Metric
energy	energy	energy	energy	Energy used by the GPU since last report.
read	PCIe receive throughput	PCIe read throughput	Unsupported in Mistral*	Bytes read by the GPU since last report.

Mistral	NVidia	Intel	AMD	Description of Mistral Metric
write	PCIe transmit throughput	PCIe write throughput	Unsupported in Mistral*	Bytes written by the GPU since last report.
utilisation	sm_utilization	GPU utilization	GPU activity	Percentage of time in which the GPU was used.
mem	memory	memory used	memory used	Virtual Memory size for the GPU.
temperature	temperature	GPU temperature	GPU hotspot temperature	Temperature reported by the GPU.

* Because of how AMD SMI and ROCm SMI retrieve these numbers, they are unsuitable for use in Mistral, taking more than one second per library call.

10.3 Log entries

If Mistral detects I/O on a mountpoint (or cpu/memory or GPU usage, if configured), it outputs log entries, either to the `MISTRAL_LOG` file or to a plug-in (see the Plug-ins section).

Mistral logs I/O on most file systems, but ignores a small number of specialised file system types, for example `devpts` which is used to implement pseudo terminals. The full list of ignored file system types is:

> `anon_inodefs`, `bdev`, `binfmt_misc`, `bpf`, `cgroup`, `cgroup2`, `configfs`, `cpuset`, `debugfs`, `devfs`, `devpts`, `dlmfs`, `efivarfs`, `fuse`, `fuse.archivemount`, `fuse.dumpfs`, `fuse.encfs`, `fuse.gvfs-fuse-daemon`, `fuse.gvfsd-fuse`, `fuse.rofiles-fuse`, `fuse.xwmfs`, `fusectl`, `hugetlbfs`, `mqueue`, `nfsd`, `none`, `nsfs`, `pipefs`, `pstore`, `ramfs`, `rpc_pipefs`, `securityfs`, `selinuxfs`, `sockfs`, `spufs`, `usbfs`

I/O on anonymous inodes is also ignored.

Log entries to the `MISTRAL_LOG` file are in JSON format, and has different formats based on the data outputted:

```
{
  "version": version,
  "timestamp": timestamp,
  "hostname": hostname,
  "jobid": job_id,
  "jobgroupid": job_group_id,
  "type": data_type,
  "jobtotal": true/false,
  "environment": {
    "ENV": var
  }
  "jobrealtime": real_time,
  "jobstarttime": start_time,
  "jobendtime": end_time,
  variable: data_object,
}
```

Note that the specific fields that are included in the output depends upon your mistral configuration. The field definitions are as follows:

`version` is the current JSON-schema version number. The full schema document is in `docs/mistral_schema.json`.

`timestamp` is the end of the time frame when the I/O occurred. The time-stamp is in ISO 8601 format with second precision and UTC offset (`YYYY-MM-DDThh:mm:ss+hhmm`).

`hostname` is the name of the host on which the job was running. The host name includes the domain name.

`jobid` is the job identifier for the job.

`jobgroupid` is the job group identifier for the job group.

`type` is the type of record and may be either `mountpoint`, `network`, `resources`, `gpu`, `mountpointsummary` or `jobsummary`.

`total` is a boolean value of whether the entry represents the total measurement for the whole job or not.

`environment` is a list of variable-value key-pairs for the environment variables requested in the config file. It is omitted if none were requested. The value will be an empty string if either the variable is not set or is set but is equal to the empty string.

`jobrealtime` is only included when `jobtotal` is `true` and is the wall clock real time of the job in microseconds.

`jobstarttime` is only included when `jobtotal` is `true` and is the milliseconds since epoch of when the job began.

`jobendtime` is only included when `jobtotal` is `true` and is the milliseconds since epoch of when the job ended.

`variable` are properties with a label that will depend on the `type` of record, as does its values `data_object`. These are described in the sections below.

10.3.1 Mountpoint Record

There are three `variable` properties for a record whose `type` is `mountpoint`, each with the following formats:

```
"timeframe": timeframe,
"cumulative": true/false,
"io": {
  "mountpoint": {
    "path": path,
    "fstype": fstype,
    "fsname": fs_name,
    "fshost": fs_host
  },
  calltype: {
    measurement: {
      "total": total_count,
      "min/s": minimum_per_second,
      "mean/s": mean_per_second,
      "median/s": median_per_second,
      "max/s": maximum_per_second,
      "mean/call": mean_per_call,
```

```
      "max/call": max_per_call
    }
  }
}
```

`timeframe` is the length of the timeframe in seconds.

