Package: hipercow (via r-universe)

July 3, 2024

Title High Performance Computing

Version 1.0.27

Description Set up cluster environments and jobs. Moo.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.1

URL https://github.com/mrc-ide/hipercow,
https://mrc-ide.github.io/hipercow

BugReports https://github.com/mrc-ide/hipercow/issues

Imports audio, cli, fs, ids, rlang, withr

Suggests bench, callr, conan2 (>= 1.9.95), dust, furrr, future, knitr, logwatch, mockery, openssl, pkgdepends, prettyunits, redux, rmarkdown, rrq, testthat (>= 3.0.0)

Config/testthat/edition 3

Remotes mrc-ide/conan2, mrc-ide/dust, mrc-ide/rrq, reside-ic/logwatch

VignetteBuilder knitr

Language en-GB

Repository https://mrc-ide.r-universe.dev

RemoteUrl https://github.com/mrc-ide/hipercow

RemoteRef main

RemoteSha c1f77abfb33e8aa9276090d7cf9f054c78fff981

Contents

`hipercow_bundle_cancel` *Cancel bundle tasks*

Description

Cancel all tasks in a bundle. This wraps task_cancel for all the ids.

Usage

```
hipercow_bundle_cancel(bundle, follow = TRUE, root = NULL)
```

Arguments

`bundle`	Either a `hipercow_bundle` object, or the name of a bundle.
`follow`	Logical, indicating if we should follow any retried tasks.
`root`	A hipercow root, or path to it. If `NULL` we search up your directory tree.

Value

A logical vector the same length as `id` indicating if the task was cancelled. This will be `FALSE` if the job was already completed, not running, etc.

Examples

```
cleanup <- hipercow_example_helper(runner = FALSE)

bundle <- task_create_bulk_expr(sqrt(x), data.frame(x = 1:5))
hipercow_bundle_cancel(bundle)
hipercow_bundle_status(bundle)

cleanup()
```

`hipercow_bundle_create` *Create task bundle*

Description

Create a bundle of tasks. This is simply a collection of tasks that relate together in some way, and we provide some helper functions for working with them that save you writing lots of loops. Each bundle has a name, which will be randomly generated if you don't provide one, and a set of task ids.

Usage

```
hipercow_bundle_create(
  ids,
  name = NULL,
  validate = TRUE,
  overwrite = TRUE,
  root = NULL
)
```

Arguments

ids	A character vector of task ids
name	A string, the name for the bundle. If not given, then a random name is generated. Names can contain letters, numbers, underscores and hyphens, but cannot contain other special characters.
validate	Logical, indicating if we should check that the task ids exist. We always check that the task ids are plausible.
overwrite	Logical, indicating that we should overwrite any existing bundle with the same name.
root	A hipercow root, or path to it. If NULL we search up your directory tree.

Value

A task bundle object

Examples

```
cleanup <- hipercow_example_helper()

# Two task that were created separately:
id1 <- task_create_expr(sqrt(1))
id2 <- task_create_expr(sqrt(2))

# Combine these tasks together in a bundle:
bundle <- hipercow_bundle_create(c(id1, id2))

# Now we can use bundle operations:
hipercow_bundle_status(bundle)
hipercow_bundle_wait(bundle)
hipercow_bundle_result(bundle)

cleanup()
```

`hipercow_bundle_delete` *Delete task bundles*

Description

Delete one or more hipercow task bundles. Note that this does not delete the underlying tasks, which is not yet supported.

Usage

```
hipercow_bundle_delete(name, root = NULL)
```

Arguments

`name`	Character vectors of names to delete
`root`	A hipercow root, or path to it. If `NULL` we search up your directory tree.

Value

Nothing, called for its side effect

Examples

```
cleanup <- hipercow_example_helper()

bundle <- task_create_bulk_expr(sqrt(x), data.frame(x = 1:5))
hipercow_bundle_list()

# Retaining the ids, delete bundle
ids <- bundle$ids
hipercow_bundle_delete(bundle$name)
hipercow_bundle_list()

# The tasks still exist:
task_status(ids)

cleanup()
```

hipercow_bundle_list *List existing bundles*

Description

List existing bundles

Usage

```
hipercow_bundle_list(root = NULL)
```

Arguments

root	A hipercow root, or path to it. If NULL we search up your directory tree.

Value

A data.frame with columns name and time, ordered by time (most recent first)

Examples

```
cleanup <- hipercow_example_helper()

# With no bundles present
hipercow_bundle_list()

# With a bundle
bundle <- task_create_bulk_expr(sqrt(x), data.frame(x = 1:5))
hipercow_bundle_list()

cleanup()
```

hipercow_bundle_load *Load existing bundle*

Description

Load an existing saved bundle by name. This is intended for where you have created a long-running bundle and since closed down your session. See hipercow_bundle_list for finding names of bundles.

Usage

```
hipercow_bundle_load(name, root = NULL)
```

Arguments

name	Name of the bundle to load
root	A hipercow root, or path to it. If NULL we search up your directory tree.

Value

A `hipercow_bundle` object

Examples

```
cleanup <- hipercow_example_helper()

bundle <- task_create_bulk_expr(sqrt(x), data.frame(x = 1:5))
name <- bundle$name

# Delete the bundle object; the bundle exists still in hipercow's store.
rm(bundle)

# With the name we can load the bundle and fetch its status
bundle <- hipercow_bundle_load(name)
hipercow_bundle_status(bundle)

# In fact, you can use just the name if you prefer:
hipercow_bundle_status(name)

cleanup()
```

`hipercow_bundle_log_value` *Fetch bundle logs*

Description

Fetch logs from tasks in a bundle.

Usage

```
hipercow_bundle_log_value(bundle, outer = FALSE, follow = TRUE, root = NULL)
```

Arguments

`bundle`	Either a `hipercow_bundle` object, or the name of a bundle.
`outer`	Logical, indicating if we should request the "outer" logs; these are logs from the underlying HPC software before it hands off to hipercow.
`follow`	Logical, indicating if we should follow any retried tasks.
`root`	A hipercow root, or path to it. If `NULL` we search up your directory tree.

Value

A list with each element being the logs for the corresponding element in the bundle.

Examples

```
cleanup <- hipercow_example_helper(with_logging = TRUE)
bundle <- task_create_bulk_expr(sqrt(x), data.frame(x = 1:2))
hipercow_bundle_wait(bundle)
hipercow_bundle_log_value(bundle)

cleanup()
```

`hipercow_bundle_result` *Fetch bundle results*

Description

Fetch all bundle results

Usage

```
hipercow_bundle_result(bundle, follow = TRUE, root = NULL)
```

Arguments

`bundle`	Either a `hipercow_bundle` object, or the name of a bundle.
`follow`	Logical, indicating if we should follow any retried tasks.
`root`	A hipercow root, or path to it. If `NULL` we search up your directory tree.

Value

An unnamed list, with each element being the result for each a task in the bundle, in the same order.

Examples

```
cleanup <- hipercow_example_helper()
bundle <- task_create_bulk_expr(sqrt(x), data.frame(x = 1:5))
hipercow_bundle_wait(bundle)
hipercow_bundle_result(bundle)

cleanup()
```

`hipercow_bundle_retry` *Retry task bundle*

Description

Retry tasks in a bundle. This has slightly different semantics to `task_retry()`, which errors if a retry is not possible. Here, we anticipate that much of the time you will be interested in retrying some fraction of your bundle and so don't need to wait until all tasks have finished in order to retry failed tasks.

Usage

```
hipercow_bundle_retry(bundle, if_status_in = NULL, driver = NULL, root = NULL)
```

Arguments

`bundle`	Either a `hipercow_bundle` object, or the name of a bundle.
`if_status_in`	Optionally, a character vector of task statuses for which we should retry tasks. For example, pass `if_status_in = c("cancelled", "failure")` to retry cancelled and failed tasks. Can only be terminal statuses (`cancelled`, `failure`, `success`).
`driver`	Name of the driver to use to submit the task. The default (`NULL`) depends on your configured drivers; if you have no drivers configured no submission happens (or indeed is possible). If you have exactly one driver configured we'll submit your task with it. If you have more than one driver configured, then we will error, though in future versions we may fall back on a default driver if you have one configured. If you pass `FALSE` here, submission is prevented even if you have no driver configured.
`root`	A hipercow root, or path to it. If `NULL` we search up your directory tree.

Value

Invisibly, a logical vector, indicating which of the tasks within the bundle were retried. This means that it's not immediately obvious how you can get the new id back from the tasks, but typically that is unimportant, as all bundle functions follow retries by default.

Examples

```
cleanup <- hipercow_example_helper()
bundle <- task_create_bulk_expr(rnorm(1, x), data.frame(x = 1:5))
hipercow_bundle_wait(bundle)

retried <- hipercow_bundle_retry(bundle)
retried
hipercow_bundle_wait(bundle)
hipercow_bundle_result(bundle, follow = FALSE)
hipercow_bundle_result(bundle, follow = TRUE)
```

```
cleanup()
```

`hipercow_bundle_status` *Bundle status*

Description

Fetch status for all tasks in a bundle.

Usage

```
hipercow_bundle_status(bundle, reduce = FALSE, follow = TRUE, root = NULL)
```

Arguments

`bundle`	Either a `hipercow_bundle` object, or the name of a bundle.
`reduce`	Reduce the status across all tasks in the bundle. This means we return a single value with the "worst" status across the bundle. We only return `success` if *all* tasks have succeeded, and will return `failed` if any task has failed.
`follow`	Logical, indicating if we should follow any retried tasks.
`root`	A hipercow root, or path to it. If `NULL` we search up your directory tree.

