arboreto Documentation

Release 0.1.5

Thomas Moerman

Jun 12, 2018

Contents

Inferring a gene regulatory network (GRN) from gene expression data is a computationally expensive task, exacerbated by increasing data sizes due to advances in high-throughput gene profiling technology.

Quick Start

- Installation
- User guide
- Report an issue
- Source code at Github
- Releases at PyPI

The *Arboreto* software library addresses this issue by providing a computational strategy that allows executing the class of GRN inference algorithms exemplified by GENIE3[1] on hardware ranging from a single computer to a multi-node compute cluster. This class of GRN inference algorithms is defined by a series of steps, one for each target gene in the dataset, where the most important candidates from a set of regulators are determined from a regression model to predict a target gene's expression profile.

Members of the above class of GRN inference algorithms are attractive from a computational point of view because they are parallelizable by nature. In arboreto, we specify the parallelizable computation as a Dask graph[2], a data structure that represents the task schedule of a computation. A Dask scheduler assigns the tasks in a Dask graph to the available computational resources. Arboreto uses the Dask distributed scheduler to spread out the computational tasks over multiple processes running on one or multiple machines.

Arboreto currently supports 2 GRN inference algorithms:

1. **GRNBoost2**: fast GRN inference algorithm using stochastic Gradient Boosting Machine[3] regression with early-stopping regularization, the Arboreto flagship algorithm.
2. **GENIE3**: the popular classic GRN inference algorithm using Random Forest (RF) or ExtraTrees (ET) regression.

```
# import python modules
import pandas as pd
from arboreto.utils import load_tf_names
from arboreto.algo import grnboost2
```

(continues on next page)

[1] Huynh-Thu VA, Irrthum A, Wehenkel L, Geurts P (2010) Inferring Regulatory Networks from Expression Data Using Tree-Based Methods. PLoS ONE

[2] Rocklin, M. (2015). Dask: parallel computation with blocked algorithms and task scheduling. In Proceedings of the 14th Python in Science Conference (pp. 130-136).

[3] Friedman, J. H. (2002). Stochastic gradient boosting. Computational Statistics & Data Analysis, 38(4), 367-378.

(continued from previous page)

```
if __name__ == '__main__':
    # load the data
    ex_matrix = pd.read_csv(<ex_path>, sep='\t')
    tf_names = load_tf_names(<tf_path>)

    # infer the gene regulatory network
    network = grnboost2(expression_data=ex_matrix,
                        tf_names=tf_names)

    network.head()
```

TF	target	importance
G109	G1406	151.648784
G16	G1440	136.741815
G188	G938	124.707570
G10	G1312	124.195566
G48	G1419	121.488200

Check out more examples.

CHAPTER 1

License

BSD 3-Clause License

CHAPTER 2

References

2.1 Installation Guide

Caution: Python Environment

It is highly recommended to prepare a Python environment with the Anaconda or Miniconda distribution and install Arboreto's dependencies using the conda package manager.

- NumPy
- SciPy
- scikit-learn
- pandas
- dask
- distributed

This avoids complexities in ensuring that libraries like NumPy and SciPy link against an optimized implementation of linear algebra routines.

2.1.1 Install using pip

The arboreto package is available from PyPI (Python Package Index), a repository of software for the Python programming language.

Using pip, installing the arboreto package is straightforward:

```
$ pip install arboreto
```

Check out the installation:

```
$ pip show arboreto

Name: arboreto
Version: 0.1.5
Summary: Scalable gene regulatory network inference using tree-based ensemble␣
↪regressors
Home-page: https://github.com/tmoerman/arboreto
Author: Thomas Moerman
Author-email: thomas.moerman@gmail.com
License: BSD 3-Clause License
Location: /vsc-hard-mounts/leuven-data/software/biomed/Anaconda/5-Python-3.6/lib/
↪python3.6/site-packages
Requires: scipy, scikit-learn, numpy, pandas, dask, distributed
```

Note: You can use pip to install arboreto in an Anaconda environment.

2.1.2 Install from source

Installing Arboreto from source is possible using following steps:

1. clone the Github repository using the git tool:

```
$ git clone https://github.com/tmoerman/arboreto.git
$ cd arboreto
```

2. build Arboreto using the provided script:

```
$ ./pypi_build.sh
```

3. install the freshly built Arboreto package using pip:

```
$ pip install dist/*
```

2.2 User Guide

2.2.1 Modules overview

Arboreto consists of multiple python modules:

arboreto.algo

- Intended for **typical users**.
- Access point for launching GRNBoost2 or GENIE3 on local or distributed hardware.

arboreto.core

- Intended for **advanced users**.
- Contains the low-level building blocks of the Arboreto framework.

arboreto.utils

- Contains small utility functions.

2.2.2 Dependencies Overview

Arboreto uses well-established libraries from the Python ecosystem. Arboreto avoids being a proverbial "batteries-included" library, as such an approach often entails unnecessary complexity and maintenance. Arboreto aims at doing only one thing, and doing it well.

