cronus

Release 1.0.1

Mar 04, 2021

Contents:

cronus is a Python tool designed to facilitate *Markov Chain Monte Carlo (MCMC)* and *Nested Sampling (NS)* in large supercomputing clusters.

It relies on the powerful MCMC sampler **zeus** to do the heavy-lifting and incorporates various MPI features along with a suite of automated *Convergence Diagnostics*.

`cronus` is designed to be used via the terminal using parameter files and it is particularly suited for Astrophysical and Cosmological applications.

CHAPTER 1

Requirements

`cronus` is compatible with Python 3.6+. It requires `numpy`, `scipy`, `mpi4py`, `iminuit`, `h5py` and `zeus` to run. If you want to use `cronus` with either `emcee` or `dynesty` please make sure that you have those installed too.

You can find information about how to install `mpi4py` and its prerequisites at https://mpi4py.readthedocs.io/en/stable/install.html

CHAPTER 2

Install using pip

We recommend to use pip to install the latest stable version of `cronus`:

```
pip install cronus-mcmc
```

CHAPTER 3

Install from source

Alternatively, install the latest version of `cronus` from source:

```
git clone https://github.com/minaskar/cronus.git
cd cronus
pip install -r requirements.txt
pip install .
```

CHAPTER 4

Making sure that cronus is installed properly

If everything went well, you should be able to import `cronus` in Python from anywhere in your directory structure:

```
$ python -c "import cronus"
```

If you get an error message, something went wrong. Check twice the instructions above, try again, or contact us.

`cronus` also installs some shell scripts. If everything went well, if you try to run in the shell `cronus-run`, you should get a message asking you for an input file, instead of a command not found error.

Note: If you do get a command not found error, this means that the folder where your local scripts are installed has not been added to your path.

To solve this on unix-based machines, look for the `cronus-run` script from your home and scratch folders with:

```
$ find `pwd` -iname cronus-run -printf %h\\n
```

in Linux, or:

```
$ which -a cronus-run
```

in Mac OS X.

This should print the location of the script, e.g. `/home/you/.local/bin`. Add:

```
$ export PATH="/home/you/.local/bin":$PATH
```

at the end of your `~/.bashrc` file, and restart the terminal or do `source ~/.bashrc`. Alternatively, simply add that line to your cluster jobscripts just before calling `cronus-run`.

CHAPTER 5

Quick Start

5.1 Overview

The main purpose of `cronus` is to facilitate large-scale Bayesian Inference (e.g. MCMC or NS) in modern supercomputing environments. `cronus` utilises `MPI` to efficiently distribute the tasks to multiple nodes. Another important feature of `cronus` is its integrated and automated suite of *Convergence Diagnostics*.

Before we go into detail about how to use `cronus` let us first discuss the way it works in a higher level. `cronus` accepts as an input a parameter file that specifies the following:

- The Python file that contains the definition of the Log Likelihood function,
- A set of priors and/or fixed values for the different parameters of the model that enters the Log Likelihood function.

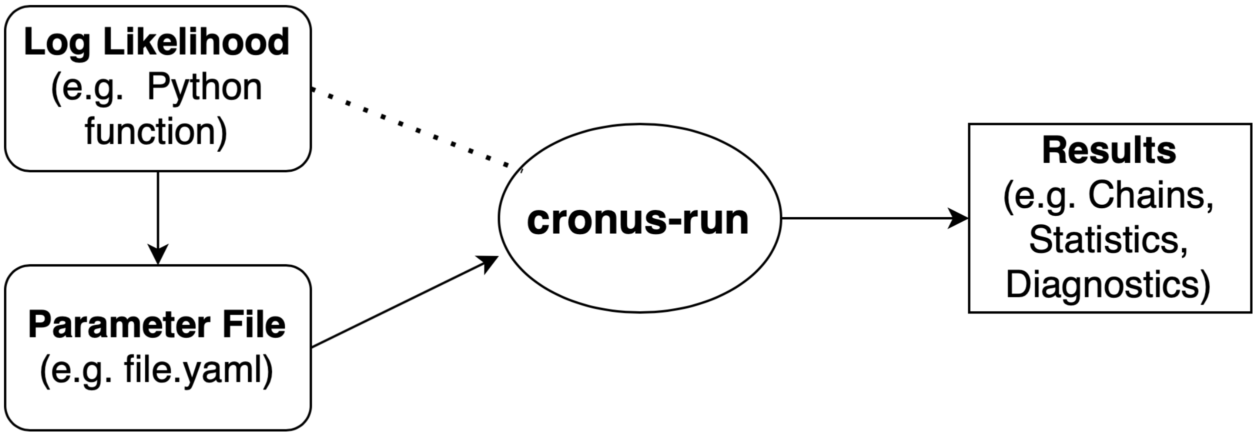
Note: The Paremeter file can also be used to specify some additional optional information, like:

- A set of parameters that configure the MCMC/NS sampler (e.g. number of walkers), those are usually trivial to define.
- A few threshold values for the *Convergence Diagnostics*,
- The path/directory for the results to be saved in.

