RADICAL-Analytics

Release 1.34.0

Jun 22, 2023

Contents

RADICAL-Analytics (RA) is a library implemented in Python to support the analysis of traces produced by RADICAL-Cybertools (RCT). Each RCT has a set of entities and a set of events associated to those entities. Each component of each RCT tool records a set of events at runtime, enabling *post-mortem* analysis.

Currently, RA supports two RCT, RADICAL-Pilot (RP) and RADICAL-EnTK (EnTK), and three event-based analyses: **duration**, **concurrency** and **utilization**. All the analyses work with pairs of arbitrarily-defined events. Duration analysis calculates the amount of time spent by one or more entities between two events. Concurrency analysis shows between which events one or more entity was in a given interval of time, and utilization analysis shows for how much time each available resource was used during runtime.

RA enables developing statistical analysis of experimental data, collected via multiple experimental runs. For example, RA supports calculation of averages, spread, and skew among durations of repeated runs, to compare groups of diverse types of entities, association among variables, and analysis of dependent variables. RA also enables introspecting the behavior of RP or EnTK, measuring and characterizing percentage of resource utilization, information flow, and distribution patterns.

RA supports the development and experimental analysis of the papers published by RADICAL at Rutgers University.

- repository: https://github.com/radical-cybertools/radical.analytics
- issues: https://github.com/radical-cybertools/radical.analytics/issues

CHAPTER 1

Introduction

RADICAL-Analytics (RA) is a library implemented in Python to support the analysis of traces produced by RADICAL-Cybertools (RCT). Using RA requires knowing the architecture and the event model of the chosen RCT tool. Without that knowledge, you will not be able to choose the events that are relevant to your analysis and to understand how the results of your analysis relate to the inner working of the chosen RCT tool.

Depending on the chosen RCT, **an understanding of the following document is precondition to the use of RA**:

1. RP architecture (outdated as for Aug 2020)
2. RP event model
3. EnTK architecture
4. EnTK event model

Note: States are special types of events. Given two states in a sequence <1, 2>, both states are always recorded at runtime and state 1 always precede state 2.

1.1 Using RA

RA supports *post-mortem* analysis:

1. Install RA and RADICAL-Pilot (RP) and/or RADICAL-EnTK.
2. Write an application in Python to execute a workload (RP) or a workflow (EnTK) on an high-performance computing (HPC) platform.
3. Set the environment variable `RADICAL_PROFILE` to `TRUE` with the command `export RADICAL_PROFILE="TRUE"`.
4. Execute your application.
5. Both RP and EnTK write traces (i.e., timestamped sequences of events) to a directory called `client sandbox`. This directory is created inside the directory from which you executed your application. The

name of the client sandbox is a session ID, e.g., `rp.session.hostname.username.018443.0002` or `en.session.hostname.username.018443.0002`.

6. Load the session traces in RA by creating an object `ra.Session`.
7. Measure entity-level or session-level durations, concurrency or resource utilization, using RA *API*.

1.2 Fundamental Notions

- **Session**: set of events generated by a single run of a RP or EnTK application. RA creates an object Session containing all the relevant information about all the events recoded at runtime by RP or EnTK. The Session object contains also information about the execution environment.
- **Entity**: object exposed by RP or EnTK. Currently, RP exposes two types of entity—Pilot and Task—while EnTK exposes three types of entity—Pipeline, Stage and Task. An instance of an entity type is an actual pilot, task, pipeline, stage or task.
- **Describing**: session and entity instances can be described by listing their properties. For example, a session instance has properties like list of a type of entity, list of events, list of timestamps for those events. A task instance has proprieties like the events of that specific instance, the timestamps of those specific events.
- **Filtering**: selecting a subset of properties of a session. This is particularly important when we want to limit an analysis to a specific type of entity. For example, assume that we want to measure the amount of time spent by the tasks waiting to be scheduled. We will want to filter the session so to have only entities of type Task in the session. Then, we will perform our measure only on those entities.

Warning: It is important to stress that description and filtering are performed on **instances** of entities. This means that if we filter for, say, the event `DONE` and all the tasks have failed, RA will return an empty list as none of task instances will have the event `DONE` as their property.

1.3 Types of Analysis

RA enables both **local** and **global** analyses. Local analyses pertain to a single instance of an entity. Currently, RP supports two entities (Pilot and Task) and EnTK supports three entities (Pipeline, Stage and Task).

Global analyses pertain to a set of entities, including all the entities of a run. For example, a very common global analysis consists of measuring the total time all the tasks took to execute. It is fundamental to note that this is **NOT** the sum of the execution time of all the tasks. Tasks execute with varying degree of concurrency, depending on resource availability.

1.4 Types of Measure

RA is agnostic towards the tools used to perform the measurements. For example, RA supports writing stand-alone Python scripts, wranglers or being loaded into a Jupyter Notebook. RA offers classes and methods to perform three types of measures:

1. **Duration**: measures the time spent by an instance of an entity (local analyses) or a set of instances of an entity (global analyses) between two timestamps. For example, staging, scheduling, pre-execute, execute time of one or more tasks; description, submission and execution time of one or more pipelines or stages; and runtime of one or more pilots.

2. **Concurrency**: measures the number of entities of the same type that are between two given events in a time range during the execution. For example, this measures how many tasks where scheduled in a time range. Note that the time range here can be as large as the whole runtime of the application.
3. **Utilization**: measures the amount of time a resource has been provided and consumed. In this context, resource indicates an hardware thread, a CPU core or a GPU. When measured for each resource, we can derive the percentage of utilization of all the resources available.

Note: Utilization is available only for RP as EnTK does not directly utilize resources but delegates that to RP.

Warning: Utilization is still under development so, for example, at the moment it does not offer an easy way to discriminate about types of resources.

CHAPTER 2

Installation

RADICAL-Analytics (RA) is a Python module. RA must be installed in a virtual environment. Site-wide installation will not work.

RA requires the following packages:

- Python >= 3.6
- virtualenv >= 20
- pip >= 20
- radical.utils >= 1.4

RA automatically installs the dependencies above. Besides that, RA requires the manual installation of the RADICAL-Cybertool (RCT) of choice.

2.1 Virtual Environment

To install RA in a virtual environment, open a terminal and run:

```
virtualenv -p python3 $HOME/ve
source $HOME/ve/bin/activate
pip install radical.analytics
```

Run the following to make sure that RA is properly installed:

```
radical-analytics-version
```

This command should print the version and release numbers of the `radical.analytics` package. For example:

```
$ radical-analytics-version
1.6.7
```

RA installation is now complete.

2.2 Troubleshooting

Missing virtualenv

If virtualenv **is not** installed on your system, you can try the following.

```
pip install git+https://github.com/pypa/virtualenv.git@master
```

Installation Problems

Many installation problems boil down to one of two causes: an Anaconda based Python distribution, or an incompatible version of pip/setuptools.

Many recent systems, specifically in the academic community, install Python in its incarnation as Anaconda Distribution. RA is not yet able to function in that environment. While support of Anaconda is planned in the near future, you will have to revert to a 'normal' Python distribution to use RADICAL-Analytics.

Python supports a large variety of module deployment paths: `easy_install`, `setuptools` and `pip` being the most prominent ones for non-compilable modules. RA only supports `pip`.

Reaching out to the RADICAL devel team

If you encounter any issue, please do not hesitate to contact us by opening an issue at https://github.com/radical-cybertools/radical.analytics/issues.

CHAPTER 3

Plotting

RADICAL-Analytics does not provide plotting primitives. Instead, it offers helper methods that can be used with 3rd party plotting libraries.

3.1 Matplotlib

RADICAL-Analytics provides a style for Matplotlib. Loading it guarantees an uniform look&feel across plots. The style is located at `styles/radical_mpl.txt`.

3.1.1 Loading RADICAL-Analytics Style

```
import matplotlib.pyplot as plt
import radical.analytics as ra

plt.style.use(ra.get_mplstyle("radical_mpl")
```

3.1.2 Default Color Cycler of RADICAL-Analytics Style

01. 02. 03. 04. 05. 06. 07. 08. 09. 10. 11. 12. 13. 14. 15. 16. 17. 18. 19. 20. 21. 22. 23. 24.