Concretely, the user will be exposed to one or more of following dependencies:

- Pandas or NumPy: the user is expected to provide the input data in an expected format. Pandas and NumPy are well equipped with functions for data preprocessing.
- Dask.distributed: to run Arboreto on a cluster, the user is responsible for setting up a network of a scheduler and workers.
- scikit-learn: relevant for advanced users only. Arboreto can run "DIY" inference where the user provides their own parameters for the Random Forest or Gradient Boosting regressors.

2.2.3 Input / Output

INPUT

- **an expression matrix (rows = observations, columns = genes)**
  - either a Pandas DataFrame or a NumPy ndarray
- **a list of gene names corresponding to the columns of the expression matrix**
  - optional
- **a list of transcription factors (a.k.a. TFs)**
  - optional

OUTPUT

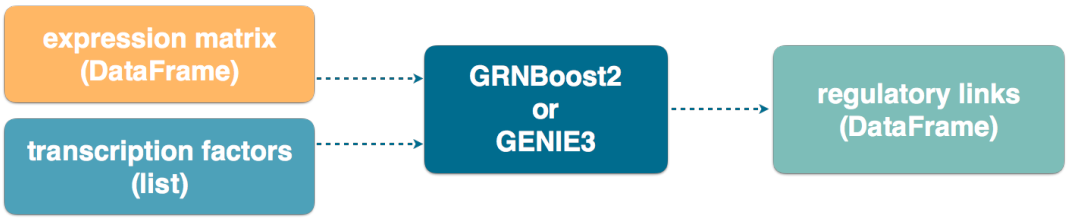
- **regulatory links**
  - a Pandas DataFrame `['TF', 'target', 'importance']`

Tip: As data for following code snippets, you can use the data for network 1 from the DREAM5 challenge (included in the resources folder of the Github repository):

- <ex_path> = net1_expression_data.tsv
- <tf_path> = net1_transcription_factors.tsv

Expression matrix as a Pandas `DataFrame`

The input can be specified in a number of ways. Arguably the most straightforward way is to specify the expression matrix as a Pandas DataFrame, which also contains the gene names as the column header.



In the following code snippet, we launch network inference with grnboost2 by specifying the `expression_data` as a DataFrame.

Listing 1: *Expression matrix as a Pandas DataFrame*

```
import pandas as pd
from arboreto.utils import load_tf_names
from arboreto.algo import grnboost2

if __name__ == '__main__':
    # ex_matrix is a DataFrame with gene names as column names
    ex_matrix = pd.read_csv(<ex_path>, sep='\t')

    # tf_names is read using a utility function included in Arboreto
    tf_names = load_tf_names(<tf_path>)

    network = grnboost2(expression_data=ex_matrix,
                        tf_names=tf_names)

    network.to_csv('output.tsv', sep='\t', index=False, header=False)
```

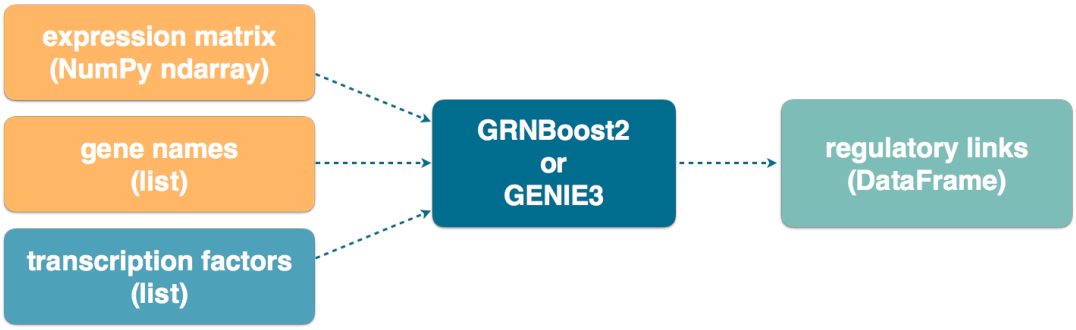
Note: Notice the emphasized line:

```
if __name__ == '__main__':
    # ... code ...
```

This is a Python idiom necessary in situations where the code spawns new Python processes, which Dask does under the hood of the `grnboost2` and `genie3` functions to parallelize the workload.

Expression matrix as a NumPy `ndarray`

Arboreto also supports specifying the expression matrix as a Numpy ndarray (in our case: a 2-dimensional matrix). In this case, the gene names must be specified explicitly.