For more information about this please read the *Advanced Use* page.

Once a parameter file is provided, `cronus` efficiently distributes the sampling tasks to all available CPUs and runs until Convergence is reached. The results are saved iteratively so that the researcher can monitor the progress.

Let us present here a simple example that will help illustrate the basic features and capabilities of `cronus`.



5.2 Log Likelihood Function

The first thing we need to do is to create a Python file in which we define the Log Likelihood function. There is no real restricton to this. The model itself can be computed in any programming language (e.g. C, C++, Fortran) and the Log Likelihood can be a Python wrapper for this. In this example we will define a strongly-correlated `5-dimensional Normal distribution`.

```
import os
os.environ["OMP_NUM_THREADS"] = "1"

import numpy as np

ndim = 5

C = np.identity(ndim)
C[C==0] = 0.95
Cinv = np.linalg.inv(C)

def log_likelihood(x):
    return - 0.5 * np.dot(x, np.dot(Cinv, x))
```

We then save the file as `logprob.py`.

Note: The important thing to note here is that the function accepts a single argument `x`. If your Log Likelihood requires more than one argument (e.g. data, covariance, etc.) we recommend to make those global like we did with the `ivar` array in the aforementioned example.

Note: Some builds of NumPy (including the version included with Anaconda) will automatically parallelize some operations using something like the MKL linear algebra. This can cause problems when used with the parallelization methods described here so it can be good to turn that off (by setting the environment variable `OMP_NUM_THREADS=1`, for example).

```
import os
os.environ["OMP_NUM_THREADS"] = "1"
```

5.3 Parameter File

The next step is to create the parameter file that we will call `file.yaml`:

```yaml
Likelihood:
  path: logprob.py
  function: log_likelihood

Parameters:
  a:
    prior:
      type: uniform
      min: -10.0
      max: 10.0
  b:
    fixed: 1.0
  c:
    prior:
      type: normal
      loc: 1.0
      scale: 1.0
  d:
    prior:
      type: normal
      loc: 0.0
      scale: 2.5
  e:
    prior:
      type: normal
      loc: -0.5
      scale: 1.0
```

You can see the following *sections* in the parameter file:

- The `Likelihood` section which includes information about the path of the Log Likelihood function (i.e. both the directory/filename and the name of the function).
- The `Parameters` section which includes the priors of fixed values for each parameter of the model.

For more information about these and additional options in the parameter file please see the *Advanced Use* page.

5.4 Run cronus

To run this example go the directory where you saved `file.yaml` and do:

```
$ mpiexec -n 8 cronus-run file.yaml
```

Here we used 8 CPUs.

5.5 Results

After a few seconds, an output directory will be created containing the following files:

```
chains/run1
          ├── chain_0.h5
          ├── chain_1.h5
          ├── IAT_0.dat
          ├── IAT_1.dat
          ├── GelmanRubin.dat
          ├── MAP.npy
          ├── hessian.npy
          ├── para.yaml
          ├── results.dat
          └── varnames.dat
```

All but the `results.dat` file will be created shortly. The files will iteratively be updated every few iterations. Once the sampling is done, the `results.dat` file will be added to the list.

Let's have a look at what each of those files contains:

- The `chain_x.h5` files contain the actual MCMC samples.
- The `IAT_x.dat` files contain the estimated *Integrated Autocorrelation Time* (IAT) for each and parameter. This is a measure of how independent the chain samples are (i.e. the lower the IAT the better).
- The `GelmanRubin.dat` file contains the *Gelman-Rubin* `R_hat` diagnostic for each parameter.
- The `MAP.npy` file contains the *Maximum a Posteriori* (MAP) estimate.
- The `hessian.npy` file contains the *Hessian matrix* evaluated at the MAP.
- The `para.yaml` file is a copy of the original parameter file with some extra information explicitly described.
- The `results.dat` file includes a summary of the results (e.g. mean, std, 1-sigma, 2-sigma, etc.).
- The `varnames.dat` file contains a list of the parameter names.

