unidist

Release 0.5.0+6.g4fa3cf2.dirty

unidist contributors

Jan 08, 2024

CONTENTS

unidist (*Unified Distributed Execution*) is a framework that is intended to provide the unified API for distributed execution by supporting various performant execution backends. At the moment the following backends are supported under the hood:

- MPI
- Dask Distributed
- Ray
- Python Multiprocessing

Also, the framework provides a Python Sequential backend (pyseq), that can be used for debugging.

unidist is designed to work in a task-based parallel model. The framework mimics Ray API and expands the existing frameworks (Ray and Dask Distributed) with additional features.

CHAPTER

ONE

QUICK START GUIDE

1.1 Installation

To install the most recent stable release for unidist run the following:

```
pip install unidist[all] # Install unidist with dependencies for all the backends
```

For further instructions on how to install unidist with concrete execution backends or using conda see our *Installation* section.

1.2 Usage

The example below describes squaring the numbers from a list using unidist:

```
# script.py
if __name__ == "__main__":
   import unidist

   unidist.init() # Initialize unidist's backend. MPI is used by default.

   @unidist.remote # Apply a decorator to make `foo` a remote function.
   def foo(x):
      return x * x

   # This will run `foo` on a pool of workers in parallel;
   # `refs` will contain object references to actual data.
   refs = [foo.remote(i) for i in range(4)]

   # Get materialized data.
   print(unidist.get(refs)) # [0, 1, 4, 9]
```

Run the *script.py* with:

```
$ mpiexec -n 1 python script.py  # for MPI backend
# python script.py  # for any other supported backend
```

1.2.1 Installation

There are several ways to install unidist. Most users will want to install with `pip` or using `conda` tool, but some users may want to build from the master branch on the GitHub repo. The master branch has the most recent patches, but may be less stable than a release installed from `pip` or `conda`.

Installing with pip

Stable version

unidist can be installed with `pip` on Linux, Windows and MacOS. To install the most recent stable release run the following:

```
pip install unidist # Install unidist with dependencies for Python Multiprocessing and␣
↪Python Sequential backends
```

unidist can also be used with MPI, Dask or Ray execution backend. If you don't have MPI, Dask or Ray installed, you will need to install unidist with one of the targets:

```
pip install unidist[all] # Install unidist with dependencies for all the backends
pip install unidist[mpi] # Install unidist with dependencies for MPI backend
pip install unidist[dask] # Install unidist with dependencies for Dask backend
pip install unidist[ray] # Install unidist with dependencies for Ray backend
```

unidist automatically detects which execution backends are installed and uses that for scheduling computation!

Note: There are different MPI implementations, each of which can be used as a backend in unidist. Mapping `unidist[mpi]` installs `mpi4py` package, which is just a Python wrapper for MPI. To enable unidist on MPI execution you need to have a working MPI implementation and certain software installed beforehand. Refer to installation page of the *mpi4py* documentation for details. Also, you can find some instructions on *MPI backend* page.

Release candidates

unidist follows Semantic Versioning and before some major or minor releases, we will upload a release candidate to test and check if there are any problems. If you would like to install a pre-release of unidist, run the following:

```
pip install --pre unidist
```

These pre-releases are uploaded for dependencies and users to test their existing code to ensure that it still works. If you find something wrong, please raise an issue.

Installing with conda

Using conda-forge channel

unidist releases can be installed using `conda` from the conda-forge channel. Starting from the first 0.1.0 release it is possible to install unidist with chosen execution backend(s) alongside. Current options are:

Package name in conda-forge	Backend(s)	Supported OSs
unidist	Python Multiprocessing, Python Sequential	Linux, Windows, MacOS
unidist-all	MPI, Dask, Ray, Python Multiprocessing, Python Sequential	Linux, Windows
unidist-mpi	MPI	Linux, Windows, MacOS
unidist-dask	Dask	Linux, Windows, MacOS
unidist-ray	Ray	Linux, Windows