Value

A character vector the same length as the number of tasks in the bundle, or length 1 if `reduce` is `TRUE`.

Examples

```
cleanup <- hipercow_example_helper()
bundle <- task_create_bulk_expr(sqrt(x), data.frame(x = 1:5))
# Immediately after submission, tasks may not all be complete so
# we may get a mix of statuses.  In that case the reduced status
# will be "submitted" or "running", even though some tasks may be
# "success"
hipercow_bundle_status(bundle)
hipercow_bundle_status(bundle, reduce = TRUE)

# After completion all tasks have status "success", as does the
# reduction.
hipercow_bundle_wait(bundle)
hipercow_bundle_status(bundle)
hipercow_bundle_status(bundle, reduce = TRUE)

cleanup()
```

`hipercow_bundle_wait` *Wait for a bundle to complete*

Description

Wait for tasks in a bundle to complete. This is the generalisation of task_wait for a bundle.

Usage

```
hipercow_bundle_wait(
  bundle,
  timeout = NULL,
  poll = 1,
  fail_early = TRUE,
  progress = NULL,
  follow = TRUE,
  root = NULL
)
```

Arguments

`bundle`	Either a `hipercow_bundle` object, or the name of a bundle.
`timeout`	The time to wait for the task to complete. The default is to wait forever.
`poll`	Time, in seconds, used to throttle calls to the status function. The default is 1 second
`fail_early`	Logical, indicating if we should fail as soon as the first task has failed. In this case, the other running tasks continue running, but we return and indicate that the final result will not succeed. If `fail_early = FALSE` we keep running until all tasks have passed or failed, even though we know we will return `FALSE`; but upon return `hipercow_bundle_result()` can be called and all results/errors returned.
`progress`	Logical value, indicating if a progress spinner should be used. The default `NULL` uses the option `hipercow.progress`, and if unset displays a progress bar in an interactive session.
`follow`	Logical, indicating if we should follow any retried tasks.
`root`	A hipercow root, or path to it. If `NULL` we search up your directory tree.

Value

A scalar logical value; `TRUE` if *all* tasks complete successfully and `FALSE` otherwise

Examples

```
cleanup <- hipercow_example_helper()

bundle <- task_create_bulk_expr(sqrt(x), data.frame(x = 1:5))
hipercow_bundle_wait(bundle)
hipercow_bundle_status(bundle)

cleanup()
```

`hipercow_cluster_info` *Describe cluster*

Description

Describe information about the cluster. This is (naturally) very dependent on the cluster but some details of the value are reliable; see Value for details.

Usage

```
hipercow_cluster_info(driver = NULL, root = NULL)
```

Arguments

`driver`	The driver to use, which determines the cluster to fetch information from (depending on your configuration). If no driver is configured, an error will be thrown.
`root`	Hipercow root, usually best `NULL`

Value

A list describing the cluster. The details depend on the driver, and are subject to change. We expect to see elements:

- resources: Describes the computational resources on the cluster, which is used by hipercow_resources_validate. Currently this is a simple list with elements `max_ram` (max RAM available, in GB), `max_cores` (max number of cores you can request), `queues` (character vector of available queues), `nodes` (character vector of available nodes), `default_queue` (the default queue). These details are subject to change but the contents should always be informative and fairly self explanatory.
- redis_url: The URL of the redis server to communicate with from outside of the cluster (i.e., from your computer), in a form suitable for use with redux::hiredis
- r_versions: A vector of R versions, as `numeric_vector` objects

Examples

```
cleanup <- hipercow_example_helper()
hipercow_cluster_info()
cleanup()
```

hipercow_configuration *Report on hipercow configuration*

Description

Report on your hipercow configuration. We will always want you to post this along side any problems; it has lots of useful information in it that will help us see how your set up is configured.

Usage

```
hipercow_configuration(show = TRUE, root = NULL)
```

Arguments

show	Display the configuration to the screen
root	Hipercow root, usually best NULL

Value

A list with a machine readable form of this information, invisibly.

Examples

```
cleanup <- hipercow_example_helper()
hipercow_configuration()

# If you have saved additional environments, they will be listed here:
file.create("functions.R")
hipercow_environment_create(
  name = "other",
  packages = "knitr",
  sources = "functions.R")
hipercow_configuration()

cleanup()
```

hipercow_configure *Configure your hipercow root*

Description

Configure your hipercow root. `hipercow_configure` creates the configuration and `hipercow_configuration` looks it up

Usage

```
hipercow_configure(driver, ..., root = NULL)
```

Arguments

`driver`	The hipercow driver; probably you want this to be `"windows"` as that is all we support at the moment!
`...`	Arguments passed to your driver; see Details for information about what is supported (this varies by driver).
`root`	Hipercow root, usually best `NULL`

Windows

Options supported by the `windows` driver:

- `shares`: Information about shares (additional to the one mounted as your working directory) that should be made available to the cluster job. The use case here is where you need access to some files that are present on a shared drive and you will access these by absolute path (say `M:/gis/shapefiles/`) from your tasks. You can provide a share as a `windows_path` object, or a list of such objects. You will not typically need to use this option.
- `r_version`: Control the R version used on the cluster. Typically hipercow will choose a version close to the one you are using to submit jobs, of the set available on the cluster. You can use this option to choose a specific version (e.g., pass "4.3.0" to select exactly that version).

See `vignette("details")` for more information about these options.

See Also

hipercow_unconfigure, which removes a driver

Examples

```
hipercow_configure("windows", r_version = "4.3.0")
```

`hipercow_driver` *Create a driver*

Description

Create a new hipercow driver; this is intended to be used from other packages, and rarely called directly. If you are trying to run tasks on a cluster you do not need to call this!

Usage

```
hipercow_driver(
  configure,
  submit,
  status,
  info,
  log,
  result,
  cancel,
  provision_run,
  provision_list,
  provision_compare,
  keypair,
  check_hello,
  cluster_info,
  default_envvars = NULL
)
```

Arguments

configure	Function used to set core configuration for the driver. This function will be called from the hipercow root directory (so `getwd()` will report the correct path). It can take any arguments, do any calculation and then must return any R object that can be serialised. The resulting configuration will be passed in as `config` to other driver functions.
submit	Submit a task to a cluster. This is run after the task is created (either automatically or manually) and takes as arguments the task id, the configuration, the path to the root.
status	Fetch a task status. Takes a vector of ids and returns a vector of the same length of statuses.
info	Fetch task info for a single task. May take longer than `status` and expected to retrieve the true status from the scheduler.
log	Fetch the task log. Takes a single task id and an integer (the number of lines already known) and returns a character vector of new logs. Return `NULL` (and not a zero length character vector) if a log is not available.
result	Fetch a task result. If needed, copies the result file into the current hipercow root. Assume that a result is available (i.e., we've already checked that the task status is terminal)
cancel	Cancel one or more tasks. Takes a vector of task ids, and requests that these tasks are cancelled, returning a list with elements `cancelled`: a logical vector the same length indicating if cancellation was successful, and `time_started`: the time that the task was started, or NA if the task was not yet started.
provision_run	Provision a library. Works with conan, and must accept `args`, `config`, and `path_root`. The `args` should be injected into `conan2::conan_configure`. It is expected this function will trigger running conan to provision a library. The return value is ignored, an error is thrown if the installation fails.

`provision_list` List previous installations. Takes `args` and if non-`NULL` injects into `conan2::conan_configure` (as for `provision_run`) in order to build a hash. Runs `conan2::conan_list` returning its value.

`provision_compare`
Test if a library is current. It is expected that this will call `conan2::conan_compare`

`keypair` Return a keypair as a list with elements `pub` and `key`; the public key as a string and the private key as a path that will be accessible when the cluster runs, but with permissions that are open only to the user who submitted the task.

`check_hello` Run any preflight checks before launching a hello world task. Return a validated resources list.

`cluster_info` Return information about a particular cluster: its maximum core count, maximum memory, node list and queue names, used for validating hipercow_resources against that cluster.

`default_envvars`
Driver-specific default environment variables. Drivers can use this to add environment variables that have a higher precedence than the hipercow defaults, but lower precedence than the `hipercow.default_envvars` option or the `envvars` argument to a task.

`hipercow_environment_create`
Manage environments

Description

Create, update, list, view and delete environments.

Usage

```
hipercow_environment_create(
  name = "default",
  packages = NULL,
  sources = NULL,
  globals = NULL,
  overwrite = TRUE,
  check = TRUE,
  root = NULL
)

hipercow_environment_list(root = NULL)

hipercow_environment_delete(name = "default", root = NULL)

hipercow_environment_show(name = "default", root = NULL)

hipercow_environment_exists(name = "default", root = NULL)
```

Arguments

name	Name of the environment. The name `default` is special; this is the environment that will be used by default (hence the name!). Environment names can contain letters, numbers, hyphens and underscores.
packages	Packages to be *attached* before starting a task. These will be loaded with `library()` before the `sources` are sourced. If you need to attach a package *after* a script for some reason, just call `library` yourself within one of your `source` files.
sources	Files to source before starting a task. These will be sourced into the global (or execution) environment of the task. The paths must be relative to the hipercow root, not the working directory.
globals	Names of global objects that we can assume exist within this environment. This might include function definitions or large data objects. The special value `TRUE` triggers automatic detection of objects within your environment (this takes a few seconds and requires that the environment is constructable on your local machine too, so is not currently enabled by default).
overwrite	On environment creation, replace an environment with the same name.
check	Logical, indicating if we should check the source files for issues. Pass `FALSE` here if you need to bypass these checks but beware the consequences that may await you.
root	A hipercow root, or path to it. If `NULL` we search up your directory tree.