3.2 Plotting for Latex Documents

In LaTeX documents, scaling images make the overall look&feel of a plot difficult to predict. Often, fonts are too small or too large, lines, bars, dots and axes too thin or too thick, and so on. Thus, plots should not be scaled in LaTeX—e.g., `width=0.49\textwidth` should not be used to scale a figure down of 50%—but, instead, plots should be created with the exact size of a column or a page. Column and page sizes depends on the `.sty` used for the LaTeX document and need to be inspected in order to know how to size a plot. Further, plots need to have their own style so that size, color, font face and overall features are consistent, readable and "pleasant" to look at.

3.2.1 Workflow with Matplotlib and Latex

The following assume the use of Matplotlib to create a plot that needs to be added to a LaTeX document for publication.

1. Create a laTeX document using the following template:

```
\documentclass{<your_style_eg_IEEEtran>}

\newcommand{\recordvalue}[1]{%
  \typeout{%
    === Value of \detokenize{#1}: \the#1%
  }%
}

\begin{document}
  % gives the width of the current document in pts
  \recordvalue{\textwidth}
  \recordvalue{\columnwidth}
\end{document}
```

2. Compile your LaTeX document—e.g., pdlatex your_document—and note down the size of the text and of the column expressed in points (pts). An example output is shown below (shortened):

```
$ pdflatex test.tex
This is pdfTeX, [...]
[...]
=== Value of \textwidth  : 252.0pt
=== Value of \columnwidth: 516.0pt
(./test.aux) )
No pages of output.
Transcript written on test.log.
```

3. Use `ra.set_size()` to compute the exact size of your plot. For a plot with a single figure that span the width of a IEEtran LaTeX column:

```
fig, ax = plt.subplots(figsize=ra.get_plotsize(252))
```

   for plot with 1 row and 3 subplots that spans the whole width of a IEEtran LaTeX page:

```
fig, axarr = plt.subplots(1, 3, figsize=(ra.set_size(516)))
```

CHAPTER 4

Inspection

RADICAL-Analytics enables deriving information about RCT sessions, pilots and tasks. For example, session ID, number of tasks, number of pilots, final state of the tasks and pilots, CPU/GPU processes for each task, etc. That information allows to derive task requirements and resource capabilities, alongside the RCT configuration parameters used for a session.

4.1 Prologue

Load the Python modules needed to profile and plot a RADICAL-Cybertool (RCT) session.

[1]:
```
import os
import tarfile

import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker

import radical.utils as ru
import radical.pilot as rp
import radical.entk as re
import radical.analytics as ra
```

Load the RADICAL Matplotlib style to obtain viasually consistent and publishable-qality plots.

[2]:
```
plt.style.use(ra.get_mplstyle('radical_mpl'))
```

Usually, it is useful to record the stack used for the analysis.

Note: The analysis stack might be different from the stack used to create the session to analyze. Usually, the two stacks must have the same minor release number (Major.Minor.Patch) in order to be compatible.

[3]:
```
! radical-stack
```

```
  python               : /home/docs/checkouts/readthedocs.org/user_builds/
↪radicalanalytics/envs/stable/bin/python3
  pythonpath           :
  version              : 3.9.15
  virtualenv           :

  radical.analytics    : 1.34.0-v1.34.0@HEAD-detached-at-0b58be0
  radical.entk         : 1.33.0
  radical.gtod         : 1.20.1
  radical.pilot        : 1.34.0
  radical.saga         : 1.34.0
  radical.utils        : 1.33.0
```

4.2 Single Session

Name and location of the session we profile.

[4]:
```
sidsbz2 = !find sessions -maxdepth 1 -type f -exec basename {} \;
sids = [s[:-8] for s in sidsbz2]
sdir = 'sessions/'
```

Unbzip and untar the session.

[5]:
```
sidbz2 = sidsbz2[0]
sid = sidbz2[:-8]
sp  = sdir + sidbz2

tar = tarfile.open(sp, mode='r:bz2')
tar.extractall(path=sdir)
tar.close()
```

Create a `ra.Session` object for the session. We do not need EnTK-specific traces so load only the RP traces contained in the EnTK session. Thus, we pass the `'radical.pilot'` session type to `ra.Session`.

Warning: We already know we need information about pilots and tasks. Thus, we save in memory two session objects filtered for pilots and tasks. This might be too expensive with large sessions, depending on the amount of memory available.

Note: We save the ouput of `ra.Session` in `capt` to avoid polluting the notebook with warning messages.

[6]:
```
%%capture capt

sp = sdir + sid

session = ra.Session(sp, 'radical.pilot')
pilots  = session.filter(etype='pilot', inplace=False)
tasks   = session.filter(etype='task' , inplace=False)
```

Information about **session** that is commonly used when analyzing and plotting one or more RCT sessions.

[7]:
```
# Session info
sinfo = {
    'sid'       : session.uid,
    'hostid'    : session.get(etype='pilot')[0].cfg['hostid'],
    'cores_node': session.get(etype='pilot')[0].cfg['resource_details']['rm_info']['cores_per_node'],
    'gpus_node' : session.get(etype='pilot')[0].cfg['resource_details']['rm_info']['gpus_per_node'],
    'smt'       : session.get(etype='pilot')[0].cfg['resource_details']['rm_info']['threads_per_core']
}

# Pilot info (assumes 1 pilot)
sinfo.update({
    'pid'       : pilots.list('uid'),
    'npilot'    : len(pilots.get()),
    'npact'     : len(pilots.timestamps(state='PMGR_ACTIVE')),
})

# Task info
sinfo.update({
    'ntask'     : len(tasks.get()),
    'ntdone'    : len(tasks.timestamps(state='DONE')),
    'ntcanceled': len(tasks.timestamps(state='CANCELED')),
    'ntfailed'  : len(tasks.timestamps(state='FAILED')),
})

# Derive info (assume a single pilot)
sinfo.update({
    'pres'      : pilots.get(uid=sinfo['pid'])[0].description['resource'],
    'ncores'    : pilots.get(uid=sinfo['pid'])[0].description['cores'],
    'ngpus'     : pilots.get(uid=sinfo['pid'])[0].description['gpus']
})
sinfo.update({
    'nnodes'    : int(sinfo['ncores']/sinfo['cores_node'])
})

sinfo
```

[7]:
```
{'sid': 'rp.session.mosto.mturilli.019432.0003',
 'hostid': 'mosto',
 'cores_node': 64,
 'gpus_node': 8,
 'smt': 1,
 'pid': ['pilot.0000'],
 'npilot': 1,
 'npact': 1,
 'ntask': 2048,
 'ntdone': 2048,
 'ntcanceled': 0,
 'ntfailed': 0,
 'pres': 'local.localhost',
 'ncores': 512,
 'ngpus': 64,
 'nnodes': 8}
```

Information about **tasks** that is commonly used when analyzing and plotting one or more RCT sessions.

Note: we use `ra.entity.description` to get each task description as a dictionary. We then select the keys of that dictionary that contain the task requirements. More keys are available, especially those about staged input/output files.

[8]:
```
tinfo = []
for task in tasks.get():

    treq = {
        'executable'       : task.description['executable'],
        'cpu_process_type' : task.description['cpu_process_type'],
        'cpu_processes'    : task.description['cpu_processes'],
        'cpu_thread_type'  : task.description['cpu_thread_type'],
        'cpu_threads'      : task.description['cpu_threads'],
        'gpu_process_type' : task.description['gpu_process_type'],
        'gpu_processes'    : task.description['gpu_processes'],
        'gpu_thread_type'  : task.description['gpu_thread_type'],
        'gpu_threads'      : task.description['gpu_threads']
    }

    if not tinfo:
        treq['n_of_tasks'] = 1
        tinfo.append(treq)
        continue

    for i, ti in enumerate(tinfo):
        counter = ti['n_of_tasks']
        ti.pop('n_of_tasks')

        if ti == treq:
            counter += 1
            tinfo[i]['n_of_tasks'] = counter
        else:
            treq['n_of_tasks'] = 1
            tinfo.append(treq)
tinfo
```

[8]:
```
[{'executable': '/home/mturilli/github/radical.analytics/docs/source/bin/radical-
↪pilot-hello.sh',
  'cpu_process_type': '',
  'cpu_processes': 0,
  'cpu_thread_type': '',
  'cpu_threads': 0,
  'gpu_process_type': '',
  'gpu_processes': 0,
  'gpu_thread_type': '',
  'gpu_threads': 0,
  'n_of_tasks': 2048}]
```

4.3 Multiple Sessions

Unbzip and untar those sessions.

[9]:
```
for sid in sids:
    sp = sdir + sid + '.tar.bz2'
```

(continues on next page)

(continued from previous page)

```
    tar = tarfile.open(sp, mode='r:bz2')
    tar.extractall(path=sdir)
    tar.close()
```

Create the session, tasks and pilots objects for each session.