`cumulative` is a boolean field indicating if the record is for all mount points or not.

`mountpoint` is an identifier for the mount point that the record is about.

`path` is the I/O mount point reported by this log entry. The value `invalid` represents an attempt to do I/O on a closed file descriptor. (Note that this is not the same as `/invalid`, which is what would appear if the system has a file system mounted on a directory called `/invalid`.) A value of "*" represents the total across all mount points, including those reported as `invalid`.

`fstype` is the filesystem type of `path`, or "*" if this entry is the total across all mount points. If `path` is `invalid` then `fstype` will also be `invalid`.

`fsname` is the so-called filesystem "name" of `path`, typically a device name or an NFS `HOST:PATH` specification, or "*" if this entry is the total across all mount points.

`fshost` is the host name part of `fsname` if present, or "*" if this entry is the total across all mount points.

`calltype` is a textual label made up of 2 parts, separated by an underscore. The first is one of the call types `read`, `write`, `open`, `access`, `create`, `delete`, `fschange`, `mmap`, `seek`, `connect`, `accept`; or `metadata` (for the total of the call types other than `read`, `write` and `seek`). The second part is the size-range and is described below.

The size-range is `<MIN>-<MAX>` for size-binned read/write data, where `<MIN>` and `<MAX>` are the lower and upper operation sizes for the bin being reported, and are both powers of two. Note that the range is exclusive of `<MAX>`, so that (for instance) if the size range is `0-32KiB` (as for small/medium/large size binning), then it will include operations up to and including 32,767 bytes, but operations of exactly 32 kibibytes will instead be reported under `32KiB-128MiB`. For entries without a size range (such as entries for `seek` or metadata functions, or resource entries), this appears as 'all.

When `read.sized` or `write.sized` is set to `all`, the value of `<MAX>` is always double the value of `<MIN>`, except for the first bin which is reserved for reads or writes of exactly zero bytes, and has a `<SIZE-RANGE>` of `0-1B`. The second bin has a `<SIZE-RANGE>` of `1B-2B`, so includes single-byte operations only.

Both `<MIN>` and `<MAX>` are expressed using the IEC standard prefixes such as "KiB" for "kibibyte" (1024 bytes), "MiB" for "mebibyte" (1,048,576 bytes), and so on.

`measurement` is a textual label describing the category what was measured. It changes per measurement and may be one of the following:

`bytes`	This entry reports data for the number of bytes read or written by the stated call type in the stated operation size range on the stated mount point in the timeframe.
`calls`	This entry reports data for the number of calls of the stated call type with the given operation size range on the stated mount point in the timeframe.
`duration`	This entry reports data for the duration sampled of stated call type with the given operation size range on the stated mount point in the timeframe.

Each of these measurements report on the following:

`total`	The total value measured in the timeframe.
`min/s`	The minimum rate measured, grouping measurements on a per second basis for the timeframe.
`mean/s`	The mean rate measured, grouping measurements on a per second basis for the timeframe.
`median/s`	The median rate measured, grouping measurements on a per second basis for the timeframe.
`max/s`	The maximum rate measured, grouping measurements on a per second basis for the timeframe.

In addition, `duration` reports two extra calculations:

`mean/call`	The mean duration measured per call.
`max/call`	The maximum duration for a single call.