Note: If we can open the `results.dat` file using a text editor we will see the following:

```
| Name     |       MAP |      mean |    median |       std |   -1 sigma |    +1␣
↪sigma |   -2 sigma |   +2 sigma |       IAT |       ESS |    R_hat |
|--------+----------+----------+----------+----------+------------+----------
↪--+------------+------------+---------+---------+---------|
| a      | 0.885898 | 0.881579 | 0.879316 | 0.304584 |  -0.301652 |    0.
↪308398 |  -0.609184 |   0.609584 | 6.82365 | 4044.76 |  1        |
| c      | 0.891147 | 0.879663 | 0.881513 | 0.298963 |  -0.301561 |    0.
↪293607 |  -0.603484 |   0.59629  | 6.87625 | 4013.82 |  1.0003 |
| d      | 0.878582 | 0.880138 | 0.881647 | 0.307091 |  -0.311894 |    0.
↪302304 |  -0.617898 |   0.611955 | 6.814   | 4050.48 |  1.0006 |
| e      | 0.818762 | 0.807181 | 0.807153 | 0.297321 |  -0.29532  |    0.
↪294845 |  -0.593549 |   0.597654 | 6.5086  | 4240.54 |  1.0002 |
```

Now let's see how we can easily access this information using `cronus`.

The first thing we want to do is read the chains using the `read_chains` module of `cronus`:

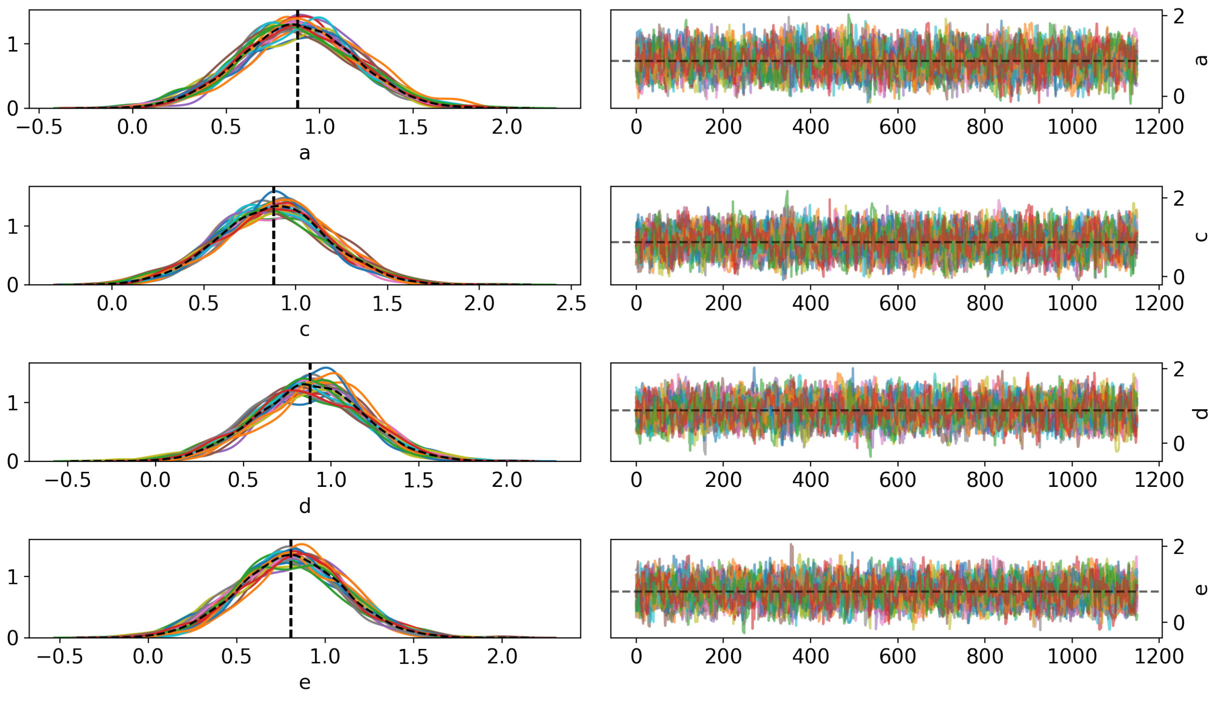
```python
import cronus

results = cronus.read_chains('chains/run1')

print(results.Summary)
```

This will print the contents of the `results.dat` file.