Value

Nothing, all are called for their side effects.

Examples

```
cleanup <- hipercow_example_helper()

# Suppose you have a file with some functions you want to use in
# your task:
writeLines("simulation <- function(n) cumsum(rnorm(n))", "myfuns.R")

# Update the default environment to include these functions (or in
# this example, just this one function)
hipercow_environment_create(sources = "myfuns.R")

# You can now use this function in your tasks:
id <- task_create_expr(simulation(5))
task_wait(id)
task_result(id)

cleanup()
```

`hipercow_envvars` *Environment variables*

Description

Create environment variables for use with a hipercow task.

Usage

```
hipercow_envvars(..., secret = FALSE)
```

Arguments

`...`	<dynamic-dots> Named environment variable. If unnamed, it is assumed to refer to an environment variable that exists. Use an `NA` value to unset an environment variable.
`secret`	Are these environment variables secret? If so we will encrypt them at saving and decrypt on use.

Value

A list with class `hipercow_envvars` which should not be modified.

Examples

```
# Declare environment variables as key-value pairs:
hipercow_envvars("MY_ENVVAR1" = "value1", "MY_ENVVAR2" = "value2")

# If an environment variable already exists in your environment
# and you want to duplicate this into a task, you can use this
# shorthand:
Sys.setenv(HIPERCOW_EXAMPLE_ENVVAR = "moo") # suppose this exists already
hipercow_envvars("HIPERCOW_EXAMPLE_ENVVAR")
hipercow_envvars("HIPERCOW_EXAMPLE_ENVVAR", ANOTHER_ENVVAR = "value")

# Secret envvars are still printed (at the moment at least) but
# once passed into a task they will be encrypted at rest.
hipercow_envvars("MY_SECRET" = "password", secret = TRUE)

# Secret and public environment variables should be created
# separately and concatenated together:
env_public <- hipercow_envvars("HIPERCOW_EXAMPLE_ENVVAR")
env_secret <- hipercow_envvars("MY_PASSWORD" = "secret", secret = TRUE)
c(env_public, env_secret)

# Cleanup
Sys.unsetenv("HIPERCOW_EXAMPLE_ENVVAR")
```

hipercow_hello *Hello world*

Description

Hello world in hipercow. This function sends a tiny test task through the whole system to confirm that everything is configured correctly.

Usage

```
hipercow_hello(progress = NULL, timeout = NULL, driver = NULL)
```

Arguments

progress	Logical value, indicating if a progress spinner should be used. The default NULL uses the option hipercow.progress, and if unset displays a progress bar in an interactive session.
timeout	The time to wait for the task to complete. The default is to wait forever.
driver	The driver to use to send the test task. This can be omitted where you have exactly one driver, but we error if not given when you have more than one driver, or if you have not configured any drivers.

Value

The string "Moo", direct from your cluster.

Examples

```
cleanup <- hipercow_example_helper()
hipercow_hello()

cleanup()
```

hipercow_init *Create a hipercow root*

Description

Create a hipercow root. This marks the directory where your task information will be saved, along with a local copy of your R packages (a "library" for the cluster). Immediately after running this the first time, you probably want to run hipercow_configure() in order to control how we set up your projects network paths and R version.

Usage

```
hipercow_init(root = ".", driver = NULL, ...)
```

Arguments

`root`	The path to the root, defaulting the current directory.
`driver`	Optionally, the name of a driver to configure
`...`	Arguments passed through to hipercow_configure if `driver` is non-NULL.

Value

Invisibly, the root object

Examples

```
# Create an empty root
path <- withr::local_tempfile()
hipercow_init(path)
```

`hipercow_parallel` *Specify parallel use of cores*

Description

Set parallel options. Having requested more than one core using hipercow_resources, here hipercow can start up a local cluster on the node you are running on, using either the `future` or `parallel` package.

Usage

```
hipercow_parallel(
  method = NULL,
  cores_per_process = 1L,
  environment = NULL,
  use_rrq = FALSE
)
```

Arguments

`method`	The parallel method that hipercow will prepare. Three options are available: the `future` package, the `parallel` package, or `NULL`, the default, will do nothing. See the details for examples.
`cores_per_process`	The number of cores allocated to each process when launching a local cluster using one of the parallel methods. By default, this will be `1`. See details.
`environment`	The name of the environment to load into your parallel workers. The default is to use the environment that you submit your task with (which defaults to `default`), which means that each worker gets the same environment as your main process. This is often what you want, but can mean that you load too much into each worker and incur a speed or memory cost. In that case you may want to create a

new environment (hipercow_environment_create) that contains fewer packages or sources fewer functions and specify that here. If you want to suppress loading any packages into the workers you can use the `empty` environment, which always exists.

`use_rrq` Logical, indicating if you intend to use `rrq`-based workers from your tasks, in which case we will set a default controller. Enabling this requires that you have configured a rrq controller via `hipercow_rrq_controller()` before submitting the task (we check this before submission) and that you have submitted some workers via `hipercow_rrq_workers_submit()` (we don't check this because you will want them running at the time that your task starts, so you may want to launch them later depending on your workflow. We'll document this more in `vignete("rrq")`.

Details

Here, hipercow automatically does some setup work for the supported methods, to initialise a local cluster of processes that can be used with `future_map` or `clusterApply`, depending on your method.

By default, hipercow initialises a cluster with the same number of processes as the number of cores you requested using `hipercow_resources`. Each process here would be use a single core.

You can also call `hipercow_parallel` with `cores_per_process`, to make hipercow launch as many processes as it can with each process having the number of cores you request, with the total cores being at most what you requested with `hipercow_resources`.

For example, you could request 32 cores with `hipercow_resources`, and then call `hipercow_parallel` with `cores_per_process = 4`, and hipercow will create a local cluster with 8 processes, each of which reporting 4 cores if that process calls `hipercow_parallel_get_cores`.

If you did the same with `cores_per_process = 5`, hipercow would create 6 local processes, each reporting 5 cores, and two cores would be effectively unallocated.

Here are some brief examples; see `vignette("parallel")` for more details. In each example, we are looking up the process id (to show that different processes are being launched), and asking each process how many cores it should be using.

For using the `future` package:

```
resources <- hipercow_resources(cores = 4)
id <- task_create_expr(
  furrr::future_map(1:4,
    ~c(Sys.getpid(), hipercow_parallel_get_cores()),
  parallel = hipercow_parallel("future"),
  resources = resources)
```

where `furrr` must be provisioned using hipercow_provision. Here is an equivalent example with `parallel`:

```
resources <- hipercow_resources(cores = 4)
id <- task_create_expr(
```

```
  parallel::clusterApply(NULL, 1:4, function(x)
    c(Sys.getpid(), hipercow_parallel_get_cores()),
  parallel = hipercow_parallel("parallel"),
  resources = resources)
```

Value

A list containing your parallel configuration.

`hipercow_parallel_get_cores` *Get number of cores*

Description

Lookup number of cores allocated to the task

Usage

```
hipercow_parallel_get_cores()
```

Value

The number of cores a cluster has allocated to your task. This will be less than or equal to the number of cores on the cluster node running your task.

`hipercow_parallel_set_cores` *Set various environment variables that report the number of cores available for execution.*

Description

Sets the environment variables `MC_CORES`, `OMP_NUM_THREADS`, `OMP_THREAD_LIMIT`, `R_DATATABLE_NUM_THREADS` and `HIPERCOW_CORES` to the given number of cores. This is used to help various thread-capable packages use the correct number of cores. You can also call it yourself if you know specifically how many cores you want to be available to code that looks up these environment variables.

Usage

```
hipercow_parallel_set_cores(cores, envir = NULL)
```

Arguments

`cores`	Number of cores to be used.
`envir`	Environment in which the variables will be set to limit their lifetime. This should not need setting in general, but see `withr::local_envvar` for example use.

`hipercow_provision` *Provision cluster library*

Description

Provision a library. This runs a small task on the cluster to set up your packages. If you have changed your R version you will need to rerun this. See `vignette("packages")` for much more on this process.

Usage

```
hipercow_provision(
  method = NULL,
  ...,
  driver = NULL,
  environment = "default",
  check_running_tasks = TRUE,
  root = NULL
)
```

Arguments

`method`	The provisioning method to use, defaulting to `NULL`, which indicates we should try and detect the best provisioning mechanism for you; this should typically work well unless you are manually adding packages into your library (see Details). If given, must be one of `auto`, `pkgdepends`, `script` or `renv`; each of these are described in the Details and in `vignette("packages")`.
`...`	Arguments passed through to conan. See Details.
`driver`	The name of the driver to use, or you can leave blank if only one is configured (this will be typical).
`environment`	The name of the environment to provision (see hipercow_environment_create for details).
`check_running_tasks`	Logical, indicating if we should check that no tasks are running before starting installation. Generally, installing packages while tasks are running is harmful as you may get unexpected results, a task may start while a package is in an inconsistent state, and on windows you may get a corrupted library if a package is upgraded while it is loaded. You can disable this check by passing `FALSE`. Not all drivers respond to this argument, but the windows driver does.
`root`	The hipercow root

Details

Our hope is that that most of the time you will not need to pass any options through `...`, and that most of the time hipercow will do the right thing. Please let us know if that is not the case and you're having to routinely add arguments here.