An example mountpoint record looks like…


```
{
  "version": "3.1.2",
  "timestamp": "2023-09-19T13:38:20+01:00",
  "hostname": "suffix",
  "jobid": "1283.pbs",
  "jobgroupid": "1283.pbs",
  "timeframe": "11s",
  "type": "mountpoint",
  "jobtotal": false,
  "mountpoint": {
    "path": "/", "fstype": "ext4", "fsname": "/dev/sda3", "fshost": ""
  },
  "io": {
    "read_all": {
      "bytes": {
        "total": 4215903517, "mean/s": 383263956, "median/s": 540497920,
        "max/s": 1151559680
      },
      "calls": {
        "total": 514540, "mean/s": 46776, "median/s": 53606, "max/s": 112457
      },
      "duration": {
        "total": 722806, "mean/s": 65709, "min/s": 0, "median/s": 82341,
        "max/s": 181955, "mean/call": 1, "max/call": 265
      }
    },
    "read_0-32KiB": {
      "bytes": {
        "total": 4215903517, "mean/s": 383263956, "median/s": 540497920,
        "max/s": 1151559680
```

```
      },
      "calls": {
        "total": 514540, "mean/s": 46776, "median/s": 53606, "max/s": 112457
      },
      "duration": {
        "total": 722806, "mean/s": 65709, "min/s": 0, "median/s": 82341,
        "max/s": 181955, "mean/call": 1, "max/call": 265
      }
    },
    "write_all": {
      "bytes": {
        "total": 4002301619, "mean/s": 363845601, "median/s": 504375934,
        "max/s": 1142160760
      },
      "calls": {
        "total": 492584, "mean/s": 44780, "median/s": 68162, "max/s": 119113
      },
      "duration": {
        "total": 2683727, "mean/s": 243975, "min/s": 0, "median/s": 356555,
        "max/s": 658933, "mean/call": 5, "max/call": 132
      }
    },
    "write_0-32KiB": {
      "bytes": {
        "total": 4002301619, "mean/s": 363845601, "median/s": 504375934,
        "max/s": 1142160760
      },
      "calls": {
        "total": 492584, "mean/s": 44780, "median/s": 68162, "max/s": 119113
      },
      "duration": {
        "total": 2683727, "mean/s": 243975, "min/s": 0, "median/s": 356555,
        "max/s": 658933, "mean/call": 5, "max/call": 132
      }
    },
    "metadata": {
      "calls": {
        "total": 436024, "min/s": 17219, "mean/s": 39638, "median/s": 38539,
        "max/s": 58592
      },
      "duration": {
        "total": 4775038, "mean/s": 434094, "min/s": 141090, "median/s": 504035,
        "max/s": 702112, "mean/call": 11, "max/call": 4905
      }
    },
    "seek": {
      "calls": { "total": 5, "max/s": 4 }
    }
  },
  "environment": { "SHELL": "/bin/bash", "USER": "user1" }
```

```
}
```

10.3.2 Network Record

This type of record is disabled by default and can be turned on by enabling network in your Mistral config. There are three `variable` properties for this record with following formats:

```
"timeframe": timeframe,
"cumulative": true/false,
"io": {
  "address": address
  calltype: {
    measurement: {
      "total": total_count,
      "min/s": minimum_per_second,
      "mean/s": mean_per_second,
      "median/s": median_per_second,
      "max/s": maximum_per_second,
      "mean/call": mean_per_call,
      "max/call": max_per_call
    }
  }
}
```

`address` is a string representation of an IP address and is an identifier for the record.

`calltype`, `timeframe` and `cumulative` are analogous to the above description for the `mountpoint` record.

An example network record looks like…

```
{
  "version": "3.1.2",
  "timestamp": "2023-09-19T13:38:27+01:00",
  "hostname": "suffix",
  "jobid": "1283.pbs",
  "jobgroupid": "1283.pbs",
  "timeframe": "19s",
  "type": "network",
  "jobtotal": false,
  "address": "104.16.212.134",
  "io": {
    "read_all": {
      "bytes": { "total": 549, "mean/s": 28, "max/s": 549 },
      "calls": { "total": 1, "max/s": 1 },
      "duration": {
        "total": 4, "mean/s": 0, "min/s": 0, "median/s": 0,
        "max/s": 4, "mean/call": 4, "max/call": 4
      }
    },
```

```
    "read_0-32KiB": {
      "bytes": { "total": 549, "mean/s": 28, "max/s": 549 },
      "calls": { "total": 1, "max/s": 1 },
      "duration": {
        "total": 4, "mean/s": 0, "min/s": 0, "median/s": 0,
        "max/s": 4, "mean/call": 4, "max/call": 4
      }
    },
    "write_all": {
      "bytes": { "total": 74, "mean/s": 3, "max/s": 74 },
      "calls": { "total": 1, "max/s": 1 },
      "duration": {
        "total": 27, "mean/s": 1, "min/s": 0, "median/s": 0,
        "max/s": 27, "mean/call": 27, "max/call": 27
      }
    },
    "write_0-32KiB": {
      "bytes": { "total": 74, "mean/s": 3, "max/s": 74 },
      "calls": { "total": 1, "max/s": 1 },
      "duration": {
        "total": 27, "mean/s": 1, "min/s": 0, "median/s": 0,
        "max/s": 27, "mean/call": 27, "max/call": 27
      }
    },
    "metadata": {
      "calls": { "total": 1, "max/s": 1 },
      "duration": {
        "total": 410409, "mean/s": 21600, "min/s": 0, "median/s": 0,
        "max/s": 410409, "mean/call": 410409, "max/call": 410409
      }
    }
  },
  "environment": { "SHELL": "/bin/bash", "USER": "user1" }
}
```