For installing unidist with dependencies for MPI and Dask execution backends into a conda environment the following command should be used:

```
conda install unidist-mpi unidist-dask -c conda-forge
```

All set of backends could be available in a conda environment by specifying:

```
conda install unidist-all -c conda-forge
```

or explicitly:

```
conda install unidist-mpi unidist-dask unidist-ray -c conda-forge
```

Note: There are different MPI implementations, each of which can be used as a backend in unidist. By default, mapping `unidist-mpi` installs a default MPI implementation, which comes with `mpi4py` package and is ready to use. The conda dependency solver decides on which MPI implementation is to be installed. If you want to use a specific version of MPI, you can install the core dependencies for MPI backend and the specific version of MPI as `conda install unidist-mpi <mpi>` as shown in the installation page of `mpi4py` documentation. That said, it is highly encouraged to use your own MPI binaries as stated in the Using External MPI Libraries section of the conda-forge documentation in order to get ultimate performance.

Using intel channel

Conda `intel` channel contains a performant MPI implementaion, which can be used in the unidist MPI backend instead of an MPI implementation from `conda-forge` channel. To install Intel MPI you should use the following:

```
conda install unidist -c conda-forge
conda install mpi4py -c intel
```

Installing from the GitHub master branch

If you'd like to try unidist using the most recent updates from the master branch, you can also use `pip`.

```
# Install unidist with dependencies for Python Multiprocessing and Python Sequential␣
↪backends
pip install git+https://github.com/modin-project/unidist
# Install unidist with dependencies for all the backends
pip install git+https://github.com/modin-project/unidist#egg=unidist[all]
# Install unidist with dependencies for MPI backend
pip install git+https://github.com/modin-project/unidist#egg=unidist[mpi]
```

This will install directly from the repo without you having to manually clone it! Please be aware that these changes have not made it into a release and may not be completely stable.

Building unidist from Source

If you're planning to *contribute* to unidist, you need to ensure that you are building unidist from the local repository that you are working of. Occasionally, there are issues in overlapping unidist installs from PyPI and from source. To avoid these issues, we recommend uninstalling unidist before installation from source:

```
pip uninstall unidist
```

To build from source, you first must clone the repo. We recommend forking the repository first through the GitHub interface, then cloning as follows:

```
git clone https://github.com/<your-github-username>/unidist.git
```

Once cloned, `cd` into the `unidist` directory and use `pip` to install:

```
cd unidist
# Install unidist with dependencies for Python Multiprocessing and Python Sequential␣
↪backends
pip install -e .
# Install unidist with dependencies for all the backends
pip install -e .[all]
# Install unidist with dependencies for MPI backend
pip install -e .[mpi]
```

1.2.2 Getting Started

unidist provides *the high-level API* to make distributed applications. To tune unidist's behavior the user has several methods described in unidist configuration settings section.

Using unidist API

The example below shows how to use unidist API to make parallel execution for functions (tasks) and classes (actors).

```
# script.py

if __name__ == "__main__":
   import unidist.config as cfg
   import unidist

   # Initialize unidist's backend. The MPI backend is used by default.
   unidist.init()

   # Apply decorator to make `square` a remote function.
   @unidist.remote
   def square(x):
      return x * x

   # Asynchronously execute remote function.
   square_refs = [square.remote(i) for i in range(4)]

   # Apply decorator to make `Counter` actor class.
   @unidist.remote
   class Cube:
      def __init__(self):
            self.volume = None

      def compute_volume(self, square):
            self.volume = square ** 1.5

      def read(self):
            return self.volume

   # Create instances of the actor class.
   cubes = [Cube.remote() for _ in range(len(square_refs))]
   # Asynchronously execute methods of the actor class.
   [cube.compute_volume.remote(square) for cube, square in zip(cubes, square_refs)]
   cube_refs = [cube.read.remote() for cube in cubes]

   # Get materialized results.
   print(unidist.get(square_refs)) # [0, 1, 4, 9]
   print(unidist.get(cube_refs)) # [0.0, 1.0, 8.0, 27.0]
```