Caution: You must specify the gene names in the same order as their corresponding columns of the NumPy matrix. **Getting this right is the user's responsibility.**

Listing 2: *Expression matrix as a NumPy ndarray*

```
import numpy as np
from arboreto.utils import load_tf_names
from arboreto.algo import grnboost2

if __name__ == '__main__':
    # ex_matrix is a numpy ndarray, which has no notion of column names
    ex_matrix = np.genfromtxt(<ex_path>, delimiter='\t', skip_header=1)

    # we read the gene names from the first line of the file
    with open(<ex_path>) as file:
        gene_names = [gene.strip() for gene in file.readline().split('\t')]

    # sanity check to verify the ndarray's nr of columns equals the length of the␣
↪gene_names list
    assert ex_matrix.shape[1] == len(gene_names)

    # tf_names is read using a utility function included in Arboreto
    tf_names = load_tf_names(<tf_path>)

    network = grnboost2(expression_data=ex_matrix,
                        gene_names=gene_names,  # specify the gene_names
                        tf_names=tf_names)

    network.to_csv('output.tsv', sep='\t', index=False, header=False)
```

2.2.4 Running with a custom Dask Client

Arboreto uses Dask.distributed to parallelize its workloads. When the user doesn't specify a dask distributed Client explicitly, Arboreto will create a LocalCluster and a Client pointing to it.

Alternatively, you can create and configure your own Client instance and pass it on to Arboreto. Situations where this is useful include:

- inferring multiple networks from different datasets
- inferring multiple networks using different parameters from the same dataset
- the user requires custom configuration for the LocalCluster (memory limit, nr of processes, etc.)

Following snippet illustrates running the gene regulatory network inference multiple times, with different initialization seed values. We create one Client and pass it to the different inference steps.

Listing 3: *Running with a custom Dask Client*

```
import pandas as pd
from arboreto.utils import load_tf_names
from arboreto.algo import grnboost2
from distributed import LocalCluster, Client

if __name__ == '__main__':
    # create custom LocalCluster and Client instances
    local_cluster = LocalCluster(n_workers=10,
                                 threads_per_worker=1,
                                 memory_limit=8e9)
    custom_client = Client(local_cluster)

    # load the data
    ex_matrix = pd.read_csv(<ex_path>, sep='\t')
    tf_names = load_tf_names(<tf_path>)

    # run GRN inference multiple times
    network_666 = grnboost2(expression_data=ex_matrix,
                            tf_names=tf_names,
                            client_or_address=custom_client,  # specify the custom␣
↪client
                            seed=666)

    network_777 = grnboost2(expression_data=ex_matrix,
                            tf_names=tf_names,
                            client_or_address=custom_client,  # specify the custom␣
↪client
                            seed=777)

    # close the Client and LocalCluster after use
    client.close()
    local_cluster.close()

    network_666.to_csv('output_666.tsv', sep='\t', index=False, header=False)
    network_777.to_csv('output_777.tsv', sep='\t', index=False, header=False)
```

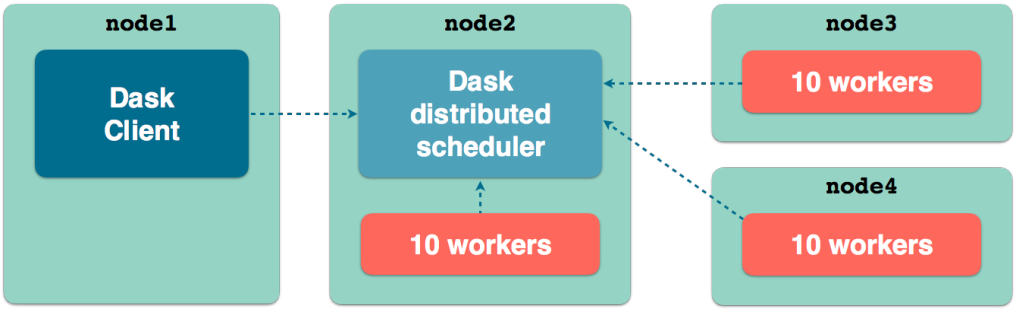
2.2.5 Running with a Dask distributed scheduler

Arboreto was designed to run gene regulatory network inference in a distributed setting. In distributed mode, some effort by the user or a systems administrator is required to set up a dask.distributed `scheduler` and some `workers`.

Tip: Please refer to the Dask distributed network setup documentation for instructions on how to set up a Dask

distributed cluster.

Following diagram illustrates a possible topology of a Dask distributed cluster.



- `node_1` runs a Python script, console or a Jupyter notebook server, a Client instance is configured with the TCP address of the distributed scheduler, running on `node_2`
- `node_2` runs a distributed scheduler and 10 workers pointing to the scheduler
- `node_3` runs 10 distributed workers pointing to the scheduler
- `node_4` runs 10 distributed workers pointing to the scheduler

With a small modification to the code, we can infer a regulatory network using all workers connected to the distributed scheduler. We specify a Client that is connected to the Dask distributed scheduler and pass it as an argument to the inference function.