Value

Nothing

Manually adding packages to an installation

One case where we do expect that you will pass options through to `hipercow_provision` is where you are manually adding packages to an existing library. The usage here will typically look like:

```
hipercow_provision("pkgdepends", refs = c("pkg1", "pkg2"))
```

where `pkg1` and `pkg2` are names of packages or pkgdepends references (e.g., `username/repo` for a GitHub package; see `vignette("packages")` for details).

Supported methods and options

There are four possible methods: `pkgdepends`, `auto`, `script` and `renv`.

The canonical source of documentation for all of these approaches is `conan2::conan_configure`.

`pkgdepends`**:**

The simplest method to understand, and probably most similar to the approach in `didehpc`. This method installs packages from a list in `pkgdepends.txt` in your hipercow root, or via a vector of provided package references. Uses pkgdepends for the actual dependency resolution and installation.

Supported options (passed via `...`)

- `refs`: A character vector of package references to override `pkgdepends.txt`
- `policy`: the policy argument to `pkgdepends::new_pkg_installation_proposal` (accepts `lazy` and `upgrade`)

`auto`**:**

Uses `pkgdepends` internally but tries to do everything automatically based on your declared environments (see `hipercow_environment_create` and `vignette("hipercow")`) and the installation information recorded in the locally installed versions of the required packages.

This is experimental and we'd love to know how it works for you.

No options are supported, the idea is it's automatic :)

`script`**:**

Runs a script (by default `provision.R`) on the cluster to install things however you want. Very flexible but you're on your own mostly. The intended use case of this option is where `pkgdepends` fails to resolve your dependencies properly and you need to install things manually. The `remotes` package will be pre-installed for you to use within your script.

Your script will run on a special build queue, which will run even when the cluster is very busy. However, this is restricted in other ways, allowing a maximum of 30 minutes and disallowing parallel running.

Supports one option:

- `script`: The path for the script to run, defaulting to `provision.R`

`renv`:

Uses `renv` to recreate your renv environment. You must be using `renv` locally for this to work, and at present your renv project root must be the same as your hipercow root.

No options are currently supported, but we may pass some renv options in the future; if you need more flexibility here please let us know.

Examples

```
cleanup <- hipercow_example_helper()
writeLines(c("knitr", "data.table"), "pkgdepends.txt")
hipercow_provision()
hipercow_provision_list()

cleanup()
```

`hipercow_provision_compare` *Compare installations*

Description

Compare installations performed into your libraries by conan.

Usage

```
hipercow_provision_compare(curr = 0, prev = -1, driver = NULL, root = NULL)
```

Arguments

`curr`	The previous installation to compare against. Can be a name (see hipercow_provision_list to get names), a negative number where `-n` indicates "`n` installations ago" or a positive number where `n` indicates "the `n`th installation". The default value of 0 corresponds to the current installation.
`prev`	The previous installation to compare against. Can be a name (see hipercow_provision_list to get names), a negative number where `-n` indicates "`n` installations ago" or a positive number where `n` indicates "the `n`th installation". The default of -1 indicates the previous installation. Must refer to an installation before `curr`. Use `NULL` or `-Inf` if you want to compare against the empty installation.
`driver`	The name of the driver to use, or you can leave blank if only one is configured (this will be typical).
`root`	The hipercow root

Value

An object of class `conan_compare`, which can be printed nicely.

Examples

```
cleanup <- hipercow_example_helper()
hipercow_provision("pkgdepends", refs = "knitr")
hipercow_provision("pkgdepends", refs = "data.table")
hipercow_provision_compare()

cleanup()
```

`hipercow_provision_list` *List installations*

Description

List previous successful installations of this hipercow root.

Usage

```
hipercow_provision_list(driver = NULL, root = NULL)

hipercow_provision_check(
  method = NULL,
  ...,
  driver = NULL,
  environment = "default",
  root = NULL
)
```

Arguments

`driver`	The name of the driver to use, or you can leave blank if only one is configured (this will be typical).
`root`	The hipercow root
`method`	The provisioning method to use, defaulting to `NULL`, which indicates we should try and detect the best provisioning mechanism for you; this should typically work well unless you are manually adding packages into your library (see Details). If given, must be one of `auto`, `pkgdepends`, `script` or `renv`; each of these are described in the Details and in `vignette("packages")`.
`...`	Arguments passed through to conan. See Details.
`environment`	The name of the environment to provision (see hipercow_environment_create for details).

Value

A data.frame with columns:

- `name`: the name of the installation. This might be useful with `conan_compare`
- `time`: the time the installation was started
- `hash`: the installation hash
- `method`: the method used for the installation
- `args`: the arguments to the installation (as a list column)
- `current`: if using `hipercow_provision_check`, does this installation match the arguments provided?

This object also has class `conan_list` so that it prints nicely, but you can drop this with `as.data.frame`.

Examples

```
cleanup <- hipercow_example_helper()
writeLines("data.table", "pkgdepends.txt")

# Before any installation has happened:
hipercow_provision_list()
hipercow_provision_check()

# After installation:
hipercow_provision()
hipercow_provision_list()
hipercow_provision_check()

# After a different installation:
hipercow_provision("pkgdepends", refs = "knitr")
hipercow_provision_list()
hipercow_provision_check()

cleanup()
```

`hipercow_purge` *Purge tasks*

Description

Purge (delete) hipercow tasks. This is a destructive operation that cannot be undone and can have unintended consequences! However, if you are running short of space and don't want to just delete everything and start again (which is our general recommendation), this function provides a mechanism for cleaning up tasks that you no longer need.

Usage

```
hipercow_purge(
  task_ids = NULL,
  finished_before = NULL,
  in_bundle = NULL,
  with_status = NULL,
  root = NULL
)
```

Arguments

`task_ids`	A character vector of task identifiers. Typically if you provide this you will not provide any other filters.
`finished_before`	A date, time, or difftime object representing the time or time ago that a task finished (here, the job might have finished for any reason; successfully or unsuccessfully unless you also provide the `with_status` argument). Everything prior to this will be deleted.
`in_bundle`	A character vector of bundle names. Wild cards are supported using shell (glob) syntax, rather than regular expression syntax. So use `data_*` to match all bundles that start with `data_` (see utils::glob2rx for details). It is an error if *no* bundles are matched, but not an error if any individual pattern does not match.
`with_status`	A character vector of statuses to match. We only purge tasks that match these statuses. Valid statuses to use are `created`, `success`, `failure` and `cancelled` (note you cannot select tasks with status of `submitted` or `running`; use task_cancel for these first).
`root`	A hipercow root, or path to it. If `NULL` we search up your directory tree.

Details

Most of the arguments describe *filters* over your tasks. We delete the intersection of these filters (not the union), and you must provide at least one filter. So to delete all tasks that were created more than a week ago you could write:

```
hipercow_purge(created_before = as.difftime(1, units = "weeks"))
```

but to restrict this to only tasks that have *also failed* you could write

```
hipercow_purge(created_before = "1 week", with_status = "failed")
```

Value

A character vector of deleted identifiers, invisibly.

Consequences of deletion

A non-exhaustive list:

- If you delete a task that is part of a task_retry chain, then all tasks (both upstream and downstream in that chain) are deleted
- Once we support task dependencies (mrc-4797), deleting tasks will mark any not-yet-run dependent task as impossible, or perhaps delete it too, or prevent you from deleting the task; we've not decided yet
- You may have a bundle that references a task that you delete, in which case the bundle will not behave as expected. As a result we delete all bundles that reference a deleted task
- Deleted bundles or deleted tasks that you hold identifiers to before deletion will not behave as expected, with tasks reported missing. Restarting your session is probably the safest thing to do after purging.
- We can't prevent race conditions, so if you are purging tasks at the same time you are also retrying tasks that you will purge, you'll create tasks that we might not want to allow, and these tasks will fail in peculiar ways.

Examples

```
cleanup <- hipercow_example_helper()

# Here are some tasks that have finished running:
bundle <- task_create_bulk_expr(sqrt(x), data.frame(x = 1:5),
                                bundle_name = "mybundle")
hipercow_bundle_wait(bundle)

# Purge all tasks contained in any bundle starting with "my":
hipercow_purge(in_bundle = "my*")

cleanup()
```

---

hipercow_resources *Hipercow Resources*

Description

Specify what resources a task requires to run. This creates a validated list of resources that can be passed in as the `resources` argument to task_create_expr or other task creation functions.

Usage

```
hipercow_resources(
  cores = 1L,
  exclusive = FALSE,
  max_runtime = NULL,
  hold_until = NULL,
  memory_per_node = NULL,
```

```
  memory_per_process = NULL,
  requested_nodes = NULL,
  priority = NULL,
  queue = NULL
)
```

Arguments

`cores`	The number of cores your task requires. This is 1 by default. Setting to `Inf` will request any single node single node, however many cores that node has has.
`exclusive`	Set this to `TRUE` to ensure no other tasks will be concurrently run on the node while it runs your task. This is done implicitly if `cores` is `Inf`. This might be useful for a single core task that uses a very large amount of memory, or for multiple tasks that for some reason cannot co-exist on the same node.
`max_runtime`	Set this to specify a time limit for running your job. Acceptable formats are either an integer number of minutes, or strings specifying any combination of hours (h), days (d) and minutes (m). Example valid values: `60`, `"1h30m"`, `"5h"`, or `"40d"`.
`hold_until`	Specify your task should wait in the queue until a certain time, or for a certain period. For the former, this can be a POSIXt (i.e., a date and time in the future), a Date (midnight on a day in the future), the special strings "tonight" (7pm), "midnight", or "weekend" (midnight Saturday morning). To delay for a period, you can specify an integer number of minutes, or strings specifying any combination of hours (h), days (d) and minutes (m). Example valid values: `60`, `"1h30m"`, `"5h"`, or `"3d"`.
`memory_per_node`	Specify your task can only run on a node with at least the specified memory. This is an integer assumed to be gigabytes, or a string in gigabytes or terabytes written as `"64G"` or `"1T"` for example.
`memory_per_process`	If you can provide an estimate of how much RAM your task requires, then the cluster can ensure the total memory required by running multiple tasks on a node does not exceed how much memory the node has. Specify this as an integer number of gigabytes, or characters such as `"10G"`
`requested_nodes`	If you have been in touch with us or DIDE IT, and you need to run your task on a selection of named compute nodes, then specify this here as a vector of strings for the node names.
`priority`	If the tasks you are launching are low priority, you can allow other queuing tasks to jump over them, by setting the priority to to `low`; otherwise, the default is `normal`. These are the only acceptable values.
`queue`	Specify a particular queue to submit your tasks to. This is in development as we decide over time what queues we best need for DIDE's common workflows. See the Details for more information, and the queues available on each cluster.