Choosing unidist's backend

The examples below use the `UNIDIST_BACKEND` environment variable to set the execution backend:

```
# Running the script with unidist on MPI backend
$ export UNIDIST_BACKEND=mpi
$ mpiexec -n 1 python script.py
# Running the script with unidist on Dask backend
$ export UNIDIST_BACKEND=dask
```

(continues on next page)

(continued from previous page)

```
$ python script.py
# Running the script with unidist on Ray backend
$ export UNIDIST_BACKEND=ray
$ python script.py
```

You probably noticed one specific thing when using the MPI backend to run the script, namely, the use of `mpiexec` command. Currently, almost all MPI implementations require `mpiexec` command to be used when running an MPI program.

1.2.3 Using Unidist

This page contains information on how to choose a concrete execution backend and run unidist with it.

- Unidist on MPI
- Unidist on Dask
- Unidist on Ray
- Unidist on PyMp
- Unidist on PySeq

Libraries powered by Unidist

Here you can find information on which libraries have already been integrated with unidist to use its performant backends to get better performance.

- **Modin**
  - Refer to Using pandas on Unidist page of the Modin documentation on how to get started with Modin on unidist.

1.2.4 Optimization Notes

This page contains optimization notes for every backend that can be applied to it to get ultimate performance.

MPI backend

We highly encourage to use external (either custom-built or system-provided) MPI installations in production scenarious to get ultimate performance.

Open Source MPI implementaions

Building MPI from source

In Building MPI from source section of `mpi4py` documentation you can find executive instructions for building some of the open-source MPI implementations out there with support for shared/dynamic libraries on POSIX environments.

Once you have a working MPI implementation, you will have to adapt your `PATH` environment variable to use the installed MPI version.

```
export PATH=path/to/mpi/bin:$PATH
```

Then, you can install `unidist` and the required dependencies for the MPI backend.

```
pip install unidist[mpi]
```

Now you can use unidist on MPI backend using the installed MPI implementaion.

Proprietary MPI implementations

Intel MPI From Intel oneAPI HPC Toolkit

The following instructions will help you install Intel MPI from Intel oneAPI HPC Toolkit to use it as the unidist's backend. We will use an offline installer an an example but you are free to use other installation options.

1. Create a directory for installing Intel MPI and go into it. You can do this in a terminal by typing

```
mkdir local
cd local
```

2. Download a toolkit installer from Intel oneAPI HPC Toolkit, e.g., using `wget` command

```
wget https://registrationcenter-download.intel.com/akdlm/IRC_NAS/1ff1b38a-8218-4c53-9956-
↪f0b264de35a4/l_HPCKit_p_2023.1.0.46346_offline.sh
```

Note that we use the specific version of the toolkit as an example. You can install any version you want.

3. Launch the installer

```
sh ./l_HPCKit_p_2023.1.0.46346_offline.sh
```

During installation process you can choose a directory in which the toolkit should be installed (e.g., `path/to/local/<toolkit>`).

4. Source the `setvars.sh` (global to the toolkit) or the `vars.sh` (local to the Intel MPI)

```
# source path/to/local/<toolkit>/oneapi/setvars.sh
source path/to/local/<toolkit>/oneapi/mpi/latest/env/vars.sh
```

5. Install `unidist` and the required dependencies for the MPI backend

```
pip install unidist[mpi]
```

Now you can use unidist on MPI backend using Intel MPI implementaion.

6. Remove the installer (optional):

```
rm l_HPCKit_p_2023.1.0.46346_offline.sh
```

Dask backend

Unidist is just a wrapper for Dask so to get ultimate performance for this backend you should read through `Build Understanding` section of Dask documentation to get more insights with respect to performance improvements.

Ray backend

Unidist is just a wrapper for Ray so to get ultimate performance for this backend you should follow Ray's Design Patterns & Anti-patterns.

PyMp backend

Coming soon…

PySeq backend

Python Sequential backend is for debugging so we do not provide optimization notes for this backend.