[10]:
```
%%capture capt

ss = {}
for sid in sids:
    sp = sdir + sid
    ss[sid] = {'s': ra.Session(sp, 'radical.pilot')}
    ss[sid].update({'p': ss[sid]['s'].filter(etype='pilot', inplace=False),
                    't': ss[sid]['s'].filter(etype='task' , inplace=False)})
```

[11]:
```
for sid in sids:
    ss[sid].update({'sid'       : ss[sid]['s'].uid,
                    'hostid'    : ss[sid]['s'].get(etype='pilot')[0].cfg['hostid'],
                    'cores_node': ss[sid]['s'].get(etype='pilot')[0].cfg['resource_
↪details']['rm_info']['cores_per_node'],
                    'gpus_node' : ss[sid]['s'].get(etype='pilot')[0].cfg['resource_
↪details']['rm_info']['gpus_per_node'],
                    'smt'       : ss[sid]['s'].get(etype='pilot')[0].cfg['resource_
↪details']['rm_info']['threads_per_core']
    })

    ss[sid].update({
                    'pid'       : ss[sid]['p'].list('uid'),
                    'npilot'    : len(ss[sid]['p'].get()),
                    'npact'     : len(ss[sid]['p'].timestamps(state='PMGR_ACTIVE'))
    })

    ss[sid].update({
                    'ntask'     : len(ss[sid]['t'].get()),
                    'ntdone'    : len(ss[sid]['t'].timestamps(state='DONE')),
                    'ntfailed'  : len(ss[sid]['t'].timestamps(state='FAILED')),
                    'ntcanceled': len(ss[sid]['t'].timestamps(state='CANCLED'))
    })


    ss[sid].update({'pres'      : ss[sid]['p'].get(uid=ss[sid]['pid'])[0].description[
↪'resource'],
                    'ncores'    : ss[sid]['p'].get(uid=ss[sid]['pid'])[0].description[
↪'cores'],
                    'ngpus'     : ss[sid]['p'].get(uid=ss[sid]['pid'])[0].description[
↪'gpus']
    })

    ss[sid].update({'nnodes'    : int(ss[sid]['ncores']/ss[sid]['cores_node'])})
```

For presentation purposes, we can convert the session information into a DataFrame and rename some of the columns to improve readability.

[12]:
```
ssinfo = []
for sid in sids:
    ssinfo.append({'session'   : sid,
```

(continues on next page)

(continued from previous page)

```
                    'resource'  : ss[sid]['pres'],
                    'cores_node': ss[sid]['cores_node'],
                    'gpus_node' : ss[sid]['gpus_node'],
                    'pilots'    : ss[sid]['npilot'],
                    'ps_active' : ss[sid]['npact'],
                    'cores'     : int(ss[sid]['ncores']/ss[sid]['smt']),
                    'gpus'      : ss[sid]['ngpus'],
                    'nodes'     : ss[sid]['nnodes'],
                    'tasks'     : ss[sid]['ntask'],
                    't_done'    : ss[sid]['ntdone'],
                    't_failed'  : ss[sid]['ntfailed']})

df_info = pd.DataFrame(ssinfo)
df_info
```

[12]:

	session	resource	cores_node	gpus_node	pilots	ps_active	cores	gpus	nodes	tasks	t_done	t_failed
0	rp.session.mosto.mturilli.019432.0003	local.localhost	64	8	1	1	512	64	8	2048	2048	0
1	rp.session.mosto.mturilli.019432.0004	local.localhost	64	8	1	1	1024	128	16	2048	2048	0
2	rp.session.mosto.mturilli.019432.0002	local.localhost	64	8	1	1	256	32	4	2048	2048	0
3	rp.session.mosto.mturilli.019432.0005	local.localhost	64	8	1	1	2048	256	32	2048	2048	0

We can then derive task information for each session.

[13]:
```
tsinfo = {}
for sid in sids:

    tsinfo[sid] = []
    for task in tasks.get():

        treq = {
            'executable'       : task.description['executable'],
            'cpu_process_type' : task.description['cpu_process_type'],
            'cpu_processes'    : task.description['cpu_processes'],
            'cpu_thread_type'  : task.description['cpu_thread_type'],
            'cpu_threads'      : task.description['cpu_threads'],
            'gpu_process_type' : task.description['gpu_process_type'],
            'gpu_processes'    : task.description['gpu_processes'],
            'gpu_thread_type'  : task.description['gpu_thread_type'],
            'gpu_threads'      : task.description['gpu_threads']
        }

        if not tsinfo[sid]:
            treq['n_of_tasks'] = 1
            tsinfo[sid].append(treq)
            continue

        for i, ti in enumerate(tsinfo[sid]):
            counter = ti['n_of_tasks']
            ti.pop('n_of_tasks')

            if ti == treq:
```

(continues on next page)

(continued from previous page)

```
                counter += 1
                tsinfo[sid][i]['n_of_tasks'] = counter
            else:
                treq['n_of_tasks'] = 1
                tsinfo[sid].append(treq)
tsinfo
```

[13]:
```
{'rp.session.mosto.mturilli.019432.0003': [{'executable': '/home/mturilli/github/
↪radical.analytics/docs/source/bin/radical-pilot-hello.sh',
   'cpu_process_type': '',
   'cpu_processes': 0,
   'cpu_thread_type': '',
   'cpu_threads': 0,
   'gpu_process_type': '',
   'gpu_processes': 0,
   'gpu_thread_type': '',
   'gpu_threads': 0,
   'n_of_tasks': 2048}],
 'rp.session.mosto.mturilli.019432.0004': [{'executable': '/home/mturilli/github/
↪radical.analytics/docs/source/bin/radical-pilot-hello.sh',
   'cpu_process_type': '',
   'cpu_processes': 0,
   'cpu_thread_type': '',
   'cpu_threads': 0,
   'gpu_process_type': '',
   'gpu_processes': 0,
   'gpu_thread_type': '',
   'gpu_threads': 0,
   'n_of_tasks': 2048}],
 'rp.session.mosto.mturilli.019432.0002': [{'executable': '/home/mturilli/github/
↪radical.analytics/docs/source/bin/radical-pilot-hello.sh',
   'cpu_process_type': '',
   'cpu_processes': 0,
   'cpu_thread_type': '',
   'cpu_threads': 0,
   'gpu_process_type': '',
   'gpu_processes': 0,
   'gpu_thread_type': '',
   'gpu_threads': 0,
   'n_of_tasks': 2048}],
 'rp.session.mosto.mturilli.019432.0005': [{'executable': '/home/mturilli/github/
↪radical.analytics/docs/source/bin/radical-pilot-hello.sh',
   'cpu_process_type': '',
   'cpu_processes': 0,
   'cpu_thread_type': '',
   'cpu_threads': 0,
   'gpu_process_type': '',
   'gpu_processes': 0,
   'gpu_thread_type': '',
   'gpu_threads': 0,
   'n_of_tasks': 2048}]}
```

CHAPTER 5

Duration

In RADICAL-Analytics (RA), `duration` is a general term to indicate a measure of the time spent by an entity (local analyses) or a set of entities (global analyses) between two timestamps. For example, data staging, scheduling, pre-executing, and executing time of one or more tasks; description, submission and execution time of one or more pipelines or stages; and runtime of one or more pilots.

We show two sets of default durations for RADICAL-Pilot (RP) and how to define arbitrary durations, depending on the specifics of a given analysis. We then see how to plot the most common durations

5.1 Prologue

Load the Python modules needed to profile and plot a RADICAL-Cybertool (RCT) session.

[1]:
```
import os
import tarfile

import numpy as np
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker

import radical.utils as ru
import radical.pilot as rp
import radical.analytics as ra

from radical.pilot import states as rps
```

Load the RADICAL Matplotlib style to obtain viasually consistent and publishable-qality plots.

[2]:
```
plt.style.use(ra.get_mplstyle('radical_mpl'))
```

Usually, it is useful to record the stack used for the analysis.

Note: The analysis stack might be different from the stack used to create the session to analyze. Usually, the two stacks must have the same minor release number (Major.Minor.Patch) in order to be compatible.