10.3.3 Resources Record

The single `variable` property of a record whose `type` is `resources` takes on the label `resources`. It is disabled by default but can be output by enabling cpu/memory in your Mistral config. The `resources` property has the following format:

```
"timeframe": timeframe,
"resources": {
  "host": {
    "cpuusertime": {
      "total": total_count,
      "min/s": minimum_per_second,
      "mean/s": mean_per_second,
      "median/s": median_per_second,
      "max/s": maximum_per_second
```

```
    },
    "cpusystemtime": {
      "total": total_count,
      "min/s": minimum_per_second,
      "mean/s": mean_per_second,
      "median/s": median_per_second,
      "max/s": maximum_per_second
    },
    "iowait": {
      "total": total_count,
      "min/s": minimum_per_second,
      "mean/s": mean_per_second,
      "median/s": median_per_second,
      "max/s": maximum_per_second
    },
    "memused": {
      "min": minimum_memused,
      "max": maximum_memused,
      "mean/s": mean_memused_per_second
    },
    "memfree": {
      "min": minimum_memfree,
      "max": maximum_memfree,
      "mean/s": mean_memfree_per_second
    },
    "memavail": {
      "min": minimum_memavail,
      "max": maximum_memavail,
      "mean/s": mean_memavail_per_second
    }
  },
  "job": {
    "cpuusertime": {
      "total": total_count,
      "min/s": minimum_per_second,
      "mean/s": mean_per_second,
      "median/s": median_per_second,
      "max/s": maximum_per_second
    },
    "cpusystemtime": {
      "total": total_count,
      "min/s": minimum_per_second,
      "mean/s": mean_per_second,
      "median/s": median_per_second,
      "max/s": maximum_per_second
    },
    "rssmem": {
      "max": maximum_rssmemory,
      "mean/s": mean_rssmemory_per_second
    },
```

```
    "vmem": {
      "max": maximum_virtual_memory,
      "mean/s": mean_virtual_memory_per_second
    },
  }
}
```

`timeframe` is as above with mountpoint and network records.

`host` is a collection of the resource usage for the current host. Each measurement reports the same as with I/O data: a `total` and various rates described above.

`cpuusertime` is the amount of time that the job has been scheduled in user mode on the reported host, measured in **micro-seconds**.

`cpusystemtime` is the amount of time that the job has been scheduled in kernel mode on the reported host, measured in **micro-seconds**.

`iowait` is the amount of time that the host has spent waiting for an I/O task related to the job to complete.

`job` is similar to `host` and is a collection of the resource usage for the entire job. The entries `cpuusertime` and `cpusystemtime` are as above but with respect to the entire job.

`rssmem` shows the size of real memory the job used, in bytes, and the mean size when sampled every second.

`vmem` is the maximum size of virtual memory that the job used, in bytes, and the mean size when sampled every second.