Value

If the function succeeds, it returns a `hipercow_resources` list of parameters which is syntactically valid, although not yet validated against a particular driver to see if the resources can be satisfied. If the function fails, it will return information about why the arguments could not be validated. Do not modify the return value.

Windows cluster (`wpia-hn`)

- Cores at present must be between 1 and 32
- Memory per node (or per task) can be 512Gb at most.
- The available queues are `AllNodes` and `Training`
- The node names are between `wpia-001` and `wpia-070`, excluding 41, 42, 49 and 50.

Linux cluster (hermod)

Coming Soon.

Examples

```
# The default set of resources
hipercow_resources()

# A more complex case:
hipercow_resources(
  cores = 32,
  exclusive = TRUE,
  priority = "low")

# (remember that in order to change resources you would pass the
# return value here into the "resources" argument of
# task_create_expr() or similar)
```

`hipercow_resources_validate` *Validate a* `hipercow_resources` *list for a driver.*

Description

Query a driver to find information about the cluster, and then validate a hipercow_resources list against that driver to see if the resources requested could be satisfied.

Usage

```
hipercow_resources_validate(resources, driver = NULL, root = NULL)
```

Arguments

resources	A hipercow_resources list returned by hipercow_resources, or NULL
driver	The name of the driver to use, or you can leave blank if only one is configured (this will be typical).
root	The hipercow root

Value

TRUE if the resources are compatible with this driver.

Examples

```
cleanup <- hipercow_example_helper()
hipercow_resources_validate(hipercow_resources(cores = 1))

# This example does not allow more than one core
tryCatch(
  hipercow_resources_validate(hipercow_resources(cores = 32)),
  error = identity)

cleanup()
```

---

hipercow_rrq_controller

Create an rrq controller

Description

Create an rrq controller for your queue, and set it as the default controller. Use this to interact with workers created with `hipercow_rrq_workers_submit()`. Proper docs forthcoming, all interfaces are subject to some change.

Usage

```
hipercow_rrq_controller(..., set_as_default = TRUE, driver = NULL, root = NULL)
```

Arguments

...	Additional arguments passed through to `rrq::rrq_controller()`; currently this is `follow` and `timeout_task_wait`.
set_as_default	Set the rrq controller to be the default; this is usually what you want.
driver	Name of the driver to use. The default (NULL) depends on your configured drivers; if you have no drivers configured we will error as we lack information required to proceed. If you have exactly one driver configured we'll submit your task with it. If you have more than one driver configured, then we will error, though in future versions we may fall back on a default driver if you have one configured.
root	A hipercow root, or path to it. If NULL we search up your directory tree.

Value

An rrq::rrq_controller object.

`hipercow_rrq_workers_submit` *Submit rrq workers*

Description

Submit workers to the cluster, use this in conjunction with hipercow_rrq_controller. A worker may sit on a single core or a whole node depending on how you set up `resources`. We use the `rrq` environment if it exists (hipercow_environment_create) otherwise we'll use the `default` environment.

Usage

```
hipercow_rrq_workers_submit(
  n,
  driver = NULL,
  resources = NULL,
  envvars = NULL,
  parallel = NULL,
  timeout = NULL,
  progress = NULL,
  root = NULL
)
```

Arguments

`n`	The number of workers to submit. This is the only required argument.
`driver`	Name of the driver to use. The default (`NULL`) depends on your configured drivers; if you have no drivers configured we will error as we lack information required to proceed. If you have exactly one driver configured we'll submit your task with it. If you have more than one driver configured, then we will error, though in future versions we may fall back on a default driver if you have one configured.
`resources`	A list generated by hipercow_resources giving the cluster resource requirements to run your task.
`envvars`	Environment variables as generated by hipercow_envvars, which you might use to control your task.
`parallel`	Parallel configuration as generated by hipercow_parallel, which defines which method, if any, will be used to initialise your worker for parallel execution (which means you have to think about parallelism at three levels at least, a diagram may help here).
`timeout`	Time to wait for workers to appear.
`progress`	Should we display a progress bar?
`root`	A hipercow root, or path to it. If `NULL` we search up your directory tree.

Value

A data.frame with information about the launch, with columns:

- `queue_id`: the rrq queue id (same for all workers)
- `worker_id`: the rrq worker identifier
- `task_id`: the hipercow task identifier
- `bundle_name`: the hipercow bundle name (same for all workers)

`hipercow_unconfigure` *Remove a driver from a hipercow configuration*

Description

Remove a driver configured by hipercow_configure. This will not affect tasks already submitted with this driver, but will prevent any future tasks being submitted with it.

Usage

```
hipercow_unconfigure(driver, root = NULL)
```

Arguments

`driver`	The name of the driver to remove
`root`	Hipercow root, usually best `NULL`

Value

Nothing, called for its side effects only.

See Also

hipercow_configuration, which shows currently enabled drivers.

task_cancel *Cancel tasks*

Description

Cancel one or more tasks

Usage

```
task_cancel(id, follow = TRUE, root = NULL)
```

Arguments

id	The task id or task ids to cancel
follow	Logical, indicating if we should follow any retried tasks.
root	A hipercow root, or path to it. If NULL we search up your directory tree.

Value

A logical vector the same length as id indicating if the task was cancelled. This will be FALSE if the task was already completed, not running, etc.

Examples

```
cleanup <- hipercow_example_helper()

ids <- c(task_create_expr(Sys.sleep(2)), task_create_expr(runif(1)))

# The first task may or not be cancelled (depends on if it was
# started already) but the second one will almost certainly be
# cancelled:
task_cancel(ids)

cleanup()
```

task_create_bulk_call *Create bulk tasks from a call*

Description

Create a bulk set of tasks based on applying a function over a vector or data.frame. This is the bulk equivalent of task_create_call, in the same way that task_create_bulk_expr is a bulk version of task_create_expr.

Usage

```
task_create_bulk_call(
  fn,
  data,
  args = NULL,
  environment = "default",
  bundle_name = NULL,
  driver = NULL,
  resources = NULL,
  envvars = NULL,
  parallel = NULL,
  root = NULL
)
```

Arguments

fn	The function to call
data	The data to apply the function over. This can be a vector or list, in which case we act like lapply and apply fn to each element in turn. Alternatively, this can be a data.frame, in which case each row is taken as a set of arguments to fn. Note that if data is a data.frame then all arguments to fn are named.
args	Additional arguments to fn, shared across all calls. These must be named. If you are using a data.frame for data, you'd probably be better off adding additional columns that don't vary across rows, but the end result is the same.
environment	Name of the hipercow environment to evaluate the task within.
bundle_name	Name to pass to hipercow_bundle_create when making a bundle. If NULL we use a random name. We always overwrite, so if bundle_name already refers to a bundle it will be replaced.
driver	Name of the driver to use to submit the task. The default (NULL) depends on your configured drivers; if you have no drivers configured no submission happens (or indeed is possible). If you have exactly one driver configured we'll submit your task with it. If you have more than one driver configured, then we will error, though in future versions we may fall back on a default driver if you have one configured. If you pass FALSE here, submission is prevented even if you have no driver configured.
resources	A list generated by hipercow_resources giving the cluster resource requirements to run your task.
envvars	Environment variables as generated by hipercow_envvars, which you might use to control your task. These will be combined with the default environment variables (see vignettes("details"), this can be overridden by the option hipercow.default_envvars), and any driver-specific environment variables (see vignette("windows")). Variables provided here have the highest precedence. You can **unset** an environment variable by setting it to NA.
parallel	Parallel configuration as generated by hipercow_parallel, which defines which method, if any, will be used to initialise your task for parallel execution.
root	A hipercow root, or path to it. If NULL we search up your directory tree.

Value

A hipercow_bundle object, which groups together tasks, and for which you can use a set of grouped functions to get status (hipercow_bundle_status), results (hipercow_bundle_result) etc.