We can easily create some plots by running:

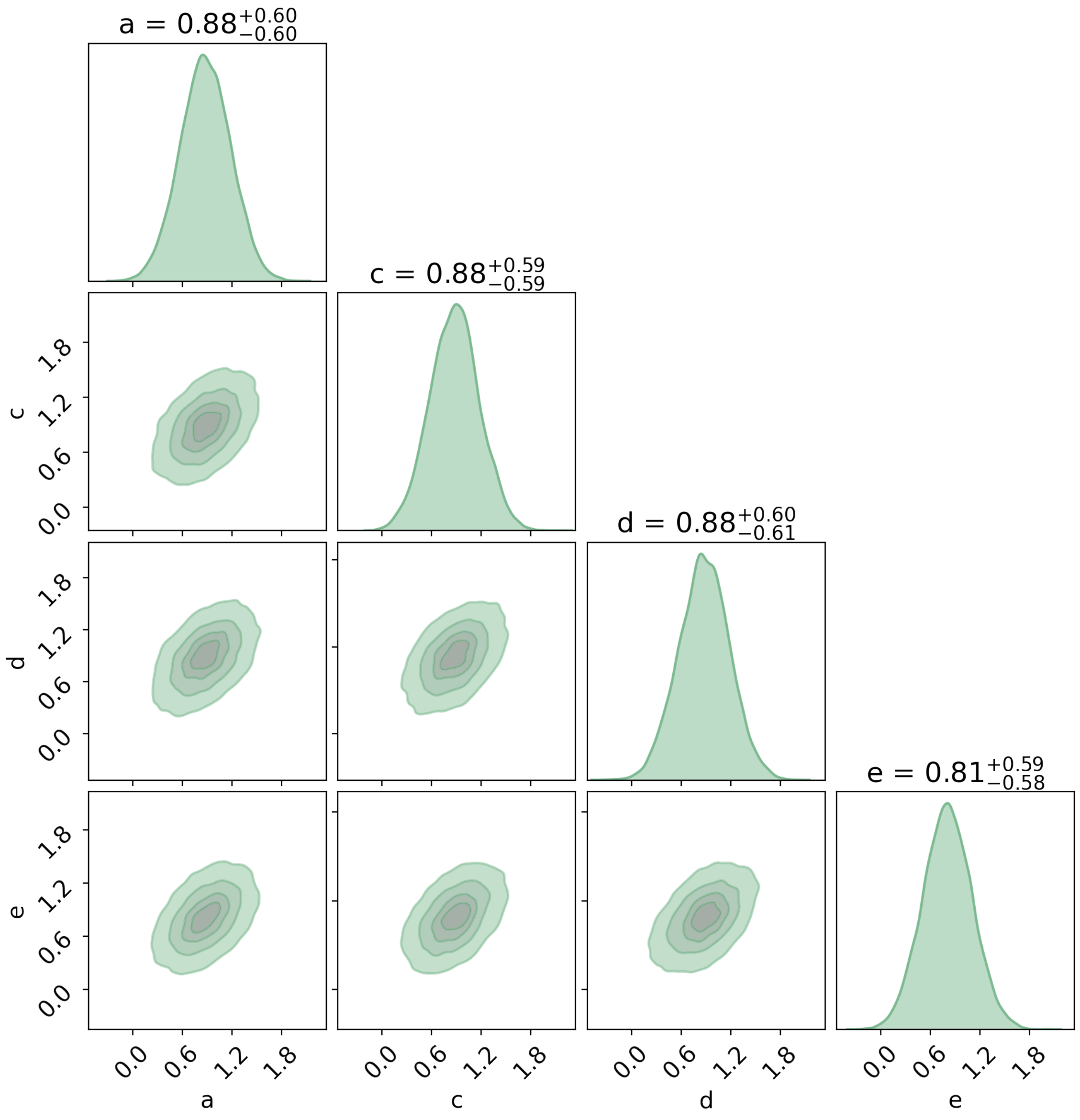
```
cronus.traceplot(results)
```

to get the following `traceplot`:



Or, run the following to get a `cornerplot`:

```
fig, axes = cronus.cornerplot(results.trace, labels=results.varnames)
```



$a = 0.88^{+0.60}_{-0.60}$
$c = 0.88^{+0.59}_{-0.59}$
$d = 0.88^{+0.60}_{-0.61}$
$e = 0.81^{+0.59}_{-0.58}$
c
d
e
a
c
d
e
0.0
0.6
1.2
1.8

CHAPTER 6

Advanced Use

6.1 Log Likelihood Function

The Log Likelihood function needs to be defined in a separate `.py` file. It should be a function of **one argument**, either a numpy array or a dictionary.