- *MPI*
- *Dask*
- *Ray*
- *PyMp*
- *PySeq*

1.2.5 Unidist Architecture

High-Level Execution View

The diagram below outlines the high-level view to the execution flow of unidist.

Unidist High-Level API

This page contains the API provided by the framework for distributed execution of operations.

init

`unidist.api.init()`

Initialize an execution backend.

Notes

The concrete execution backend can be set via *UNIDIST_BACKEND* environment variable or `Backend` config value. MPI backend is used by default.

is_initialized

`unidist.api.`**`is_initialized`**`()`

Check if a unidist backend has already been initialized.

Returns
True or False.

Return type
bool

remote

`unidist.api.`**`remote`**`(`**args*, ***kwargs*`)`

Define a remote function or an actor class.

Parameters

- **`*args`** (*`iterable`*) – Positional arguments to be passed in a remote function or an actor class.
- **`**kwargs`** (*`dict`*) – Keyword arguments to be passed in a remote function or an actor class.

Return type
RemoteFunction or ActorClass

get

`unidist.api.`**`get`**`(`*object_refs*`)`

Get a remote object or a list of remote objects from distributed memory.

Parameters
`object_refs` (*`ObjectRef or list`*) – `ObjectRef` or a list of `ObjectRef`-s to get data from.

Returns
A Python object or a list of Python objects.

Return type
object

put

`unidist.api.put(data)`

Put data into distributed memory.

Parameters
`data` (*object*) – Data to be put.

Returns
`ObjectRef` matching to data.

Return type
ObjectRef

wait

`unidist.api.wait(object_refs, num_returns=1)`

Wait until *object_refs* are finished.

This method returns two lists. The first list consists of `ObjectRef`-s that correspond to objects that completed computations. The second list corresponds to the rest of the `ObjectRef`-s (which may or may not be ready).

Parameters

- `object_refs` (*ObjectRef or list*) – `ObjectRef` or list of `ObjectRef`-s to be waited.
- `num_returns` (*int, default: 1*) – The number of `ObjectRef`-s that should be returned as ready.

Returns
List of `ObjectRef`-s that are ready and list of the remaining `ObjectRef`-s.

Return type
two lists

is_object_ref

`unidist.api.is_object_ref(obj)`

Whether an object is `ObjectRef` or not.

Parameters
`obj` (*object*) – An object to be checked.

Returns
True if an object is `ObjectRef`, *False* otherwise.

Return type
bool

get_ip

unidist.api.**get_ip**()

Get node IP address.

Returns
Node IP address.

Return type
str

num_cpus

unidist.api.**num_cpus**()

Get the number of CPUs used by the execution backend.

Return type
int

cluster_resources

unidist.api.**cluster_resources**()

Get resources of the cluster.

Returns
Dictionary with cluster nodes info in the form *{"node_ip0": {"CPU": x0}, "node_ip1": {"CPU": x1}, . . . }*.

Return type
dict

When calling an operation of the *API* provided by the framework unidist appeals to the `BackendProxy` object that dispatches the call to the concrete backend class instance (`MPIBackend`, `DaskBackend`, `RayBackend`, `PyMpBackend` or `PySeqBackend`). These classes are childs of the `Backend` interface and should override operations declared in it. Then, the concrete backend performs passed operation and hands over the result back to the `BackendProxy` that postprocesses it if necessary and returns it to the user.

Class View

unidist performs operations using the following key base classes:

- `BackendProxy`
- `RemoteFunction`
- `ActorClass`
- `Actor`
- `ActorMethod`
- `ObjectRef`