Examples

```
cleanup <- hipercow_example_helper()

# The simplest way to use this function is like lapply:
x <- runif(5)
bundle <- task_create_bulk_call(sqrt, x)
hipercow_bundle_wait(bundle)
hipercow_bundle_result(bundle) # lapply(x, sqrt)

# You can pass additional arguments in via 'args':
x <- runif(5)
bundle <- task_create_bulk_call(log, x, list(base = 3))
hipercow_bundle_wait(bundle)
hipercow_bundle_result(bundle) # lapply(x, log, base = 3)

# Passing in a data.frame acts like Map (though with all arguments named)
x <- data.frame(a = runif(5), b = rpois(5, 10))
bundle <- task_create_bulk_call(function(a, b) sum(rnorm(b)) / a, x)
hipercow_bundle_wait(bundle)
hipercow_bundle_result(bundle) # Map(f, x$a, x$b)

cleanup()
```

---

task_create_bulk_expr *Create bulk tasks from an expression*

Description

Create a bulk set of tasks. This is an experimental interface and does not have an analogue within didehpc. Variables in data take precedence over variables in the environment in which expr was created. There is no "pronoun" support yet (see rlang docs). Use !! to pull a variable from the environment if you need to, but be careful not to inject something really large (e.g., any vector really) or you'll end up with a revolting expression and poor backtraces. We will likely change some of these semantics later, be careful.

Usage

```
task_create_bulk_expr(
  expr,
  data,
  environment = "default",
  bundle_name = NULL,
```

```
  driver = NULL,
  resources = NULL,
  envvars = NULL,
  parallel = NULL,
  root = NULL
)
```

Arguments

expr	An expression, as for task_create_expr
data	Data that you wish to inject *row-wise* into the expression
environment	Name of the hipercow environment to evaluate the task within.
bundle_name	Name to pass to `hipercow_bundle_create` when making a bundle. If `NULL` we use a random name. We always overwrite, so if `bundle_name` already refers to a bundle it will be replaced.
driver	Name of the driver to use to submit the task. The default (`NULL`) depends on your configured drivers; if you have no drivers configured no submission happens (or indeed is possible). If you have exactly one driver configured we'll submit your task with it. If you have more than one driver configured, then we will error, though in future versions we may fall back on a default driver if you have one configured. If you pass `FALSE` here, submission is prevented even if you have no driver configured.
resources	A list generated by hipercow_resources giving the cluster resource requirements to run your task.
envvars	Environment variables as generated by hipercow_envvars, which you might use to control your task. These will be combined with the default environment variables (see `vignettes("details")`, this can be overridden by the option `hipercow.default_envvars`), and any driver-specific environment variables (see `vignette("windows")`). Variables provided here have the highest precedence. You can **unset** an environment variable by setting it to `NA`.
parallel	Parallel configuration as generated by hipercow_parallel, which defines which method, if any, will be used to initialise your task for parallel execution.
root	A hipercow root, or path to it. If `NULL` we search up your directory tree.

Value

A `hipercow_bundle` object, which groups together tasks, and for which you can use a set of grouped functions to get status (`hipercow_bundle_status`), results (`hipercow_bundle_result`) etc.

See Also

hipercow_bundle_wait, hipercow_bundle_result for working with bundles of tasks

Examples

```
cleanup <- hipercow_example_helper()

# Suppose we have a data.frame:
d <- data.frame(a = 1:5, b = runif(5))

# We can create a "bundle" by applying an expression involving "a"
# and "b":
bundle <- task_create_bulk_expr(sqrt(a * b), d)

# Once you have your bundle, interact with it using the bundle
# analogues of the usual task functions:
hipercow_bundle_wait(bundle)
hipercow_bundle_result(bundle)

cleanup()
```

`task_create_call` *Create task from call*

Description

Create a task based on a function call. This is fairly similar to callr::r, and forms the basis of `lapply()`-like task submission. Sending a call may have slightly different semantics than you expect if you send a closure (a function that binds data), and we may change behaviour here until we find a happy set of compromises. See Details for more on this. The expression `task_create_call(f, list(a, b, c))` is similar to `task_create_expr(f(a, b, c))`, use whichever you prefer.

Usage

```
task_create_call(
  fn,
  args,
  environment = "default",
  driver = NULL,
  resources = NULL,
  envvars = NULL,
  parallel = NULL,
  root = NULL
)
```

Arguments

`fn`	The function to call.
`args`	A list of arguments to pass to the function
`environment`	Name of the hipercow environment to evaluate the task within.

driver	Name of the driver to use to submit the task. The default (NULL) depends on your configured drivers; if you have no drivers configured no submission happens (or indeed is possible). If you have exactly one driver configured we'll submit your task with it. If you have more than one driver configured, then we will error, though in future versions we may fall back on a default driver if you have one configured. If you pass FALSE here, submission is prevented even if you have no driver configured.
resources	A list generated by hipercow_resources giving the cluster resource requirements to run your task.
envvars	Environment variables as generated by hipercow_envvars, which you might use to control your task. These will be combined with the default environment variables (see `vignettes("details")`, this can be overridden by the option `hipercow.default_envvars`), and any driver-specific environment variables (see `vignette("windows")`). Variables provided here have the highest precedence. You can **unset** an environment variable by setting it to NA.
parallel	Parallel configuration as generated by hipercow_parallel, which defines which method, if any, will be used to initialise your task for parallel execution.
root	A hipercow root, or path to it. If NULL we search up your directory tree.

Details

Things are pretty unambiguous when you pass in a function from a package, especially when you refer to that package with its namespace (e.g. `pkg::fn`).

If you pass in the name *without a namespace* from a package that you have loaded with `library()` locally but you have not loaded with `library` within your hipercow environment, we may not do the right thing and you may see your task fail, or find a different function with the same name. We may change the semantics here in a future version to attach your package immediately before running the task.

If you pass in an anonymous function (e.g., `function(x) x + 1`) we may or may not do the right thing with respect to environment capture. We never capture the global environment so if your function is a closure that tries to bind a symbol from the global environment it will not work. Like with `callr::r`, anonymous functions will be easiest to think about where they are fully self contained (i.e., all inputs to the functions come through `args`). If you have bound a *local* environment, we may do slightly better, but semantics here are undefined and subject to change.

R does some fancy things with function calls that we don't try to replicate. In particular you may have noticed that this works:

```
c <- "x"
c(c, c) # a vector of two "x"'s
```

You can end up in this situation locally with:

```
f <- function(x) x + 1
local({
  f <- 1
  f(f) # 2
})
```

this is because when R looks for the symbol for the call it skips over non-function objects. We don't reconstruct environment chains in exactly the same way as you would have locally so this is not possible.

Value

A task id, a string of hex characters. Use this to interact with the task.

Examples

```
cleanup <- hipercow_example_helper()

# Similar to the example in task_create_call
id <- task_create_call(stats::runif, list(5))
task_info(id)
task_wait(id)
task_result(id)

# Unlike task_create_explicit, variables are automatically included:
id <- task_create_call(function(x, y) x + y, list(2, 5))
task_info(id)
task_wait(id)
task_result(id)

cleanup()
```

---

task_create_explicit *Create explicit task*

Description

Create an explicit task. Explicit tasks are the simplest sort of task in hipercow and do nothing magic. They accept an R expression (from `quote` or friends) and possibly a set of variables to export from the global environment. This can then be run on a cluster by loading your variables and running your expression. If your expression depends on packages being *attached* then you should pass a vector of package names too. This function may disappear, and is used by us to think about the package, it's not designed to really be used.

Usage

```
task_create_explicit(
  expr,
  export = NULL,
  envir = parent.frame(),
  environment = "default",
  driver = NULL,
  resources = NULL,
  envvars = NULL,
  parallel = NULL,
```

```
  root = NULL
)
```

Arguments

expr	Unevaluated expression object, e.g., from quote
export	Optional character vector of names of objects to export into the evaluating environment
envir	Local R environment in which to find variables for export. The default is the parent frame, which will often do the right thing. Another sensible choice is .GlobalEnv to use the global environment.
environment	Name of the hipercow environment to evaluate the task within.
driver	Name of the driver to use to submit the task. The default (NULL) depends on your configured drivers; if you have no drivers configured no submission happens (or indeed is possible). If you have exactly one driver configured we'll submit your task with it. If you have more than one driver configured, then we will error, though in future versions we may fall back on a default driver if you have one configured. If you pass FALSE here, submission is prevented even if you have no driver configured.
resources	A list generated by hipercow_resources giving the cluster resource requirements to run your task.
envvars	Environment variables as generated by hipercow_envvars, which you might use to control your task. These will be combined with the default environment variables (see vignettes("details"), this can be overridden by the option hipercow.default_envvars), and any driver-specific environment variables (see vignette("windows")). Variables provided here have the highest precedence. You can **unset** an environment variable by setting it to NA.
parallel	Parallel configuration as generated by hipercow_parallel, which defines which method, if any, will be used to initialise your task for parallel execution.
root	A hipercow root, or path to it. If NULL we search up your directory tree.

Value

A task id, a string of hex characters. Use this to interact with the task.

Examples

```
cleanup <- hipercow_example_helper()

# About the most simple task that can be created:
id <- task_create_explicit(quote(sqrt(2)))
task_wait(id)
task_result(id)

# Variables are not automatically included with the expression:
a <- 5
id <- task_create_explicit(quote(sqrt(a)))
task_info(id)
```

```
task_wait(id)
task_result(id)

# Include variables by passing them via 'export':
id <- task_create_explicit(quote(sqrt(a)), export = "a")
task_info(id)
task_wait(id)
task_result(id)

cleanup()
```

`task_create_expr` *Create a task based on an expression*

Description

Create a task based on an expression. This is similar to task_create_explicit except more magic, and is closer to the interface that we expect people will use.