[3]:
```
! radical-stack
```

```
  python               : /home/docs/checkouts/readthedocs.org/user_builds/
↪radicalanalytics/envs/stable/bin/python3
  pythonpath           :
  version              : 3.9.15
  virtualenv           :

  radical.analytics    : 1.34.0-v1.34.0@HEAD-detached-at-0b58be0
  radical.entk         : 1.33.0
  radical.gtod         : 1.20.1
  radical.pilot        : 1.34.0
  radical.saga         : 1.34.0
  radical.utils        : 1.33.0
```

5.2 Default Durations

Currently, we offer a set of default durations for pilot and task entities of RP.

[4]:
```
pd.DataFrame(ra.utils.tabulate_durations(rp.utils.PILOT_DURATIONS_DEBUG))
```

[4]:

	Duration Name	Start Timestamp	Stop Timestamp
0	p_pmgr_create	NEW	PMGR_LAUNCHING_PENDING
1	p_pmgr_launching_init	PMGR_LAUNCHING_PENDING	PMGR_LAUNCHING
2	p_pmgr_launching	PMGR_LAUNCHING	staging_in_start
3	p_pmgr_stage_in	staging_in_start	staging_in_stop
4	p_pmgr_submission_init	staging_in_stop	submission_start
5	p_pmgr_submission	submission_start	submission_stop
6	p_pmgr_scheduling_init	submission_stop	PMGR_ACTIVE_PENDING
7	p_pmgr_scheduling	PMGR_ACTIVE_PENDING	bootstrap_0_start
8	p_agent_ve_setup_init	bootstrap_0_start	ve_setup_start
9	p_agent_ve_setup	ve_setup_start	ve_setup_stop
10	p_agent_ve_activate_init	ve_setup_stop	ve_activate_start
11	p_agent_ve_activate	ve_activate_start	ve_activate_stop
12	p_agent_install_init	ve_activate_stop	rp_install_start
13	p_agent_install	rp_install_start	rp_install_stop
14	p_agent_launching	rp_install_stop	PMGR_ACTIVE
15	p_agent_terminate_init	PMGR_ACTIVE	cmd
16	p_agent_terminate	cmd	bootstrap_0_stop
17	p_agent_finalize	bootstrap_0_stop	DONE, CANCELED, FAILED
18	p_agent_runtime	bootstrap_0_start	bootstrap_0_stop

[5]:
```
pd.DataFrame(ra.utils.tabulate_durations(rp.utils.TASK_DURATIONS_DEBUG))
```

[5]:

	Duration Name	Start Timestamp	\
0	t_tmgr_create	NEW	
1	t_tmgr_schedule_queue	TMGR_SCHEDULING_PENDING	
2	t_tmgr_schedule	TMGR_SCHEDULING	
3	t_tmgr_stage_in_queue	TMGR_STAGING_INPUT_PENDING	

(continues on next page)

(continued from previous page)

```
4               t_tmgr_stage_in            TMGR_STAGING_INPUT
5        t_agent_stage_in_queue   AGENT_STAGING_INPUT_PENDING
6              t_agent_stage_in          AGENT_STAGING_INPUT
7        t_agent_schedule_queue      AGENT_SCHEDULING_PENDING
8              t_agent_schedule             AGENT_SCHEDULING
9         t_agent_execute_queue       AGENT_EXECUTING_PENDING
10      t_agent_execute_prepare               AGENT_EXECUTING
11        t_agent_execute_mkdir                    task_mkdir
12  t_agent_execute_layer_start               task_mkdir_done
13        t_agent_execute_layer                task_run_start
14             t_agent_lm_start                task_run_start
15            t_agent_lm_submit                  launch_start
16           t_agent_lm_execute                     exec_start
17              t_agent_lm_stop                      exec_stop
18      t_agent_stage_out_start                 task_run_stop
19      t_agent_stage_out_queue  AGENT_STAGING_OUTPUT_PENDING
20            t_agent_stage_out          AGENT_STAGING_OUTPUT
21         t_agent_push_to_tmgr   TMGR_STAGING_OUTPUT_PENDING
22               t_tmgr_destroy           TMGR_STAGING_OUTPUT
23           t_agent_unschedule              unschedule_start

                  Stop Timestamp
0        TMGR_SCHEDULING_PENDING
1                TMGR_SCHEDULING
2      TMGR_STAGING_INPUT_PENDING
3              TMGR_STAGING_INPUT
4    AGENT_STAGING_INPUT_PENDING
5            AGENT_STAGING_INPUT
6       AGENT_SCHEDULING_PENDING
7              AGENT_SCHEDULING
8        AGENT_EXECUTING_PENDING
9              AGENT_EXECUTING
10                  task_mkdir
11               task_mkdir_done
12                task_run_start
13                   task_run_ok
14                 launch_start
15                   exec_start
16                    exec_stop
17                task_run_stop
18  AGENT_STAGING_OUTPUT_PENDING
19          AGENT_STAGING_OUTPUT
20   TMGR_STAGING_OUTPUT_PENDING
21           TMGR_STAGING_OUTPUT
22        DONE, CANCELED, FAILED
23               unschedule_stop
```

Most of those default durations are meant for **debugging**. They are as granular as possible and (almost completely) contiguous. Only few of them are commonly used in experiment analyses. For example:

- **p_agent_runtime**: the amount of time for which one or more pilots (i.e., RP Agent) were active.
- **p_pmngr_scheduling**: the amount of time one or more pilots waited in the queue of the HPC batch system.
- **t_agent_stage_in**: the amount of time taken to stage the input data of one or more tasks.
- **t_agent_schedule**: the amount of time taken to schedule of one or more tasks.
- **t_agent_t_pre_execute**: the amount of time taken to execute the `pre_exec` of one or more tasks.

- **t_agent_t_execute**: the amount of time taken to execute the executable of one or more tasks.
- **t_agent_t_stage_out**: the amount of time taken to stage the output data of one or more tasks.

5.3 Arbitrary Durations

RA enables the **arbitrary** definition of durations, depending on the analysis requirements. For example, given an experiment to characterize the performance of one of RP's executors, it might be useful to measure the amount of time spent by each task in RP's Executor component.

> **Warning:** Correctly defining a duration requires a **detailed** understanding of both RP architecture and event model.

Once we acquired an understanding of RP architecture and event model, we can define our duration as the sum of the time spent by tasks in RP's Executor component, before and after the execution of the tasks' executable.

[6]:
```
t_executor_before = [{ru.STATE: rps.AGENT_EXECUTING},
                     {ru.EVENT: 'rank_start'} ]

t_executor_after  = [{ru.EVENT: 'rank_stop'},
                     {ru.EVENT: 'task_run_stop'} ]
```

5.4 Duration Analysis

Every analysis with RA requires to load the traces produced by RP or RADICAL-EnsembleToolkit (EnTK) into a session object. Both RP and EnTK write traces (i.e., timestamped and annotated sequences of events) to a directory called `client sandbox`. This directory is created inside the directory from which the application executed. The name of the client sandbox is a session ID, e.g., `rp.session.hostname.username.000000.0000` for RP and `en.session.hostname.username.000000.0000` for EnTK.

5.4.1 Session

Name and location of the session we profile.

[7]:
```
sidsbz2 = !find sessions -maxdepth 1 -type f -exec basename {} \;
sids = [s[:-8] for s in sidsbz2]
sdir = 'sessions/'
```

Unbzip and untar the session.

[8]:
```
sidbz2 = sidsbz2[0]
sid = sidbz2[:-8]
sp  = sdir + sidbz2

tar = tarfile.open(sp, mode='r:bz2')
tar.extractall(path=sdir)
tar.close()
```

Create a `ra.Session` object for the session. We do not need EnTK-specific traces so load only the RP traces contained in the EnTK session. Thus, we pass the `'radical.pilot'` session type to `ra.Session`.

Warning: We already know we need information about pilots and tasks. Thus, we save in memory two session objects filtered for pilots and tasks. This might be too expensive with large sessions, depending on the amount of memory available.

Note: We save the ouput of `ra.Session` in `capt` to avoid polluting the notebook with warning messages.

[9]:
```
%%capture capt

sp = sdir + sid
display(sp)

session = ra.Session(sp, 'radical.pilot')
pilots  = session.filter(etype='pilot', inplace=False)
tasks   = session.filter(etype='task' , inplace=False)
```

As seen above, each duration measures the time spent by an entity (local analyses) or a set of entities (global analyses) between two timestamps.

We starts with a global analysis to measure for how long all the pilots of a session have been active. Looking at RP's event model of the `pilot` entity and to `rp.utils.PILOT_DURATIONS_DEBUG`, we know that a pilot is active between the events `bootstrap_0_start` and `bootstrap_0_stop`. We also know that we have a default duration with those events: `p_agent_runtime`.