Module View

unidist modules layout is shown below. To deep dive into unidist internal implementation details just pick module you are interested in.

```
└──unidist
    ├── api
    ├── config
    └──core
        ├──backends
        |   ├──common
        |   |   └── data_id
        |   ├──dask
        |   |   ├── actor
        |   |   ├── backend
        |   |   └── remote_function
        |   ├──mpi
        |   |   ├──core
        |   |   |   ├── async_operations
        |   |   |   ├── common
        |   |   |   ├── communication
        |   |   |   ├── controller
        |   |   |   ├── monitor
        |   |   |   ├── local_object_store
        |   |   |   ├── shared_object_store
        |   |   |   ├── serialization
        |   |   |   └── worker
        |   |   ├── actor
        |   |   ├── backend
        |   |   └── remote_function
        |   ├──pymp
        |   |   ├──core
        |   |   |   ├── actor
        |   |   |   ├── api
        |   |   |   ├── object_store
        |   |   |   └── process_manager
        |   |   ├── actor
        |   |   ├── backend
        |   |   └── remote_function
        |   ├──pyseq
        |   |   ├──core
        |   |   |   ├── api
        |   |   |   └── object_store
        |   |   ├── actor
        |   |   ├── backend
        |   |   └── remote_function
        |   └──ray
        |       ├── actor
        |       ├── backend
        |       └── remote_function
        └──base
            ├── actor
            ├── backend
            ├── common
            |
            ├── object_ref
            └── remote_function
```

1.2.6 Contributing

Certificate of Origin

To keep a clear track of who did what, we use a *sign-off* procedure (same requirements for using the signed-off-by process as the Linux kernel has https://www.kernel.org/doc/html/v4.17/process/submitting-patches.html) on patches or pull requests that are being sent. The sign-off is a simple line at the end of the explanation for the patch, which certifies that you wrote it or otherwise have the right to pass it on as an open-source patch. The rules are pretty simple: if you can certify the below:

CERTIFICATE OF ORIGIN V 1.1

"By making a contribution to this project, I certify that:

1.) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or 2.) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or 3.) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it. 4.) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved."

```
This is my commit message

Signed-off-by: Awesome Developer <developer@example.org>
```

Code without a proper signoff cannot be merged into the master branch. Note: You must use your real name (sorry, no pseudonyms or anonymous contributions.)

The text can either be manually added to your commit body, or you can add either `-s` or `--signoff` to your usual `git commit` commands:

```
git commit --signoff
git commit -s
```

This will use your default git configuration which is found in .git/config. To change this, you can use the following commands:

```
git config --global user.name "Awesome Developer"
git config --global user.email "awesome.developer.@example.org"
```

If you have authored a commit that is missing the signed-off-by line, you can amend your commits and push them to GitHub.

```
git commit --amend --signoff
```

If you've pushed your changes to GitHub already you'll need to force push your branch after this with `git push -f`.

Commit Message formatting

We request that your first commit follow a particular format, and we **require** that your PR title follow the format. The format is:

```
FEAT-#9999: Add some functionality to enable something
```

The `FEAT` component represents the type of commit. This component of the commit message can be one of the following:

- FEAT: A new feature that is added
- DOCS: Documentation improvements or updates
- FIX: A bugfix contribution
- REFACTOR: Moving or removing code without change in functionality
- TEST: Test updates or improvements
- PERF: Performance enhancements

The #9999 component of the commit message should be the issue number in the unidist GitHub issue tracker: https://github.com/modin-project/unidist/issues. This is important because it links commits to their issues.

The commit message should follow a colon (:) and be descriptive and succinct.

A unidist CI job on GitHub will enforce that your pull request title follows the format we suggest. Note that if you update the PR title, you have to push another commit (even if it's empty) or amend your last commit for the job to pick up the new PR title. Re-running the job in Github Actions won't work.

General Rules for Committers

- Try to write a PR name as descriptive as possible.
- Try to keep PRs as small as possible. One PR should be making one semantically atomic change.
- Don't merge your own PRs even if you are technically able to do it.

Development Dependencies

We recommend doing development in a virtualenv or conda environment, though this decision is ultimately yours. You will want to run the following in order to install all of the required dependencies for running the tests and formatting the code:

```
conda env create --file environment_linux.yml # for Linux
conda env create --file environment_win.yml # for Windows
# or
pip install -r requirements.txt
```

Code Formatting and Lint

We use black for code formatting. Before you submit a pull request, please make sure that you run the following from the project root:

```
black .
```

We also use flake8 to check linting errors. Running the following from the project root will ensure that it passes the lint checks on Github Actions:

```
flake8 .
```

We test that this has been run on our Github Actions test suite. If you do this and find that the tests are still failing, try updating your version of black and flake8.