Usage

```
task_create_expr(
  expr,
  environment = "default",
  driver = NULL,
  resources = NULL,
  envvars = NULL,
  parallel = NULL,
  root = NULL
)
```

Arguments

`expr`	The expression, does not need quoting. See Details.
`environment`	Name of the hipercow environment to evaluate the task within.
`driver`	Name of the driver to use to submit the task. The default (`NULL`) depends on your configured drivers; if you have no drivers configured no submission happens (or indeed is possible). If you have exactly one driver configured we'll submit your task with it. If you have more than one driver configured, then we will error, though in future versions we may fall back on a default driver if you have one configured. If you pass `FALSE` here, submission is prevented even if you have no driver configured.
`resources`	A list generated by hipercow_resources giving the cluster resource requirements to run your task.

envvars	Environment variables as generated by hipercow_envvars, which you might use to control your task. These will be combined with the default environment variables (see `vignettes("details")`, this can be overridden by the option `hipercow.default_envvars`), and any driver-specific environment variables (see `vignette("windows")`). Variables provided here have the highest precedence. You can **unset** an environment variable by setting it to `NA`.
parallel	Parallel configuration as generated by hipercow_parallel, which defines which method, if any, will be used to initialise your task for parallel execution.
root	A hipercow root, or path to it. If `NULL` we search up your directory tree.

Details

The expression passed as `expr` will typically be a function call (e.g., `f(x)`). We will analyse the expression and find all variables that you reference (in the case of `f(x)` this is `x`) and combine this with the function name to run on the cluster. If `x` cannot be found in your calling environment we will error; this behaviour is subject to change so let us know if you have other thoughts.

Alternatively you may provide a multiline statement by using `{}` to surround multiple lines, such as:

```
task_create_expr({
  x <- runif(1)
  f(x)
}, ...)
```

in this case, we apply a simple heuristic to work out that `x` is locally assigned and should not be saved with the expression.

If you reference values that require a lot of memory, `hipercow` will error and refuse to save the task. This is to prevent you accidentally including values that you will make available through an environment, and to prevent making the `hipercow` directory excessively large. Docs on controlling this process are still to be written.

Value

A task id, a string of hex characters. Use this to interact with the task.

Examples

```
cleanup <- hipercow_example_helper()

# Similar to task_create_explicit, but we don't include the 'quote'
id <- task_create_expr(runif(5))
task_wait(id)
task_result(id)

# Unlike task_create_explicit, variables are automatically included:
n <- 3
id <- task_create_expr(runif(n))
task_info(id)
task_wait(id)
```

```
task_result(id)

cleanup()
```

`task_create_script` *Create script task*

Description

Create a task from a script. This will arrange to run the file `script` via hipercow. The script must exist within your hipercow root, but you may change to the directory of the script as it executes (otherwise we will evaluate from your current directory relative to the hipercow root, as usual).

Usage

```
task_create_script(
  script,
  chdir = FALSE,
  echo = TRUE,
  environment = "default",
  driver = NULL,
  resources = NULL,
  envvars = NULL,
  parallel = NULL,
  root = NULL
)
```

Arguments

`script`	Path for the script
`chdir`	Logical, indicating if we should change the working directory to the directory containing `script` before executing it (similar to the `chdir` argument to `source`).
`echo`	Passed through to `source` to control printing while evaluating. Generally you will want to leave this as `TRUE`
`environment`	Name of the hipercow environment to evaluate the task within.
`driver`	Name of the driver to use to submit the task. The default (`NULL`) depends on your configured drivers; if you have no drivers configured no submission happens (or indeed is possible). If you have exactly one driver configured we'll submit your task with it. If you have more than one driver configured, then we will error, though in future versions we may fall back on a default driver if you have one configured. If you pass `FALSE` here, submission is prevented even if you have no driver configured.
`resources`	A list generated by hipercow_resources giving the cluster resource requirements to run your task.

envvars	Environment variables as generated by hipercow_envvars, which you might use to control your task. These will be combined with the default environment variables (see `vignettes("details")`, this can be overridden by the option `hipercow.default_envvars`), and any driver-specific environment variables (see `vignette("windows")`). Variables provided here have the highest precedence. You can **unset** an environment variable by setting it to `NA`.
parallel	Parallel configuration as generated by hipercow_parallel, which defines which method, if any, will be used to initialise your task for parallel execution.
root	A hipercow root, or path to it. If `NULL` we search up your directory tree.

Value

A task id, a string of hex characters. Use this to interact with the task.

Examples

```
cleanup <- hipercow_example_helper()

# Create a small script; this would usually be several lines of
# course.  The script will need to do something as a side effect
# to be worth calling, so here we write a file.
writeLines("saveRDS(mtcars, 'data.rds')", "script.R")

# Now create a task from this script
id <- task_create_script("script.R")
task_info(id)
task_wait(id)
task_result(id)
dir()

cleanup()
```

---

task_eval *Run a task*

Description

Run a task that has been created by a `task_create_*` function, e.g., `task_create_explicit()`, `task_create_expr()`. Generally users should not run this function directly.

Usage

```
task_eval(id, envir = .GlobalEnv, verbose = FALSE, root = NULL)
```

Arguments

id	The task identifier
envir	An environment in which to evaluate the expression. For non-testing purposes, generally ignore this, the global environment will be likely the expected environment.
verbose	Logical, indicating if we should print information about what we do as we do it.
root	A hipercow root, or path to it. If NULL we search up your directory tree.

Value

Logical indicating success (TRUE) or failure (FALSE)

Examples

```
cleanup <- hipercow_example_helper(runner = FALSE)
id <- task_create_expr(runif(1), driver = FALSE)
# Status is only 'created', not 'submitted', as we did not submit
# task.  This task can never run.
task_status(id)

# Explicitly evaluate the task:
task_eval(id, verbose = TRUE)
task_result(id)

cleanup()
```

---

task_info *Fetch task information*

Description

Fetch information about a task. This is much more detailed than the information in task_status. If a task is running we also fetch the true status via its driver, which can be slower.

Usage

```
task_info(id, follow = TRUE, root = NULL)
```

Arguments

id	A single task id to fetch information for
follow	Logical, indicating if we should follow any retried tasks.
root	A hipercow root, or path to it. If NULL we search up your directory tree.

Value

An object of class `hipercow_task_info`, which will print nicely. This is just a list with elements:

- `id`: the task identifier
- `status`: the retrieved status
- `driver`: the driver used to run the task (or NA)
- `data`: the task data (depends on the type of task)
- `times`: a vector of times
- `retry_chain`: the retry chain (or `NULL`)

You can see and access these elements more easily by running `unclass()` on the result of `task_info()`.

Examples

```
cleanup <- hipercow_example_helper()
id <- task_create_expr(runif(1))
task_wait(id)

# Task information at completion includes times:
task_info(id)

# If you need to work with these times, use the "times" element:
task_info(id)$times

# If a task is retried, this information appears as a retry chain:
id2 <- task_retry(id)
task_info(id2, follow = FALSE)
task_info(id2)

cleanup()
```

`task_log_show` *Get task log*

Description

Get the task log, if the task has produced one. Tasks run by the `windows` driver will generally produce a log. A log might be quite long, and you might want to print it to screen in its entirety (`task_log_show`), or return it as character vector (`task_log_value`).

Usage

```
task_log_show(id, outer = FALSE, follow = TRUE, root = NULL)

task_log_value(id, outer = FALSE, follow = TRUE, root = NULL)

task_log_watch(
```

```
  id,
  poll = 1,
  skip = 0,
  timeout = NULL,
  progress = NULL,
  follow = TRUE,
  root = NULL
)
```

Arguments

id	The task identifier
outer	Logical, indicating if we should request the "outer" logs; these are logs from the underlying HPC software before it hands off to hipercow.
follow	Logical, indicating if we should follow any retried tasks.
root	A hipercow root, or path to it. If NULL we search up your directory tree.
poll	Time, in seconds, used to throttle calls to the status function. The default is 1 second
skip	Optional integer indicating how to handle log content that exists at the point where we start watching. The default (0) shows all log contents. A positive integer skips that many lines, while a negative integer shows only that many lines (so -5 shows the first five lines in the log). You can pass Inf to discard all previous logs, but stream all new ones.
timeout	The time to wait for the task to complete. The default is to wait forever.
progress	Logical value, indicating if a progress spinner should be used. The default NULL uses the option hipercow.progress, and if unset displays a progress bar in an interactive session.

Details

The function task_log_watch has similar semantics to task_wait but does not error on timeout, and always displays a log.

Value

Depending on the function:

- task_log_show returns the log value contents invisibly, but primarily displays the log contents on the console as a side effect
- task_log_value returns a character of log contents
- task_log_watch returns the status converted to logical (as for task_wait)

Examples

```
cleanup <- hipercow_example_helper(with_logging = TRUE)

# Tasks that don't produce any output (print, cat, warning, etc)
```

```
# will only contain logging information from hipercow itself
id <- task_create_expr(runif(1))
task_wait(id)
task_log_show(id)

# If your task creates output then it will appear within the
# horizontal rules:
id <- task_create_expr({
  message("Starting analysis")
  x <- mean(runif(100))
  message("all done!")
  x
})
task_wait(id)
task_log_show(id)

# Use "task_log_value" to get the log value as a character vector
task_log_value(id)

# Depending on the driver you are using, there may be useful
# information in the "outer" log; the logs produced by the
# submission system before hipercow takes over:
task_log_show(id, outer = TRUE)

cleanup()
```

`task_result` *Get task result*

Description

Get the task result. This might be an error if the task has failed.

Usage

```
task_result(id, follow = TRUE, root = NULL)
```

Arguments

`id`	The task identifier
`follow`	Logical, indicating if we should follow any retried tasks.
`root`	A hipercow root, or path to it. If `NULL` we search up your directory tree.