To measure that duration, first, we filter the session object so to keep only the entities of type Pilot; and, second, we get the **cumulative** amount of time for which all the pilot were active. It is that cumulative measure that defines this analysis as global.

[10]:
```
p_runtime = pilots.duration(event=rp.utils.PILOT_DURATIONS_DEBUG['p_agent_runtime'])
p_runtime
```

[10]:
```
3327.484554052353
```

Note: `ra.session.duration` works with a set of pilots, including the case in which we have a single pilot. If we have a single pilot, the cumulative active time of all the pilots is equal to the active time of the only available pilot.

If we have more than one pilot and we want to measure the active time of only one of them, then we need to perform a local analysis. A rapid way to get a list of all the pilot entities in the session and, for example, see their unique identifiers (uid) is:

[11]:
```
puids = [p.uid for p in pilots.get()]
puids
```

[11]:
```
['pilot.0000']
```

Once we know the ID of the pilot we want to analyze, first we filter the session object so to keep only the pilot we want to analyze; and, second, we get the amount of time for which that specific pilot was active:

[12]:
```
p0000 = pilots.filter(uid='pilot.0000')
p0000_runtime = p0000.duration(event=rp.utils.PILOT_DURATIONS_DEBUG['p_agent_runtime
↪'])
p0000_runtime
```

[12]:
```
3327.484554052353
```

The same approach and both global and local analyses can be used for every type of entity supported by RA (currently: pilot, task, pipeline, and stage).

Total task execution time (TTX) and RCT overheads (OVH) are among the most common metrics used to describe the global behavior of RCT. TTX measures the time taken by **ALL** the tasks to execute, accounting for their cocurrency. This means that if `Task_a` and `task_b` both start at the same time and `Task_a` terminates after 10 minutes and `Task_b` after 15, TTX will be 15 minutes. Conversely, if `task_b` starts to execute 5 minutes after `task_a`, TTX will be 20 minutes. Finally, if `task_b` starts to execute 10 minutes after `task_a` terminated, TTX will be 25 minutes as the gap between the two tasks will not be considered.

[13]:
```
ttx = tasks.duration(event=rp.utils.TASK_DURATIONS_DEBUG['t_agent_lm_execute'])
ovh = p_runtime - ttx

print('TTX: %f\nOVH: %f' % (ttx, ovh))
```

```
TTX: 3267.044225
OVH: 60.440329
```

5.4.2 Plotting

We plot TTX and OVH for 4 sessions of an experiment. We create suitable data structures to suppor the plotting and we produce a figure with 4 subplots. Unbzip and untar those sessions.

[14]:
```
for sid in sidsbz2:
    sp = sdir + sid
    tar = tarfile.open(sp, mode='r:bz2')
    tar.extractall(path=sdir)
    tar.close()
```

Create the session, tasks and pilots objects for each session.

[15]:
```
%%capture capt

ss = {}
for sid in sids:
    sp = sdir + sid
    ss[sid] = {'s': ra.Session(sp, 'radical.pilot')}
    ss[sid].update({'p': ss[sid]['s'].filter(etype='pilot'    , inplace=False),
                    't': ss[sid]['s'].filter(etype='task'     , inplace=False)})
```

Derive the information about each session we need to use in our plots.

[16]:
```
for sid in sids:
    ss[sid].update({'cores_node': ss[sid]['s'].get(etype='pilot')[0].cfg['resource_
↪details']['rm_info']['cores_per_node'],
                    'pid'       : ss[sid]['p'].list('uid'),
                    'ntask'     : len(ss[sid]['t'].get())
    })

    ss[sid].update({'ncores'    : ss[sid]['p'].get(uid=ss[sid]['pid'])[0].description[
↪'cores'],
                    'ngpus'     : ss[sid]['p'].get(uid=ss[sid]['pid'])[0].description[
↪'gpus']
    })

    ss[sid].update({'nnodes'    : int(ss[sid]['ncores']/ss[sid]['cores_node'])})
```

Use the default global durations to calculate TTX and OVH for each session of the experiment.

[17]:
```
for sid in sids:
    t  = ss[sid]['t']
    p  = ss[sid]['p']

    ss[sid].update({
      'ttx': t.duration(event=rp.utils.TASK_DURATIONS_DEBUG['t_agent_lm_execute']),
      'p_runtime': p.duration(event=rp.utils.PILOT_DURATIONS_DEBUG['p_agent_runtime'])
    })

    ss[sid].update({'ovh': ss[sid]['p_runtime'] - ss[sid]['ttx']})
```

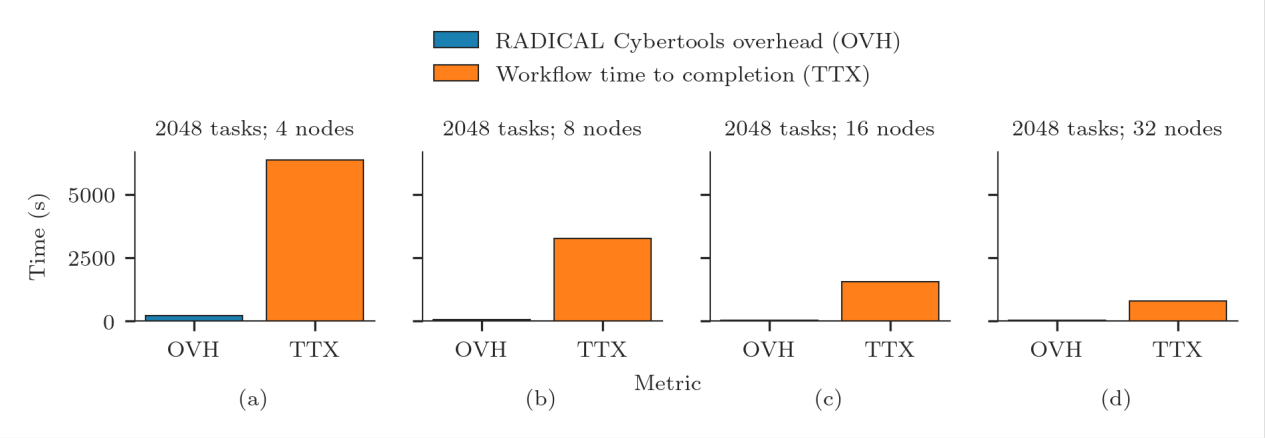
When plotting TTX and OVH in a plot with a subplot for each session, we want the subplots to be ordered by number of nodes. In that way, we will be able to 'see' the strong scaling behavior. Thus, we sort `sids` for number of cores.

[18]:
```
sorted_sids = [s[0] for s in sorted(ss.items(), key=lambda item: item[1]['ncores'])]
```

Plot TTX and OVH for each session, add information about each run and letters for each subplot fo easy referencing in a paper.

[19]:
```
nsids = len(sids)

fwidth, fhight = ra.get_plotsize(516,subplots=(1, nsids))
fig, axarr = plt.subplots(1, nsids, sharey=True, figsize=(fwidth, fhight))

i = 0
j = 'a'
for sid in sorted_sids:

    if len(sids) > 1:
        ax = axarr[i]
    else:
        ax = axarr

    ax.title.set_text('%s tasks; %s nodes' % (ss[sid]['ntask'], int(ss[sid]['nnodes
↪'])))

    ax.bar(x = 'OVH', height = ss[sid]['ovh'])
    ax.bar(x = 'TTX', height = ss[sid]['ttx'])

    ax.set_xlabel('(%s)' % j, labelpad=10)

    i = i + 1
    j = chr(ord(j) + 1)

fig.text(  0.05,  0.5 , 'Time (s)', va='center', rotation='vertical')
fig.text(  0.5 , -0.2, 'Metric'  , ha='center')
fig.legend(['RADICAL Cybertools overhead (OVH)',
            'Workflow time to completion (TTX)'],
           loc='upper center',
           bbox_to_anchor=(0.5, 1.5),
           ncol=1)
```

[19]:
```
<matplotlib.legend.Legend at 0x7ff99b0b4df0>
```



5.5 Danger of Duration Analysis

Warning: Most of the time, the durations of **global analyses** are **NOT** additive.

For example, the sum of the total time taken by RP Agent to manage all the tasks and the total amount of time taken to execute all those tasks is **greater** than the time taken to execute the workload. This is because RP is a distributed system that performs multiple operations at the same time on multiple resources. Thus, while RP Agent manages a task, it might be executing another task.