Adding a test

If you find yourself fixing a bug or adding a new feature, don't forget to add a test to the test suite to verify its correctness! We ask that you follow the existing structure of the tests for ease of maintenance.

Running the tests

To run the entire test suite, run the following from the project root:

```
python -m pytest unidist/test
```

If you've only modified a small amount of code, it may be sufficient to run a single test or some subset of the test suite. In order to run a specific test run:

```
python -m pytest unidist/test/test_new_functionality.py::test_new_functionality
```

The entire test suite is automatically run for each pull request.

Building documentation

To build the documentation, please follow the steps below from the project root (it is supposed you have dependencies from `environment_linux.yml` or `environment_win.yml` or `requirements.txt` installed):

```
# Build unidist to make C++ extensions available and also
# for correct module imports when building the documentation.
pip install -e .
cd docs
sphinx-build -b html . build
```

To visualize the documentation locally, run the following from *build* folder:

```
python -m http.server <port>
# python -m http.server 1234
```

then open the browser at *0.0.0.0:<port>* (e.g. *0.0.0.0:1234*).

1.2.7 Troubleshooting

We hope your experience with Unidist is bug-free, but there are some quirks about Unidist that may require troubleshooting. If you are still having issues, please open a Github issue.

Frequently encountered issues

This is a list of the most frequently encountered issues when using Unidist. Some of these are working as intended, while others are known bugs that are being actively worked on.

Error when using Open MPI while running in a cluster: `bash: line 1: orted: command not found`

Sometimes, when you run a program with Open MPI in a cluster, you may see the following error:

```
bash: line 1: orted: command not found
--------------------------------------------------------------------------
ORTE was unable to reliably start one or more daemons.
This usually is caused by:

* not finding the required libraries and/or binaries on
  one or more nodes. Please check your PATH and LD_LIBRARY_PATH
  settings, or configure OMPI with --enable-orterun-prefix-by-default

* lack of authority to execute on one or more specified nodes.
  Please verify your allocation and authorities.

* the inability to write startup files into /tmp (--tmpdir/orte_tmpdir_base).
  Please check with your sys admin to determine the correct location to use.

*  compilation of the orted with dynamic libraries when static are required
  (e.g., on Cray). Please check your configure cmd line and consider using
  one of the contrib/platform definitions for your system type.

* an inability to create a connection back to mpirun due to a
  lack of common network interfaces and/or no route found between
  them. Please check network connectivity (including firewalls
  and network routing requirements).
--------------------------------------------------------------------------
```

Solution

You should add the `--prefix` parameter to the `mpiexec` command with the path to the installed Open MPI library. If you are using a conda environment, then the required path will be: `$CONDA_PATH/envs/<ENV_NAME>`.

```
mpiexec -n 1 --prefix $CONDA_PATH/envs/<ENV_NAME> python script.py
```

Error when using Open MPI while running in a cluster: `OpenSSL version mismatch. Built against 30000020, you have 30100010`

Sometimes, when you run a program with Open MPI in a cluster, you may see the following error:

```
OpenSSL version mismatch. Built against 30000020, you have 30100010
--------------------------------------------------------------------------
ORTE was unable to reliably start one or more daemons.
This usually is caused by:

* not finding the required libraries and/or binaries on
  one or more nodes. Please check your PATH and LD_LIBRARY_PATH
  settings, or configure OMPI with --enable-orterun-prefix-by-default

* lack of authority to execute on one or more specified nodes.
  Please verify your allocation and authorities.

* the inability to write startup files into /tmp (--tmpdir/orte_tmpdir_base).
  Please check with your sys admin to determine the correct location to use.

*  compilation of the orted with dynamic libraries when static are required
  (e.g., on Cray). Please check your configure cmd line and consider using
  one of the contrib/platform definitions for your system type.

* an inability to create a connection back to mpirun due to a
  lack of common network interfaces and/or no route found between
  them. Please check network connectivity (including firewalls
  and network routing requirements).
--------------------------------------------------------------------------
```

This may happen due to the fact that OpenMPI uses OpenSSH but its version is built on a different version of OpenSSL than yours.