Value

The value of the queued expression

Examples

```
cleanup <- hipercow_example_helper()

# Typical usage
id <- task_create_expr(runif(1))
task_wait(id)
task_result(id)

# Tasks that error return error values as results
id <- task_create_expr(readRDS("nosuchfile.rds"))
task_wait(id)
task_result(id)

cleanup()
```

`task_retry` *Retry a task*

Description

Retry one or more tasks. This creates a new task that copies the work of the old one. Most of the time this is transparent. We'll document this in the "advanced" vignette once it's written.

Usage

```
task_retry(id, driver = NULL, resources = NULL, root = NULL)
```

Arguments

`id`	The identifier or identifiers of tasks to retry.
`driver`	Name of the driver to use to submit the task. The default (`NULL`) depends on your configured drivers; if you have no drivers configured no submission happens (or indeed is possible). If you have exactly one driver configured we'll submit your task with it. If you have more than one driver configured, then we will error, though in future versions we may fall back on a default driver if you have one configured. If you pass `FALSE` here, submission is prevented even if you have no driver configured.
`resources`	A list generated by hipercow_resources giving the cluster resource requirements to run your task.
`root`	A hipercow root, or path to it. If `NULL` we search up your directory tree.

Details

This ends up being a little more complicated than ideal in order to keep things relatively fast, while keeping our usual guarantees about race conditions etc. Basically; retrying is the only way a task can move out of a terminal state but it still does not modify the existing task. Instead, we keep a separate register of whether a task has been retried or not. Each time we retry we write into this

register. When you query about the status etc of a task you can then add a `follow` argument to control whether or not we check the register. We assume that you never call this in parallel; if you do then retries may be lost. You can run `task_retry(NULL)` to refresh the cached copy of the retry map if you need to.

Value

New identifiers for the retried tasks

Examples

```
cleanup <- hipercow_example_helper()

# For demonstration, we just generate random numbers as then it's
# more obvious that things have been rerun:
id1 <- task_create_expr(runif(1))
task_wait(id1)
task_result(id1)

# Now retry the task and get the retried result:
id2 <- task_retry(id1)
task_wait(id2)
task_result(id2)

# After a retry, both the original and derived tasks know about
# each other:
task_info(id1)
task_info(id2)

# By default every task will "follow" and access the most recent
# task in the chain:
task_result(id1) == task_result(id2)

# You can prevent this by passing follow = FALSE to get the value
# of this particular attempt:
task_result(id1, follow = FALSE)

# Tasks can be retried as many times as needed, creating a
# chain. It does not matter which task you retry as we always
# follow all the way to the end of the chain before retrying:
id3 <- task_retry(id1)
task_info(id1, follow = FALSE)
task_info(id3)

cleanup()
```

`task_status` *Get task status*

Description

Get the status of a task. See Details for the lifecycle.

Usage

```
task_status(id, follow = TRUE, root = NULL)
```

Arguments

id	The task identifier
follow	Logical, indicating if we should follow any retried tasks.
root	A hipercow root, or path to it. If NULL we search up your directory tree.

Details

A task passes through a lifecycle:

- created
- submitted
- running
- success, failure, cancelled

These occur in increasing order and the result of this function is the furthest through this list.

Later, we will introduce other types to cope with tasks that are blocked on dependencies (or have become impossible due to failed dependencies).

Value

A string with the task status. Tasks that do not exist will have a status of NA.

Examples

```
cleanup <- hipercow_example_helper()

ids <- c(task_create_expr(runif(1)), task_create_expr(runif(1)))
# Depending on how fast these tasks get picked up they will be one
# of 'submitted', 'running' or 'success':
task_status(ids)

# Wait until both tasks are complete
task_wait(ids[[1]])
task_wait(ids[[2]])
# And both are success now
task_status(ids)

cleanup()
```

task_submit *Submit a task*

Description

Submit a task to a queue. This is a lower-level function that you will not often need to call. Typically a task will be submitted automatically to your driver on creation (e.g., with `task_create_expr()`), unless you specified `driver = FALSE` or you had not yet configured a driver.

Usage

```
task_submit(id, ..., resources = NULL, driver = NULL, root = NULL)
```

Arguments

`id`	A vector of task ids
`...`	Disallowed additional arguments, don't use.
`resources`	A list generated by `hipercow_resources` giving the cluster resource requirements to run your task.
`driver`	The name of the driver to use, or you can leave blank if only one is configured (this will be typical).
`root`	The hipercow root

task_wait *Wait for a task to complete*

Description

Wait for a single task to complete (or to start). This function is very similar to task_log_watch, except that it errors if the task does not complete (so that it can be used easily to ensure a task has completed) and does not return any logs.

Usage

```
task_wait(
  id,
  for_start = FALSE,
  timeout = NULL,
  poll = 1,
  progress = NULL,
  follow = TRUE,
  root = NULL
)
```

Arguments

id	The task identifier
for_start	Logical value, indicating if we only want to wait for the task to *start* rather than complete. This will block until the task moves away from submitted, and will return when it takes the status running or any terminal status (success, failure, cancelled). Note that this does not guarantee that your task will still be running by the time task_wait exits, your task may have finished by then!
timeout	The time to wait for the task to complete. The default is to wait forever.
poll	Time, in seconds, used to throttle calls to the status function. The default is 1 second
progress	Logical value, indicating if a progress spinner should be used. The default NULL uses the option hipercow.progress, and if unset displays a progress bar in an interactive session.
follow	Logical, indicating if we should follow any retried tasks.
root	A hipercow root, or path to it. If NULL we search up your directory tree.

Details

The progress spinners here come from the cli package and will respond to cli's options. In particular cli.progress_clear and cli.progress_show_after.

Value

Logical value, TRUE if the task completed successfully, FALSE otherwise.

Examples

```
cleanup <- hipercow_example_helper()

id <- task_create_expr(sqrt(2))
task_wait(id)

cleanup()
```

---

windows_authenticate *DIDE windows credentials*

Description

Register DIDE windows credentials.

Usage

```
windows_authenticate()
```

Details

In order to be able to communicate with the Windows DIDE HPC system, we need to be able to communicate with the HPC portal (https::mrcdata.dide.ic.ac.uk/hpc), and for this we need your **DIDE** password and username. This is typically, but not always, the same as your Imperial credentials. We store this information securely using the keyring package, so when unlocking your credentials you will be prompted for your **computer** password, which will be your DIDE password if you use a windows machine connected to the DIDE domain, but will likely differ from either your DIDE or Imperial password if you are outside the DIDE domain, or if you don't use Windows.

Value

Nothing, this function is called for its side effect of setting or updating your credentials within the keyring.

Examples

```
windows_authenticate()
```

`windows_check` *Check we can use windows cluster*

Description

Perform some basic checks to make that your system is configured to use the windows cluster properly. Calling this when something goes wrong is never a bad idea.

Usage

```
windows_check(path = getwd())
```

Arguments

`path`	Path to check; typically this will be your working directory.

Value

Invisibly, a logical; `TRUE` if all checks succeed and `FALSE` otherwise.

Examples

```
windows_check()
```

windows_generate_keypair *Generate keypair*

Description

Generate a keypair for encrypting small data to send to the windows cluster. This can be used to encrypt environment variables, and possibly other workflows in future. By default, if you have ever created a keypair we do not replace it if it already exists, unless you set `update = TRUE` so you may call this function safely to ensure that you do have a keypair set up.

Usage

```
windows_generate_keypair(update = FALSE)
```

Arguments

`update`	Replace the existing keypair. You will need to use this if you accidentally remove the `.hipercow/` directory from your network home share, or if you want to renew your key.

Value

Nothing, called for its side effect

Examples

```
# Generate a new keypair, if one does not exist
windows_generate_keypair()
```

windows_path *Describe a path mapping*

Description

Describe a path mapping for use when setting up jobs on the cluster.

Usage

```
windows_path(path_local, path_remote, drive_remote, call = NULL)
```

Arguments

path_local	The point where the drive is attached locally. On Windows this will be something like "Q:/", on Mac something like "/Volumes/mountname", and on Linux it could be anything at all, depending on what you used when you mounted it (or what is written in /etc/fstab)
path_remote	The network path for this drive. It will look something like \\\\fi--didef3.dide.ic.ac.uk\\tmp\\. Unfortunately backslashes are really hard to get right here and you will need to use twice as many as you expect (so *four* backslashes at the beginning and then two for each separator). If this makes you feel bad know that you are not alone: https://xkcd.com/1638 – alternatively you may use forward slashes in place of backslashes (e.g. //fi--didef3.dide.ic.ac.uk/tmp)
drive_remote	The place to mount the drive on the cluster. We're probably going to mount things at Q: and T: already so don't use those. And things like C: are likely to be used. Perhaps there are some guidelines for this somewhere?
call	The name of the calling function, for error reporting.

Examples

```
# Suppose you have mounted your malaria share at "~/net/malaria"
# (e.g., on a Linux machine).  You can tell the cluster to mount
# this as "M:" when running tasks by first creating a path
# mapping:
share <- windows_path("~/net/malaria",
                      "//fi--didenas1.dide.ic.ac.uk/Malaria",
                      "M:")

# This share object contains information about how to relate your
# local and remote paths:
share

# When configuring the cluster you might pass this:
hipercow_configure("windows", shares = share)
```

---

windows_username *Report windows username*

Description

Report the username used to log into the web portal for use with the windows cluster. This may or may not be the same as your local username. We may ask you to run this when helping debug cluster failures.

Usage

```
windows_username()
```

Value

Your username, as a string

Examples

```
# Return your windows username
windows_username()
```

Index