Consider three durations:

1. **t_agent_t_load**: the time from when RP Agent receives a compute task to the time in which the compute task's executable is launched.
2. **t_agent_t_execute**: default duration for the time taken by a compute task's executable to execute.
3. **t_agent_t_load**: the time from when a compute task's executable finishes to execute to when RP Agent mark the compute task with a final state (DONE, CANCELED or FAILED).

For a single task, `t_agent_t_load`, `t_agent_t_execute` and `t_agent_t_load` are contiguos and therefore additive. A single task cannot be loaded by RP Agent while it is also executed. For multiple tasks, this does not apply: one task might be loaded by RP Agent while another task is being executed.

5.6 Distribution of Durations

We want to calculate the statistical distribution of default and arbitrary durations. Variance and outliers characterize the runtime behavior of both tasks and RCT.

Global durations like TTX and OVH are aggregated across all entities: TTX aggregates the duration of each task while OVH that of all the RCT components active when no tasks are executed. For a distribution, we need instead the individual measure for each entity and component. For example, to calculate the distribution of task execution time, we have to measure the execution time of each task.

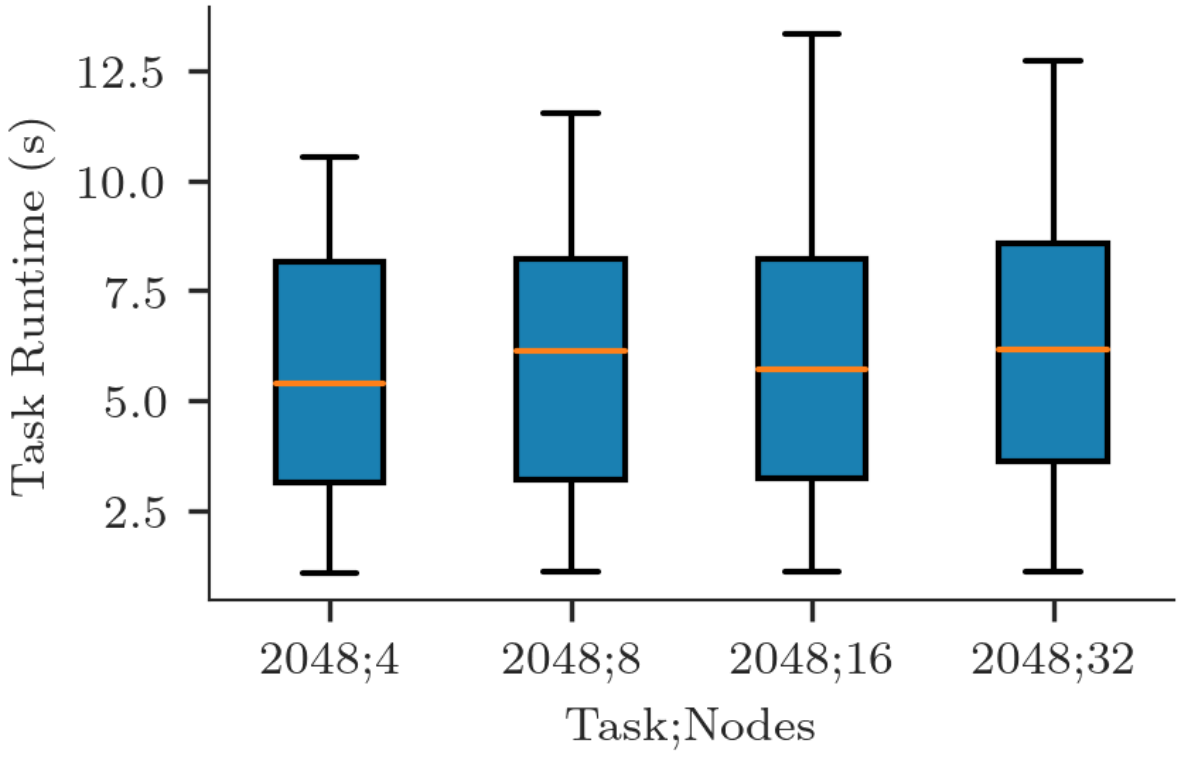
We use RA to cycle through all the task entities and then the `get` and `duration` methods to return the wanted duration for each task. We use both the default duration for task runtime and the two arbitary durations we defined above for the time taken by RP executor to manage the execution of the task.

[20]:
```
t_duration = {}
events = {'tx': rp.utils.TASK_DURATIONS_DEBUG['t_agent_lm_execute'],
          't_executor_before': t_executor_before,
          't_executor_after': t_executor_after}

for sid in sorted_sids:
    t_duration[sid] = {}
    for name, event in events.items():
        t_duration[sid].update({name: []})
        for tid in ss[sid]['t'].list('uid'):
            task = ss[sid]['t'].get(etype='task', uid=tid)[0]
            duration = task.duration(event=event)
            t_duration[sid][name].append(duration)
```

We can now plot the distribution of task execution time as a boxplot for each session:

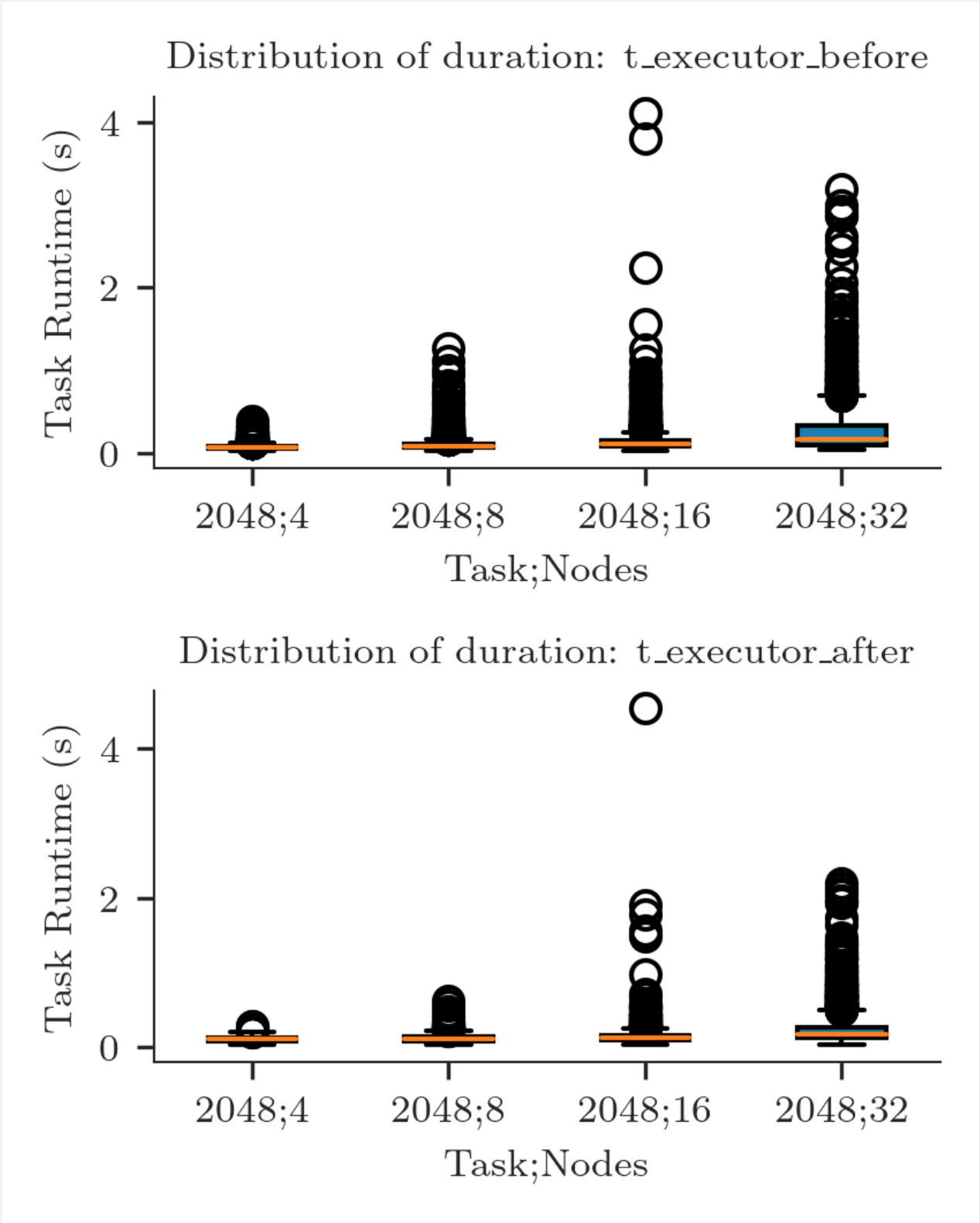
[21]:
```
fwidth, fhight = ra.get_plotsize(212)
fig, ax = plt.subplots(figsize=(fwidth, fhight))

data   = [t_duration[sid]['tx'] for sid in sorted_sids]
labels = ['%s;%s' % (ss[sid]['ntask'], int(ss[sid]['nnodes'])) for sid in sorted_sids]

ax.boxplot(data, labels=labels, patch_artist=True)

ax.set_xlabel('Task;Nodes')
ax.set_ylabel('Task Runtime (s)')
```