An example resources record looks like…


```
{
  "version": "3.1.2",
  "timestamp": "2023-09-19T13:38:27+01:00",
  "hostname": "suffix",
  "jobid": "1283.pbs",
  "jobgroupid": "1283.pbs",
  "timeframe": "8s",
  "type": "resources",
  "jobtotal": false,
  "resources": {
    "host": {
      "cpuusertime": {
        "total": 1420000, "min/s": 170000, "mean/s": 202857,
        "median/s": 210000, "max/s": 230000
      },
      "cpusystemtime": {
        "total": 8030000, "min/s": 1030000, "mean/s": 1147142,
        "median/s": 1150000, "max/s": 1280000
      },
      "cputotaltime": {
        "total": 9450000, "min/s": 1250000, "mean/s": 1350000,
        "median/s": 1360000, "max/s": 1450000
      },
      "iowait": {
        "total": 4210000, "min/s": 290000, "mean/s": 601428,
```

```
        "median/s": 620000, "max/s": 790000
      }
    },
    "job": {
      "cpuusertime": {
        "total": 1340000, "min/s": 10000, "mean/s": 167500,
        "median/s": 185000, "max/s": 210000
      },
      "cpusystemtime": {
        "total": 5670000, "min/s": 10000, "mean/s": 708750,
        "median/s": 800000, "max/s": 840000
      },
      "cputotaltime": {
        "total": 7010000, "min/s": 20000, "mean/s": 876250,
        "median/s": 1000000, "max/s": 1010000
      },
      "cpus": { "total": 7, "mean/s": 1, "median/s": 1, "max/s": 1 },
      "rssmem": { "max": 12058624, "mean/s": 5980160 },
      "vmem": { "max": 102293504, "mean/s": 22807552 }
    }
  },
  "environment": { "SHELL": "/bin/bash", "USER": "user1" }
}
```


10.3.4 GPU Record

The three `variable` properties of a record whose `type` is `gpu` are `name`, `cumulative` and `usage`: these refer to the path to the device id; whether representing all GPUs or not; and usage values of a particular GPU, respectively. This is disabled by default but has the following format:

```
"timeframe": timeframe
"gpu": path,
"cumulative": true/false
"usage": {
  "energy": {
    "total": total_energy,
    "min/s": minimum_per_second,
    "mean/s": mean_per_second,
    "median/s": median_per_second,
    "max/s": maximum_per_second
  },
  "read": {
    "total": total_bytes_read,
    "min/s": minimum_per_second,
    "mean/s": mean_per_second,
    "median/s": median_per_second,
    "max/s": maximum_per_second
  },
  "write": {
    "total": total_bytes_written,
```

```
    "min/s": minimum_per_second,
    "mean/s": mean_per_second,
    "median/s": median_per_second,
    "max/s": maximum_per_second
  },
  "utilisation": {
    "min/s": minimum_per_second,
    "mean/s": mean_per_second,
    "median/s": median_per_second,
    "max/s": maximum_per_second
  },
  "temperature": {
    "max": maximum_temperature
    "mean/s": mean_temperature_per_second,
  },
  "mem": {
    "max": maximum_memory
    "mean/s": mean_memory_per_second,
  }
}
```

`timeframe` is as above with resources, mountpoint and network records.

`cumulative` is analogous with mountpoint and network records but across GPU devices.

`gpu` is the path to the device or `"*"` when `cumulative` is `true`.

`usage` is the collection of metrics that are reported for the GPU.

`energy` shows the total energy consumed, in joules, as well as rates.

`read` is the number of bytes read from the PCI-e bus by the GPU.

`write` is the number of bytes transmitted to the PIC-e bus by the GPU.

`utilisation` is the amount of time the GPU was being used by some process, as a percentage. Note that this does not represent the saturation of the GPU i.e. not considering the capacity of the GPU, but rather a frequency of use.

`temperature` shows the maximum temperature the GPU reached, in degrees Celsius, as well as the mean temperature since the last report.

`mem` shows the maximum memory used by the GPU, in bytes, as well as mean use when sampled every second.

An example of a GPU record is as below…


```
{
  "version": "3.2.5",
  "timestamp": "2024-09-10T15:45:20+02:00",
  "hostname": "suffix",
  "jobid": "1283.pbs",
  "jobgroupid": "1283.pbs",
  "timeframe": "8s",
  "type": "gpu",
  "jobtotal": false,
```

```
  "gpu": "/dev/nvidia0",
  "usage": {
    "energy": {
     "total": 1108.0, "min/s": 4.0, "mean/s": 110.0, "median/s": 126.0, "max/s":
      ↪  127.0
    },
    "read": {
      "total": 3072.0, "mean/s": 307.0, "max/s": 2048.0
    },"write": {
      "total": 3072.0, "mean/s": 307.0, "max/s": 2048.0
    },
    "utilisation": {
      "mean/s": 89.0, "median/s": 99.0, "max/s": 99.0
    },
    "mem": {
      "max": 1321855877.0, "mean/s": 1231823044.0
    },
    "temperature": {
      "max": 49.0, "mean/s": 42.0
    }
  },
  "environment": { "SHELL": "/bin/bash", "USER": "user1" }
}
```