If you need to pass more information (e.g. data, covariance matrix, precision matrix, etc.) to the Log Likelihood function you should declare those as global variables. This is the easiest and most consistent way to make MPI not complain; it's also the most computationally efficient method (i.e. passing the whole dataset to all processes eveytime you call the function can be slow).

Here we show a short toy example where we demonstrate how we should define such a function.

```
import numpy as np

ndim = 10

data = np.random.randn(ndim) # Random data vector
C = np.identity(ndim) # Identity Covariance Matrix
Cinv = np.linalg.inv(C) # Inverse Covariance Matrix

def log_likelihood(x): # Normal distribution
    return -0.5*np.dot(x, np.dot(Cinv, x))
```

6.2 Parameter File

The parameter file can generally include more information than the options presented in the *Quick Start* page.

6.2.1 Likelihood

Usually the argument of the Log Likelihood function is a *1D numpy array* but we can also use a dictionary instead. To do so we need to add the `dictionary: True` option to the Likelihood block, for instance:

```
Likelihood:
  path: path/to/logprob.py
  function: log_likelihood
  dictionary: True
```

6.2.2 Parameters

Every parameter needs to be either fixed or free:

- For fixed parameters we need to specify their value in Parameter block (i.e. parameter `a` in the following example).
- For free parameters we need to specify a prior instead. So far, only `uniform` and `normal` priors are supported. For a `uniform` prior we need to specify the uniform interval `(min, max)` (i.e. parameter `b` in the following example). For a `normal` prior we need to specify the mean `loc` and standard deviation `scale` (i.e. parameter `c` in the following example).

```
Parameters:
  a:
    fixed: 1.0
  b:
    prior:
      type: uniform
      min: -1.0
      max: 1.0
  c:
    prior:
      type: normal
      loc: 0.0
      scale: 1.0
```

6.2.3 Sampler

`cronus` supports three different samplers, `zeus` (Default), `emcee`, and `dynesty`. The prefered sampler can be specified using the `name` option in the `Sampler` section of the parameter file, for instance:

```
Sampler:
  name: zeus
  ...
```

When either `zeus` or `emcee` is used as the prefered sampler then the following options are available:

- `ndim` is the total number of parameters/dimensions.
- `nwalkers` is the total number of walkers (i.e. internal parallel chains for zeus or emcee). This number needs to be at least twice the number of free parameters.
- `nchains` is the number of parallel chains, we recommend at least two and preferably 4 to get good estimate of the *Gelman-Rubin* diagnostic.

- `ncheck` specifies the number of steps after which the samples are saved and the *Convergence Criteria* are assessed. The default value is 100 which means that the samples are saved and convergence is diagnosed every 100 steps.
- `maxiter` specifies the maximum number of iterations (Default is inf).
- `miniter` specifies the minimum number of iterations (Default is 0).
- `maxcall` specifies the maximum number of Log Likelihood evalluations/calls (Default is inf).
- `initial` controls the initialization of the walker positions. The available options are: `ellipse` (this is a small ellipse around the *Maximum a posteriori* estimate, this is the default and recommended choice), `laplace` (sample the initial positions of the walkers from the *Laplace approximation* of the posterior distribution), and `prior` (sample the initial positions from the prior distribution, not the best choice).
- `thin` is the thinning rate for the chains (i.e. if `thin=5` then save every 5th element to the chain). This can significantly reduce the size of the output files if the autocorrelation time of the chain is large. The default value is 1.

When `dynesty` is used as the prefered sampler then the following options are available:

- `ndim` is the total number of parameters/dimensions.
- `bound`
- `dlogz`
- `maxiter` specifies the maximum number of iterations (Default is inf).
- `maxcall` specifies the maximum number of Log Likelihood evalluations/calls (Default is inf).
- `pfrac`

6.2.4 Diagnostics

So far `cronus` includes two distinct convergence diagnostics, the Gelman-Rubin statistic and the Autocorrelation Time test. Their combination seems to work well in Astrophysical and Cosmological likelihoods.