[21]:
```
Text(0, 0.5, 'Task Runtime (s)')
```



We can do the same for the arbitrary durations defined above: `t_executor_before` and `t_executor_after`

[22]:
```
fwidth, fhight = ra.get_plotsize(212, subplots=(2, 1))
fig, axarr = plt.subplots(2, 1, figsize=(fwidth, fhight))
plt.subplots_adjust(hspace=0.6)

i = 0
for dname in ['t_executor_before', 't_executor_after']:
    ax = axarr[i]

    data   = [t_duration[sid][dname] for sid in sorted_sids]
    labels = ['%s;%s' % (ss[sid]['ntask'], int(ss[sid]['nnodes'])) for sid in sorted_
↪sids]

    ax.boxplot(data, labels=labels, patch_artist=True)

    ax.set_title('Distribution of duration: %s' % ra.to_latex(dname))
    ax.set_xlabel('Task;Nodes')
    ax.set_ylabel('Task Runtime (s)')

    i += 1
```


Distribution of duration: t_executor_before
Task Runtime (s)
0
2
4
2048;4
2048;8
2048;16
2048;32
Task;Nodes
Distribution of duration: t_executor_after
Task Runtime (s)
0
2
4
2048;4
2048;8
2048;16
2048;32
Task;Nodes

CHAPTER 6

Resource Utilization

RADICAL-Analytics (RA) allows to calculate resource utilization for single and multiple RADICAL-Pilot (RP) sessions. Currently, RA supports CPU and GPU resources but in the future may support also RAM and I/O.

Resource utilization is expressed as the amount of time for which each task and pilot utilized available resources. For example, `task_000000` may have used 6 GPUs and 1 core for 15 minutes, and `pilot_0000` may have utilized (better, held) all the available resources for 1 hour.

RA can further characterize resource utilization by differentiating among the state in which each task and pilot were when utilizing (or holding) available resources. For example, `pilot_0000` may have held all the available resources for 5 minutes while bootstrapping or a variable amount of resources while scheduling each task. Similarly, tasks may held resources while being in a pre_execution or cmd_execution state.

Calculating resource utilization for all the entities and all their states is computationally expensive: given a 2020 laptop with 8 cores and 32GB of RAM, RA takes ~4 hours to plot the resource utilization of 100,000 heterogeneous tasks executed on a pilot with 200,000 CPUs and 24,000 GPUs. For sessions with 1M+ tasks, RA cannot be utilized to plot completed resource utilization in a reasonable amount of time.

Thus, RA offers two ways to compute resource utilization: fully detailed and aggregated. In the former, RA calculates the utilization for each core (e.g., core and GPU); in the latter, RA calculates the aggregated utilization of the resources over time, without mapping utilization over resource IDs. Aggregated utilization is less computationally intensive and it has been used to plot runs with 10M+ tasks.

6.1 Prologue

Load the Python modules needed to profile and plot a RP session.

[1]:
```
import os
import glob
import tarfile

import pandas as pd
import matplotlib as mpl
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker

import radical.utils as ru
import radical.pilot as rp
import radical.entk as re
import radical.analytics as ra
```

```
1681200946.455 : radical.analytics    : 828   : 140663965345600 : INFO     : radical.
↪analytics    version: 1.20.0-v1.20.0-26-g84002ce@docs-fix
```

Load the RADICAL Matplotlib style to obtain viasually consistent and publishable-qality plots.

[2]:
```
plt.style.use(ra.get_mplstyle('radical_mpl'))
```

Usually, it is useful to record the stack used for the analysis.

Note: The analysis stack might be different from the stack used to create the session to analyze. Usually, the two stacks must have the same minor release number (Major.Minor.Patch) in order to be compatible.

---

[3]:
```
! radical-stack
```

```
1681200947.153 : radical.analytics    : 862   : 140648076404544 : INFO     : radical.
↪analytics    version: 1.20.0-v1.20.0-26-g84002ce@docs-fix

  python               : /mnt/home/merzky/radical/radical.analytics.devel/ve3/bin/
↪python3
  pythonpath           :
  version              : 3.10.11
  virtualenv           : /mnt/home/merzky/radical/radical.analytics.devel/ve3

  radical.analytics    : 1.20.0-v1.20.0-26-g84002ce@docs-fix
  radical.entk         : 1.30.0
  radical.gtod         : 1.20.1
  radical.pilot        : 1.21.0
  radical.saga         : 1.22.0
  radical.utils        : 1.22.0
```

6.2 Detailed Resource Utilization

Given a RP session, RA helper functions take one resource type as input and return utilization, patches and legends for that type of resource. Plotting multiple types of resources requires creating separate plots. If needed, plots can be stacked, maintaining their time alignment. Here the default workflow to create a detailed utilization plot, with stacked plots for CPU and GPU resources.

6.2.1 Metrics

Define the metrics you want RA to use to calculate resource utilization of task(s) and pilot(s). A metric is used to measure the amount of time for which a set of resource was used by an entity in a specific state. The list of all available durations is in `rp.utils.PILOT_DURATIONS`; `rp.utils.TASK_DURATIONS_DEFAULT`;

`rp.utils.TASK_DURATIONS_APP`; `rp.utils.TASK_DURATIONS_PRTE`; and `rp.utils.ASK_DURATIONS_PRTE_APP`. Each metric has a label—the name of the metric—, a list of durations, and a color used when plotting that metric.

One can use an arbitrary number of metrics, depending on the information that the plot needs to convey. For example, using only 'Exec Cmd' will show the time for which each resource was utilized to execute a given task. The rest of the plot will be white, indicating that the resources where otherwise utilized or idling.

Barring exceptional cases, colors should not be changed when using RA for RADICAL publications.

[4]:
```
    metrics = [
        ['Bootstrap', ['boot', 'setup_1']                          , '#c6dbef'],
        ['Warmup'   , ['warm' ]                                    , '#f0f0f0'],
        ['Schedule' , ['exec_queue','exec_prep', 'unschedule']     , '#c994c7'],
        ['Exec RP'  , ['exec_rp', 'exec_sh', 'term_sh', 'term_rp'], '#fdbb84'],
        ['Exec Cmd' , ['exec_cmd']                                 , '#e31a1c'],
        ['Cooldown' , ['drain']                                    , '#addd8e']
    ]
```

6.2.2 Sessions

Name a location of all the sessions of the experiment.

[5]:
```
sessions = glob.glob('**/rp.session.*.tar.bz2')
sessions
```

[5]:
```
['sessions/rp.session.mosto.mturilli.019432.0005.tar.bz2',
 'sessions/rp.session.mosto.mturilli.019432.0003.tar.bz2',
 'sessions/rp.session.mosto.mturilli.019432.0002.tar.bz2',
 'sessions/rp.session.mosto.mturilli.019432.0004.tar.bz2']
```

Create a `ra.Session` object for the session. We do not need EnTK-specific traces so load only the RP traces contained in the EnTK session. Thus, we pass the `'radical.pilot'` session type to `ra.Session`.

> **Warning:** We already know we need information about pilots and tasks. Thus, we save in memory two session objects filtered for pilots and tasks. This might be too expensive with large sessions, depending on the amount of memory available.

Note: We save the ouput of `ra.Session` in `capt` to avoid polluting the notebook with warning messages.

[6]:
```
%%capture capt

ss = {}
sids = list()
for session in sessions:
    ra_session = ra.Session(session, 'radical.pilot')
    sid = ra_session.uid
    sids.append(sid)
    ss[sid] = {'s': ra_session}
    ss[sid].update({'p': ss[sid]['s'].filter(etype='pilot', inplace=False),
```

(continues on next page)

(continued from previous page)

```
                    't': ss[sid]['s'].filter(etype='task' , inplace=False)})
```

Derive the information about each session we need to use in our plots.