10.3.5 Job Summary Record

The `variable` property of a record whose `type` is `jobsummary` takes on the label `iosummary`. This type of record is disabled by default and can be enabled by turning on totals in your Mistral config file. It is a collection of aggregated statistics about the type of I/O that was performed. This has been split into three categories: good (green), medium (yellow) and bad (red). The `iosummary` property has the following format:

```
"iosummary": {
  "total": {
    "accumulatedruntime": run_time,
    "accumulatediotime": io_time,
    "iotimepercentage": io_time_percentage ,
    "calls": io_calls
  },
  "red": {
    "time": {
      "iotime": total_red_time,
      "iopercentage": percentage_red_time
    },
    "calls": {
      "iocount": total_red_calls,
      "iopercentage": percentage_red_calls
    }
  },
  "yellow": {
    "time": {
```

```
      "iotime": total_yellow_time,
      "iopercentage": percentage_yellow_time
    },
    "calls": {
      "iocount": total_yellow_calls,
      "iopercentage": percentage_yellow_calls
    }
  },
  "green": {
    "time": {
      "iotime": total_green_time,
      "iopercentage": percentage_green_time
    },
    "calls": {
      "iocount": total_green_calls,
      "iopercentage": percentage_green_calls
    }
  }
}
```

`total` gives a basic overview of the entire job.

`accumulatedruntime` is the sum of each process' run time (wall-clock time), in microseconds. This may be higher than the `jobrealtime` for jobs with multiple processes running in parallel.

`accumulatediotime` is the sum of time each process spent in I/O operations, in microseconds.

`iotimepercentage` is the percentage of time spent in I/O operations, calculated from `accumulatediotime` and `jobrealtime`.

`calls` is the total number of I/O calls made.

`red`, `yellow` and `green` represent categories of I/O done, each of which is defined and explained below. In each category, the following is reported:

`time` reports the time (`iotime`) spent doing that category of I/O and the percentage (`iopercentage`) of I/O time in said category.

`calls` reports the number of calls (`iocount`) made that fall in to the category definition and the percentage (`iopercentage`) of I/O calls in that category too.

The following is an example of a job summary

```
{
  "version": "3.1.2",
  "timestamp": "2023-09-19T14:48:01+01:00",
  "hostname": "suffix",
  "jobid": "1283.pbs",
  "jobgroupid": "1283.pbs",
  "type": "jobsummary",
  "jobtotal": true,
  "iosummary": {
    "total": {
      "accumulatedruntime": 38341377,
      "accumulatediotime": 14886960,
```

```
      "iotimepercentage": 75.4415180210036,
      "calls": 2623116
    },
    "red": {
      "time": { "iotime": 9427079, "iopercentage": 63.324406057381765 },
      "calls": { "iocount": 1977438, "iopercentage": 75.3850763748153 }
    },
    "yellow": {
      "time": { "iotime": 2685358, "iopercentage": 18.038323472354328 },
      "calls": { "iocount": 206474, "iopercentage": 7.871325553273283 }
    },
    "green": {
      "time": { "iotime": 2774523, "iopercentage": 18.637270470263907 },
      "calls": { "iocount": 439204, "iopercentage": 16.74359807191142 }
    }
  },
  "environment": { "SHELL": "/bin/bash", "USER": "user1" },
  "jobrealtime": 19733329,
  "jobstarttime": 16951336815,
  "jobendtime": 16951356548
}
```


10.3.5.1 How red, yellow and green percentages are calculated

Each I/O call has a duration measured in microseconds. Once the call is categorised under bad, medium or good I/O, we accumulate the call duration to get the time spent in red, yellow and green I/O operations. In addition we need to measure the total time the application spent doing I/O. The percentages are then simply calculated as:

- % Red time = (Time spent in bad I/O ops) / (Total time spent in I/O ops)
- % Yellow time = (Time spent in medium I/O ops) / (Total time spent in I/O ops)
- % Green time = (Time spent in good I/O ops) / (Total time spent in I/O ops)

For more information on how the durations are measured, see the Duration sampling section. Note that regardless of the `duration` settings in your Mistral config file for each individual call type, if `totals` is turned on, which it is by default, duration sampling will occur. As a result, turning off `totals` is likely to reduce the overhead of running Mistral.