Solution

You should check for version compatibility of OpenSSH and OpenSSL and update them if necessary.

```
$ openssl version
OpenSSL 3.0.9 30 May 2023 (Library: OpenSSL 3.0.9 30 May 2023)
$ ssh -V
OpenSSH_8.9p1 Ubuntu-3ubuntu0.1, OpenSSL 3.0.2 15 Mar 2022
```

If you use `conda`, just add `openssh` library to your environment.

```
conda install -c conda-forge openssh
```

Error when using MPI backend: `mpi4py.MPI.Exception: MPI_ERR_SPAWN: could not spawn processes`

This error usually happens on Open MPI when you try to start the number of workers exceeding the number of physical cores. Open MPI binds workers to physical cores by default.

```
mpi4py.MPI.Exception: MPI_ERR_SPAWN: could not spawn processes
--------------------------------------------------------------------------
Primary job  terminated normally, but 1 process returned
a non-zero exit code. Per user-direction, the job has been aborted.
--------------------------------------------------------------------------
--------------------------------------------------------------------------
mpiexec detected that one or more processes exited with non-zero status, thus causing
the job to be terminated. The first process to do so was:

  Process name: [[35427,1],0]
  Exit code:    1
--------------------------------------------------------------------------
```

Solution

You should add one of the flags below to `mpiexec` command when running your application.

- `--bind-to hwthread`
- `--use-hwthread-cpus`
- `--oversubscribe`

```
mpiexec -n 1 --bind-to hwthread python script.py
```

To get more information about the flags refer to Open MPI's mpiexec command documentation.

Error when using MPI backend: `There are not enough slots available in the system to satisfy the <N> slots`

This error usually happens on Open MPI when you try to start the number of workers exceeding the number of physical cores. Open MPI binds workers to physical cores by default.

```
--------------------------------------------------------------------------
There are not enough slots available in the system to satisfy the <N>
slots that were requested by the application:

  python

Either request fewer slots for your application, or make more slots
available for use.

A "slot" is the Open MPI term for an allocatable unit where we can
launch a process.  The number of slots available are defined by the
environment in which Open MPI processes are run:

  1. Hostfile, via "slots=N" clauses (N defaults to number of
    processor cores if not provided)
```

(continues on next page)

(continued from previous page)

```
  2. The --host command line parameter, via a ":N" suffix on the
    hostname (N defaults to 1 if not provided)
  3. Resource manager (e.g., SLURM, PBS/Torque, LSF, etc.)
  4. If none of a hostfile, the --host command line parameter, or an
    RM is present, Open MPI defaults to the number of processor cores

In all the above cases, if you want Open MPI to default to the number
of hardware threads instead of the number of processor cores, use the
--use-hwthread-cpus option.

Alternatively, you can use the --oversubscribe option to ignore the
number of available slots when deciding the number of processes to
launch.
--------------------------------------------------------------------------
```

Solution

You should add one of the flags below to `mpiexec` command when running your application to allow Open MPI to start the number of workers exceeding the number of physical cores.

- `--bind-to hwthread`
- `--use-hwthread-cpus`
- `--oversubscribe`

```
mpiexec -n 1 --bind-to hwthread python script.py
```

To get more information about the flags refer to Open MPI's mpiexec command documentation.

Shared object store for MPI backend is not supported in C/W model for MPICH version less than 4.2.0

MPICH versions less than 4.2.0 have an issue related to shared memory feature in Controller/Worker model.

Solution

You can run your script using MPICH in SPMD model, or use other MPI implementations such as Open MPI, Intel MPI, or MPICH above 4.2.0.

To get started with unidist refer to the *getting started* page.

To deep dive into unidist internals refer to *the framework architecture*.

INDEX