Listing 4: *Running with a Dask distributed scheduler*

```
import pandas as pd
from arboreto.utils import load_tf_names
from arboreto.algo import grnboost2
from distributed import Client

if __name__ == '__main__':
    ex_matrix = pd.read_csv(<ex_path>, sep='\t')
    tf_names = load_tf_names(<tf_path>)

    scheduler_address = 'tcp://10.118.224.134:8786'  # example address of the remote␣
↪scheduler
    cluster_client = Client(scheduler_address)       # create a custom Client

    network = grnboost2(expression_data=ex_matrix,
                        tf_names=tf_names,
                        client_or_address=cluster_client)  # specify Client connected␣
↪to the remote scheduler

    network.to_csv('output.tsv', sep='\t', index=False, header=False)
```

2.3 Examples

2.3.1 Python script

- Example python script running GRNBoost2 on files located in the same folder.

Listing 5: *<arboreto repo>/resources/dream5/net1/run_grnboost2.py*

```
import pandas as pd

from distributed import Client, LocalCluster
from arboreto.utils import load_tf_names
from arboreto.algo import grnboost2

if __name__ == '__main__':

    in_file  = 'net1_expression_data.tsv'
    tf_file  = 'net1_transcription_factors.tsv'
    out_file = 'net1_grn_output.tsv'

    # ex_matrix is a DataFrame with gene names as column names
    ex_matrix = pd.read_csv(in_file, sep='\t')

    # tf_names is read using a utility function included in Arboreto
    tf_names = load_tf_names(tf_file)

    # instantiate a custom Dask distributed Client
    client = Client(LocalCluster())

    # compute the GRN
    network = grnboost2(expression_data=ex_matrix,
                        tf_names=tf_names,
                        client_or_address=client)

    # write the GRN to file
    network.to_csv(out_file, sep='\t', index=False, header=False)
```

Run as a classic python script:

```
cd <arboreto repo>/resources/dream5/net1
python run_grnboost2
```

2.3.2 Jupyter notebooks

Following are links to example Jupyter notebooks that illustrate different Arboreto usage scenarios (links render notebooks in Jupyter nbviewer).

- Example 01 - GRNBoost2 local

  A basic usage scenario where we infer the gene regulatory network from a single dataset on the local machine.

- Example 02 - GRNBoost2 with custom Dask Client

  A slightly more advanced scenario where we infer the gene regulatory network from a single dataset, using a custom Dask client.

- Example 03 - GRNBoost2 with transposed input file

  Illustrates how to easily prepare the input data using a Pandas DataFrame, in case the input file happens to be transposed with respect to the Arboreto input conventions.

2.4 GRN Inference Algorithms

Arboreto hosts multiple (currently 2, contributions welcome!) algorithms for inference of gene regulatory networks from high-throughput gene expression data, for example *single-cell RNA-seq* data.

2.4.1 GRNBoost2

GRNBoost2 is the flagship algorithm for gene regulatory network inference, hosted in the Arboreto framework. It was conceived as a fast alternative for GENIE3, in order to alleviate the processing time required for larger datasets (tens of thousands of observations).

GRNBoost2 adopts the GRN inference strategy exemplified by GENIE3, where for each gene in the dataset, the most important feature are a selected from a trained regression model and emitted as candidate regulators for the target gene. All putative regulatory links are compiled into one dataset, representing the inferred regulatory network.

In GENIE3, Random Forest regression models are trained.

2.4.2 GENIE3

We consider GENIE3 as the blueprint of "multiple regression GRN inference" strategy.

2.4.3 DREAM5 benchmark

2.5 Concept and Background

Arboreto was conceived to address the need for a faster alternative for the classic GENIE3 implementation for inferring gene regulatory networks from high-throughput gene expression profiles.

In summary, GENIE3 performs a number of independent learning tasks. This inference "architecture" suggests two approaches for speeding up the algorithm:

1. Speeding up the individual learning tasks.
2. Specifying the task coordination logic so that the tasks can be executed in parallel on distributed hardware.

2.6 FAQ

2.6.1 Q: How can I use the Dask diagnostics (bokeh) dashboard?

Dask distributed features a nice web interface for monitoring the execution of a Dask computation graph.

By default, when no custom Client is specified, Arboreto creates a LocalCluster instance with the diagnostics dashboard **disabled**:

```
...
local_cluster = LocalCluster(diagnostics_port=None)
client = Client(local_cluster)
...
```

You can easily create a custom LocalCluster, with the dashboard enabled, and pass a custom Client connected to that cluster to the GRN inference algorithm:

```
local_cluster = LocalCluster()  # diagnostics dashboard is enabled
custom_client = Client(local_cluster)

...

network = grnboost2(expression_data=ex_matrix,
                    tf_names=tf_names,
                    client=custom_client)  # specify the custom client
```

By default, the dashboard is available on port `8787`.

For more information, consult:

- Dask web interface documentation
- Running with a custom Dask Client

2.6.2 Q: My gene expression matrix is transposed, what now?

The Python scikit-learn library expects data in a format where rows represent observations and columns represent features (in our case: genes), for example, see the GradientBoostingRegressor API.

However, in some fields (like single-cell genomics), the default is inversed: the rows represent genes and the columns represent the observations.