Lets see how one can customize the thresholds of those criteria:

- Either of them can be turned off or on (Default) using the `use` argument.
- `|R_hat - 1| < epsilon` is the threshold for the *Potential Scale Reduction Factor* (PSRF). We recommend to use a value of `epsilon` that it is smaller than 0.05 (Default).
- In terms of the *Integrated Autocorrelation Time* (IAT) we provide two criteria, if the chain is longer than `nact = 20` (Default) times the estimated IAT and the IAT has changed less than `dact = 3%` (Default) the criteria are satisfied. If both *Gelman-Rubin* and IAT criteria are satisfied then sampling stops.

All of the diagnostic options can be seen here:

```
Diagnostics:
  Gelman-Rubin:
    use: True
    epsilon: 0.05
  Autocorrelation:
    use: True
    nact: 20
    dact: 0.03
```

6.2.5 Output

The only option of the `Output` block is a directory path in which the samples/results will be saved. If the provided directory doesn't exist one will be created by `cronus`. The default directory is the current one.

```
Output: path/to/output/folder/chains
```

6.3 Running cronus

To run `cronus`, given a parameter file `file.yaml`, we execute the following command:

```
$ mpiexec -n [nprocesses] cronus-run file.yaml
```

where, `nprocesses` is the number of available CPUs. Depending on the cluster you are using you may need to use `mpirun` or `srun` instead of `mpiexec`.

Note: For better performance we recommend to use a number of processes that can be divided by the number of chains `nchains`. Ideally, we recommend to use `nchains * (nwalkers/2 + 1)` if available, there's no real computational benefit in using more than this.

6.4 Results

6.4.1 zeus or emcee

When either `zeus` or `emcee` is used as the prefered sampler then the results are saved as `h5` files. There are as many `h5` files saved as the number of chains `nchains`. Each file contains two datasets, one called `samples` which constists of the samples as the name suggests, and one named `logprob` which includes the respective values of the Log Posterior Distribution.

After a few seconds of running the following files will be created in the provided `Output` directory:

```
chains
    ├── chain_0.h5
    ├── chain_1.h5
    ├── ...
    └── chain_[nchains].h5
```

The files will iteratively be updated every few iterations.

Note: You can access those results by doing:

```python
import numpy as np
import h5py

with h5py.File('chains/chain_0.h5', "r") as hf:
    samples = np.copy(hf['samples'])
    logprob = np.copy(hf['logprob'])
```

The shape of the samples array would be `(Iteration, nwalkers, ndim)` and the shape of the Log Posterior array will be `(Iteration, nwalkers)`. You can easily *flatten* this, combining all the walkers into one chain and discarding the first half of the chain, by running:

```
nsamples, nwalkers, ndim_prime = np.shape(samples)

samples_flat = samples[nsamples//2:].reshape(-1, ndim_prime)

logprob_flat = logprob[nsamples//2:].reshape(-1, 1)
```

6.4.2 dynesty

When `dynesty` is used as the sampler then the results are saved as a numpy `npy` format file.

CHAPTER 7

FAQ

7.1 What if the Log Likelihood requires more than one argument?

.

CHAPTER 8

Troubleshooting

8.1 infiniband

There seem to be some issues with some `mpi4py` features when used in a computing cluster with *infiniband*. This leads to `cronus` to hang in an `infiniband` multi-node setting.

8.1.1 OpenMPI

If you are using `OpenMPI` you can try including the following command which in your jobscript.

```
export OMPI_MCA_pml=ob1
```

This should disable the *infiniband* interface.

8.1.2 Intel MPI

The mpi4py package is using matching probes `(MPI_Mpobe)` for the receiving function `recv()` instead of regular `MPI_Recv` operations per default. These matching probes from the `MPI 3.0` standard however are not supported for all fabrics, which may lead to a hang in the receiving function.

Therefore, users are recommended to leverage the `OFI` fabric instead of `TMI` for `Omni-Path` systems. For the `Intel MPI Library`, the configuration could look like the following environment variable setting:

```
export I_MPI_FABRICS=ofi
```

CHAPTER 9

API

9.1 Results Module

9.2 Plotting Module

CHAPTER 10

Getting Started

- See the *Installation* page for instruction on how to easily install `cronus`.
- See the *Quick Start* page for a simple example.
- See the *Advanced Use* page for more information about the ways `cronus` can be configured.
- See the *FAQ* page for frequently asked questions.
- See the *Troubleshooting* page for any problems with `cronus`.
- See the *API* page for a detailed API documentation.

CHAPTER 11

Attribution

- Please cite X if you find this code useful in your research.

CHAPTER 12

Licence

CHAPTER 13

Changelog

1.0.0 (07/09/20)

- First public release.