Additionally, the duration measurements are not reported by Mistral if there were not enough measurements. This is to prevent scenarios whereby short jobs can have vastly overreported I/O time spent in any particular category. The minimum number of samples required is equal to the `MISTRAL_MONITOR_DURATION_SAMPLE` environment variable. If this is *n*, then at least *n* calls will have to have been sampled. If `MISTRAL_MONITOR_DURATION_SAMPLE` isn't set, then the default value of `10` will be used.

We don't calculate the percentages against the total wallclock runtime, because the application spends time also doing CPU intensive tasks, memory I/O, synchronization (locks), sleeping, etc.

In similar fashion, we calculate the percentages using call counts:

- % Red calls = (Number of bad I/O calls) / (Total I/O calls)

- % Yellow calls = (Number of medium I/O calls) / (Total I/O calls)
- % Green calls = (Number of good I/O calls) / (Total I/O calls)

We log the total time spent in I/O ops, which is:

- Total time spent in I/O ops = Red time + Yellow time + Green time and similarly for total number of I/O calls:
- Total number of I/O calls = Red calls + Yellow calls + Green calls

We also log how much of the total running time was spent in I/O:

- % I/O Time = (Total time spent in I/O ops) / (Total accumulated runtime of processes)

For multi-threaded processes, the times and call counts are accumulated from each thread. Therefore the total time spent in I/O may be greater than the total wallclock runtime, and equally % I/O Time may be greater than 100%.

10.3.5.2 Rules for bad I/O

Definition of bad I/O:

- Small reads or writes.
- Opens for files where nothing was written or read.
- Stats that succeeded on files that were not used.
- Failed I/O.
- Backward seeks.
- Zero seeks, reads, writes.
- Failed network I/O.

10.3.5.3 Rules for medium I/O

Definition of medium I/O:

- Opens for files from which less than N bytes were read or written.
- Stats of files that were used later.
- Forward seeks.
- Sync calls including `sync`, `fsync`, `fdatasync`, `sync_file_range` and `syncfs`. (Note that calls to `fflush` on a file opened for writing are treated as writes, and classified as *good* or *bad* depending on the amount of data actually written.)

10.3.5.4 Rules for good I/O

Definition of good I/O:

- Reads and writes greater than `MISTRAL_PROFILE_SMALL_IO`
- Opens for files from which at least `MISTRAL_PROFILE_SMALL_IO` bytes were read or written.
- Successful closes
- Successful network I/O.

10.3.6 Mount Point Summary Record

A mount point summary record is similar to a job summary but relates to I/O on a single mount point. The `type` property will be `mountpointsummary` and it includes a `mountpoint` property which identifies the mount point in question.

The following is an example of a mount point summary.


```
{
  "version": "3.1.2",
  "timestamp": "2023-09-19T14:48:01+01:00",
  "hostname": "suffix",
  "jobid": "1283.pbs",
  "jobgroupid": "1283.pbs",
  "type": "mountpointsummary",
  "jobtotal": true,
  "mountpoint": {
    "path": "/",
    "fstype": "ext4",
    "fsname": "/dev/sda3",
    "fshost": ""
  },
  "iosummary": {
    "total": {
      "accumulatediotime": 14840348,
      "iotimepercentage": 75.2053059241084,
      "calls": 2623097
    },
    "red": {
      "time": { "iotime": 9427071, "iopercentage": 63.523247568048944 },
      "calls": { "iocount": 1977436, "iopercentage": 75.38554616928005 }
    },
    "yellow": {
      "time": { "iotime": 2685354, "iopercentage": 18.094953029403353 },
      "calls": { "iocount": 206469, "iopercentage": 7.871191953633434 }
    },
    "green": {
      "time": { "iotime": 2727923, "iopercentage": 18.381799402547703 },
      "calls": { "iocount": 439192, "iopercentage": 16.743261877086514 }
```

```
    }
  },
  "environment": { "SHELL": "/bin/bash", "USER": "user1" },
  "jobrealtime": 19733329,
  "jobstarttime": 16951336815,
  "jobendtime": 16951356548
}
```