In order to maintain an API that is as lean is possible, Arboreto adopts the scikit-learn convention (rows=observations, columns=features). This means that the user is responsible for providing the data in the right shape.

Fortunately, the Pandas and Numpy libraries feature all the necessary functions to preprocess your data.

Example: reading a transposed text file with Pandas

```
df = pd.read_csv(<ex_path>, index_col=0, sep='\t').T
```

Caution: Don't carelessly copy/paste above snippet. Take into account absence or presence of 1 or multiple header lines in the file.

Always check whether the your DataFrame has the expected dimensions!

```
In[10]: df.shape

Out[10]: (17650, 14086)  # example
```

2.6.3 Q: Different runs produce different network outputs, why?

Both GENIE3 and GRNBoost2 are based on stochastic machine learning techniques, which use a random number generator internally to perform random sub-sampling of observations and features when building decision trees.

To stabilize the output, Arboreto accepts a seed value that is used to initialize the random number generator used by the machine learning algorithms.

```
network_df = grnboost2(expression_data=ex_matrix,
                       tf_names=tf_names,
                       seed=777)
```

2.7 Troubleshooting

2.7.1 Bokeh error when launching Dask scheduler

```
vsc12345@r6i0n5 ~ 12:00 $ dask-scheduler

distributed.scheduler - INFO - -----------------------------------------------
distributed.scheduler - INFO - Could not launch service: ('bokeh', 8787)
Traceback (most recent call last):
File "/data/leuven/software/biomed/Anaconda/5-Python-3.6/lib/python3.6/site-packages/
↪distributed/scheduler.py", line 430, in start_services
    service.listen((listen_ip, port))
    File "/data/leuven/software/biomed/Anaconda/5-Python-3.6/lib/python3.6/site-
↪packages/distributed/bokeh/core.py", line 31, in listen
        **kwargs)
File "/data/leuven/software/biomed/Anaconda/5-Python-3.6/lib/python3.6/site-packages/
↪bokeh/server/server.py", line 371, in __init__
    tornado_app = BokehTornado(applications, extra_websocket_origins=extra_websocket_
↪origins, prefix=self.prefix, **kwargs)
TypeError: __init__() got an unexpected keyword argument 'host'
distributed.scheduler - INFO -   Scheduler at: tcp://10.118.224.134:8786
distributed.scheduler - INFO -        http at:                      :9786
distributed.scheduler - INFO - Local Directory:    /tmp/scheduler-y6b8mnih
```

(continues on next page)

(continued from previous page)

```
distributed.scheduler - INFO - -----------------------------------------------
distributed.scheduler - INFO - Receive client connection: Client-7b476bf6-c6d8-11e7-
↪b839-a0040220fe80
distributed.scheduler - INFO - End scheduler at 'tcp://:8786'
```

- **known error**: see Github issue (closed), fixed in Dask.distributed version `0.20.0`
- **workaround**: launch with bokeh disabled: `dask-scheduler --no-bokeh`
- **solution**: upgrade to Dask distributed `0.20.0` or higher

2.7.2 Workers do not connect with Dask scheduler

We have observed that sometimes when running the `dask-worker` command, the workers start but no connections are made to the scheduler.

Solutions:

- delete the `dask-worker-space` directory before starting the workers.
- specifying the `local_dir` (with enough space) when instantiating a Dask

distributed `Client`:

```
>>> from dask.distributed import Client, LocalCluster
>>> worker_kwargs = {'local_dir': '/tmp'}
>>> cluster = LocalCluster(**worker_kwargs)
>>> client = Client(cluster)
>>> client

<Client: scheduler='tcp://127.0.0.1:41803' processes=28 cores=28>
```

- **Github issue**: https://github.com/dask/distributed/issues/1707

2.8 LCB Notes

This page contains additional documentation relevant for the Stein Aerts Lab of Computation Biology (LCB).

2.8.1 VSC access

First you will need access to the VSC front nodes. For this, a VSC account is required plus additional ssh configuration.

Tip: Kindly ask Gert for assistance setting up your ssh configuration for the VSC using the `https://git.aertslab.org/connect_to_servers/` script.

Front nodes

We will work with following machines:

Alias	HostName	CPU	Memory
hpc2-big1	r10n1	10 core (20 threads)	256 GB
hpc2-big2	r10n2	10 core (20 threads)	256 GB
hpc2-big3	r6i0n5	2x 12-core (48 threads)	512 GB
hpc2-big4	r6i0n12	2x 12-core (48 threads)	512 GB
hpc2-big5	r6i0n13	2x 12-core (48 threads)	512 GB
hpc2-big6	r6i1n12	2x 12-core (48 threads)	512 GB
hpc2-big7	r6i1n13	2x 12-core (48 threads)	512 GB

The aliases are the ones defined by the `https://git.aertslab.org/connect_to_servers/` script.

2.8.2 Running Arboreto on the front nodes

Following section describes the steps requires for inferring a GRN using Arboreto in distributed mode, using the front nodes.

Tip: Setting up a Dask.distributed cluster requires ssh access to multiple nodes. We recommend using a terminal multiplexer tool like tmux for managing multiple ssh sessions.

On the VSC, tmux is available by loading following module:

```
$ module load tmux/2.5-foss-2014a
```

We will set up a cluster using about half the CPU resources of the 5 larger nodes (`hpc2-big3` to `hpc2-big7`). One of the large nodes will also host the Dask scheduler. One a smaller node, we run a Jupyter notebook server from which we run the GRN inference using Arboreto.

0. Software preparation

As recommended in the Installation Guide, we will use an Anaconda distribution. On the front nodes we do this by loading a module:

Listing 6: `vsc12345@r6i0n5`

```
$ module load Anaconda/5-Python-3.6
```

We obviously need Arboreto (make sure you have the latest version):

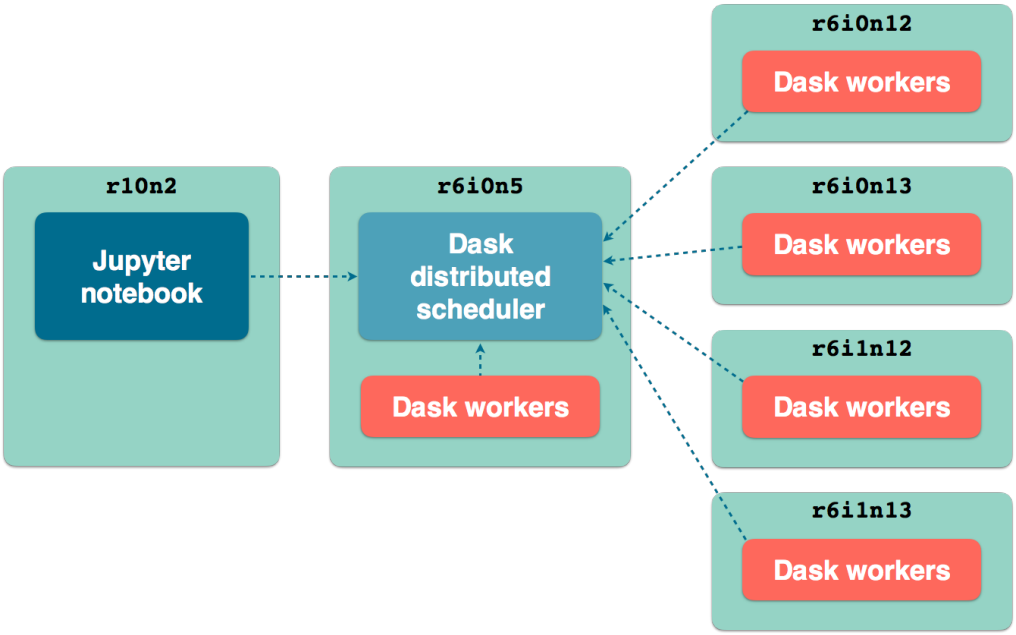


Fig. 1: LCB front nodes distributed architecture

Listing 7: `vsc12345@r6i0n5`

```
$ pip install arboreto

$ pip show arboreto

Name: arboreto
Version: 0.1.5
Summary: Scalable gene regulatory network inference using tree-based ensemble␣
↪regressors
Home-page: https://github.com/tmoerman/arboreto
Author: Thomas Moerman
Author-email: thomas.moerman@gmail.com
License: BSD 3-Clause License
Location: /vsc-hard-mounts/leuven-data/software/biomed/Anaconda/5-Python-3.6/lib/
↪python3.6/site-packages
Requires: scikit-learn, dask, numpy, scipy, distributed, pandas
```

We now proceed with launching the Dask scheduler and workers. Make sure that on the nodes, the Anaconda module was loaded like explained above.

1. Starting the Dask scheduler

On node `r6i0n5`, we launch the Dask scheduler.

Listing 8: `vsc12345@r6i0n5`

```
$ dask-scheduler

distributed.scheduler - INFO - -----------------------------------------------
↪
↪                            distributed.worker - INFO -         Registered to:  tcp:/
↪/10.118.224.134:8786
distributed.scheduler - INFO -   Scheduler at: tcp://10.118.224.134:8786
↪
↪                            distributed.worker - INFO - ------------------------------
↪---------------------
distributed.scheduler - INFO -       bokeh at:                     :35874
↪
↪                            distributed.worker - INFO -         Registered to:  tcp:/
↪/10.118.224.134:8786
distributed.scheduler - INFO - Local Directory:    /tmp/scheduler-wu5odlrh
↪
↪                            distributed.worker - INFO - ------------------------------
↪---------------------
distributed.scheduler - INFO - -----------------------------------------------
```

The command launches 2 services:

- The Dask scheduler on address: `tcp://10.118.224.134:8786`
- The Dask diagnostics dashboard on address: `tcp://10.118.224.134:35874`

Tip: The Dask diagnostics dashboard is useful for monitoring the progress of long-running Dask jobs. In order to view the dashboard, which runs on the VSC front node `r6i0n5`, use ssh port forwarding as follows:

```
ssh -L 8787:localhost:35874 hpc2-big3
```

You can now view the Dask dashboard on url: `http://localhost:8787`.

---

2. Adding workers to the scheduler

We will need the scheduler address: `tcp://10.118.224.134:8786` (highlighted above) when launching worker processes connected to the scheduler.

First, we launch 24 worker processes on the same machine where the scheduler is running:

Listing 9: `vsc12345@r6i0n5`

```
$ nice -n 10 dask-worker tcp://10.118.224.134:8786 --nprocs 24 --nthreads 1
```

The command above consists of several parts, let's briefly discuss them:

- `nice -n 10`

  Setting a nice value of higher than 0 gives the process a lower priority, which is sometimes desirable to not highjack the resources on compute nodes used by multiple users.

  Setting a nice value is **entirely optional** and up to the person setting up the distributed network. You can safely omit this.

- `dask-worker tcp://10.118.224.134:8786 --nprocs 24 --nthreads 1`

Spins up 24 worker processes with 1 thread per process. For Arboreto, it is recommended to always set `--nthreads 1`.

In this case we have chosen 24 processes because we planned to use only half the CPU capacity of the front nodes.

In the terminal where the scheduler was launched, you should see messages indicating workers have been connected to the scheduler:

```
distributed.scheduler - INFO - Register tcp://10.118.224.134:43342
distributed.scheduler - INFO - Starting worker compute stream, tcp://10.118.224.
↪134:43342
```

We now repeat the same command on the other compute nodes that will run Dask worker processes:

Listing 10: `vsc12345@r6i0n12`

```
$ nice -n 10 dask-worker tcp://10.118.224.134:8786 --nprocs 24 --nthreads 1
```

Listing 11: `vsc12345@r6i0n13`

```
$ nice -n 10 dask-worker tcp://10.118.224.134:8786 --nprocs 24 --nthreads 1
```

Listing 12: `vsc12345@r6i1n12`

```
$ nice -n 10 dask-worker tcp://10.118.224.134:8786 --nprocs 24 --nthreads 1
```

Listing 13: `vsc12345@r6i1n13`

```
$ nice -n 10 dask-worker tcp://10.118.224.134:8786 --nprocs 24 --nthreads 1
```

3. Running Arboreto from a Jupyter notebook

So far, we have a scheduler running with 5*24 worker processes connected to it and a diagnostics dashboard. Let's now run a Jupyter notebook or Jupyter Lab server so that we can interact with the Dask cluster from within a Jupyter environment.

Listing 14: `vsc12345@r10n2`

```
$ jupyter lab --port 9999 --no-browser

[I 12:16:08.725 LabApp] JupyterLab alpha preview extension loaded from /data/leuven/
↪software/biomed/Anaconda/5-Python-3.6/lib/python3.6/site-packages/jupyterlab
JupyterLab v0.27.0
Known labextensions:
[I 12:16:08.739 LabApp] Running the core application with no additional extensions or␣
↪settings
[I 12:16:08.766 LabApp] Serving notebooks from local directory: /ddn1/vol1/staging/
↪leuven/stg_00002/lcb/tmoerman/nb
[I 12:16:08.766 LabApp] 0 active kernels
[I 12:16:08.766 LabApp] The Jupyter Notebook is running at:
[I 12:16:08.766 LabApp] http://localhost:9999/?
↪token=2dca6ce946265895846795c4983191c9f76ba954f414efdf
[I 12:16:08.766 LabApp] Use Control-C to stop this server and shut down all kernels␣
↪(twice to skip confirmation).
[C 12:16:08.767 LabApp]
```

(continues on next page)

(continued from previous page)

```
    Copy/paste this URL into your browser when you connect for the first time,
    to login with a token:
        http://localhost:9999/?token=2dca6ce946265895846795c4983191c9f76ba954f414efdf
```

Again, use ssh port forwarding to access the notebook server. Execute following command in a shell on your *local* machine:

Listing 15: `localhost`

```
$ ssh -L 9999:localhost:9999 hpc2-big2
```

To access the notebook open a browser and navigate to following url:

```
http://localhost:9999/?token=2dca6ce946265895846795c4983191c9f76ba954f414efdf
```

Note: Using Jupyter is **entirely optional**. Everything explained in the following section is equally applicable to running Arboreto from a simple Python session or script.

As an example, please consider this script. Remember that the main code should be in a code block protected by:

```
if __name__ == '__main__':
    # ... code ...
```

Now we are ready to create a new notebook in Jupyter and write some Python code to check whether the cluster was set up correctly:

```
In [1]: from distributed import Client

In [2]: client = Client('tcp://10.118.224.134:8786')

In [3]: client

Out[3]:

    Client
    * Scheduler: tcp://10.118.224.134:8786
    * Dashboard: http://10.118.224.134:35874

    Cluster
    * Workers: 120
    * Cores: 120
    * Memory: 1354.63 GB
```

The cluster is set up and ready for Arboreto GRN inference work. Please review the section Running with a Dask distributed scheduler on how to use Arboreto in distributed mode.

To run in distributed mode, we need to make one modification to the code launching the inference algorithm: specifying `client_or_address` in the (in this case) `genie3` function:

```
network_df = genie3(expression_data=ex_matrix,
                    tf_names=tf_names,
                    client_or_address=client)
```

While our computation is running, we can consult the Dask diagnostics dashboard to monitor progress. Point a browser to `localhost:8787/status`, you should see a dynamic visualization like this:

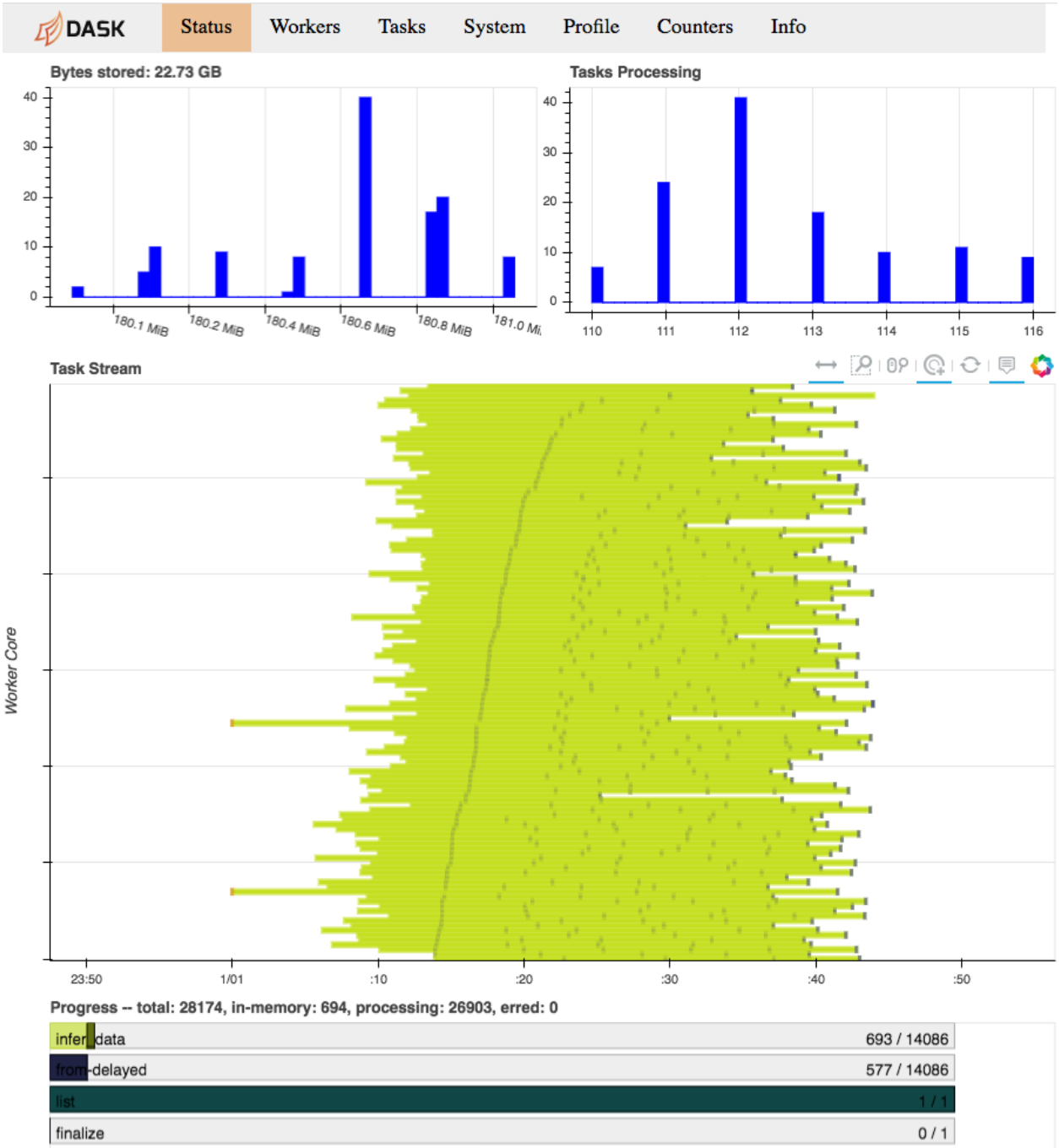


Fig. 2: Dask diagnostics dashboard visualizing Arboreto progress

Note the progress gauges in the bottom:

`infer_data` -> `693 / 14086` means that 693 out of 14086 inference steps have been completed so far. As the inference steps entail almost the entire workload of the algorithm, this is a pretty accurate progress indicator.