[7]:
```
for sid in sids:

    print(sid)
    ss[sid].update({'cores_node': ss[sid]['s'].get(etype='pilot')[0].cfg['resource_
↪details']['rm_info']['cores_per_node'],
                    'pid'       : ss[sid]['p'].list('uid'),
                    'ntask'     : len(ss[sid]['t'].get())
    })

    ss[sid].update({'ncores'    : ss[sid]['p'].get(uid=ss[sid]['pid'])[0].description[
↪'cores'],
                    'ngpus'     : ss[sid]['p'].get(uid=ss[sid]['pid'])[0].description[
↪'gpus']
    })

    ss[sid].update({'nnodes'    : int(ss[sid]['ncores']/ss[sid]['cores_node'])})
```

```
rp.session.mosto.mturilli.019432.0005
rp.session.mosto.mturilli.019432.0003
rp.session.mosto.mturilli.019432.0002
rp.session.mosto.mturilli.019432.0004
```

When plotting resource utilization with a subplot for each session, we want the subplots to be ordered by number of nodes. Thus, we sort sids for number of cores.

[8]:
```
sorted_sids = [s[0] for s in sorted(ss.items(), key=lambda item: item[1]['ncores'])]
```

6.2.3 Experiment

Construct a `ra.Experiment` object and calculate the starting point of each pilot in order to zero the X axis of the plot. Without that, the plot would start after the time spent by the pilot waiting in the queue. The experiment object exposes a method to calculate the consumption of each resource for each entity and metric.

[9]:
```
%%capture capt

exp = ra.Experiment(sessions, stype='radical.pilot')
```

Use `ra.Experiment.utilization()` to profile GPU resources utilization. Use the metrics defined above and all the sessions of the experiment `exp`.

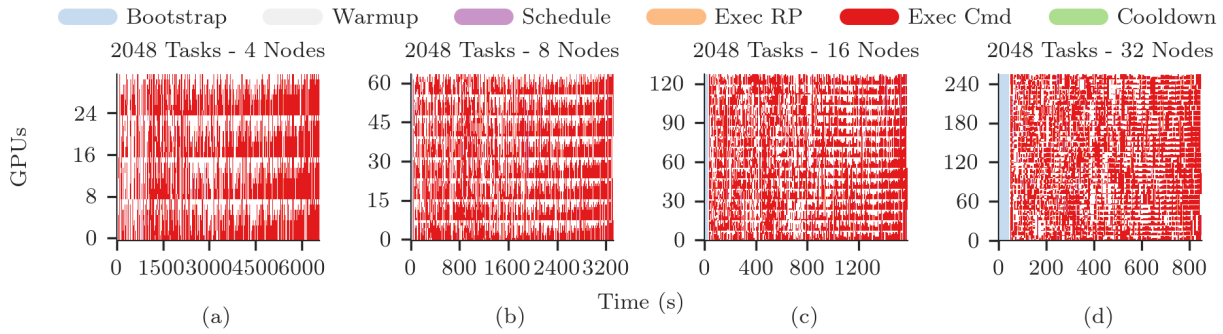
[10]:
```
# Type of resource we want to plot: cpu or gpu
rtypes=['cpu', 'gpu']

provided, consumed, stats_abs, stats_rel, info = exp.utilization(metrics=metrics,␣
↪rtype=rtypes[1])
```

6.2.4 Plotting GPU Utilization

We now have everything we need to plot the detailed GPU utilization of the experiment with Matplotlib.

[11]:
```
# sessions you want to plot
nsids = len(sorted_sids)

# Get the start time of each pilot
p_zeros = ra.get_pilots_zeros(exp)

# Create figure and 1 subplot for each session
# Use LaTeX document page size (see RA Plotting Chapter)
fwidth, fhight = ra.get_plotsize(516, subplots=(1, nsids))
fig, axarr = plt.subplots(1, nsids, sharex='col', figsize=(fwidth, fhight))

# Avoid overlapping between Y-axes ticks and sub-figures
plt.subplots_adjust(wspace=0.45)

# Generate the subplots with labels
i = 0
j = 'a'
legend = None
for sid in sorted_sids:

    # Use a single plot if we have a single session
    if nsids > 1:
        ax = axarr[i]
        ax.set_xlabel('(%s)' % j, labelpad=10)
    else:
        ax = axarr

    # Plot legend, patched, X and Y axes objects (here we know we have only 1 pilot)
    pid = ss[sid]['p'].list('uid')[0]
    legend, patches, x, y = ra.get_plot_utilization(metrics, consumed, p_
↪zeros[sid][pid], sid)

    # Place all the patches, one for each metric, on the axes
    for patch in patches:
        ax.add_patch(patch)

    # Title of the plot. Facultative, requires info about session (see RA Info␣
↪Chapter)
    # NOTE: you may have to change font size, depending on space available
    ax.set_title('%s Tasks - %s Nodes' % (ss[sid]['ntask'], int(ss[sid]['nnodes'])))

    # Format axes
    ax.set_xlim([x['min'], x['max']])
    ax.set_ylim([y['min'], y['max']])
    ax.yaxis.set_major_locator(mticker.MaxNLocator(5))
    ax.xaxis.set_major_locator(mticker.MaxNLocator(5))

    i = i+1
    j = chr(ord(j) + 1)

# Add legend
fig.legend(legend, [m[0] for m in metrics],
           loc='upper center', bbox_to_anchor=(0.5, 1.25), ncol=6)

# Add axes labels
fig.text( 0.05,  0.5, '%ss' % rtypes[1].upper(), va='center', rotation='vertical')
fig.text( 0.5 , -0.2, 'Time (s)', ha='center')
```

[11]:
```
Text(0.5, -0.2, 'Time (s)')
```



6.2.5 Plotting CPU/GPU Utilization

One plot for each type of resource, stacked for each session. For 4 sessions, we have 8 subplots, stackes in two raws, each with 4 columns.

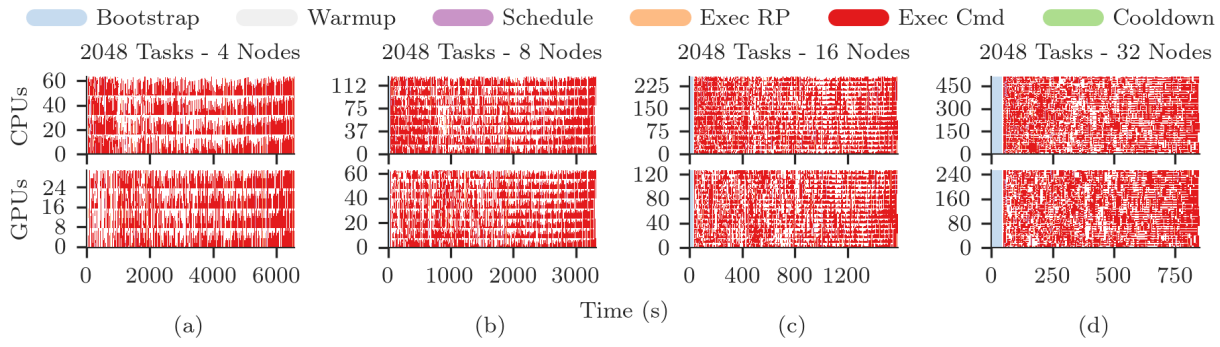
[12]:
```
# sessions you want to plot
nsids = len(sorted_sids)

# Create figure and 1 subplot for each session
# Use LaTeX document page size (see RA Plotting Chapter)
fwidth, fhight = ra.get_plotsize(516, subplots=(1, nsids))
fig, axarr = plt.subplots(2, nsids, sharex='col', figsize=(fwidth, fhight))

# Avoid overlapping between Y-axes ticks and sub-figures
plt.subplots_adjust(wspace=0.45)

# Generate the subplots with labels

legend = None
for k, rtype in enumerate(rtypes):

    _, consumed, _, _, _ = exp.utilization(metrics=metrics, rtype=rtype)

    j = 'a'
    for i, sid in enumerate(sorted_sids):

        # we know we have only 1 pilot
        pid = ss[sid]['p'].list('uid')[0]

        # Plot legend, patched, X and Y axes objects
        legend, patches, x, y = ra.get_plot_utilization(metrics, consumed,
                        p_zeros[sid][pid], sid)

        # Place all the patches, one for each metric, on the axes
        for patch in patches:
            axarr[k][i].add_patch(patch)

        # Title of the plot. Facultative, requires info about session (see RA
        # Info Chapter). We set the title only on the first raw of plots
        if rtype == 'cpu':
            axarr[k][i].set_title('%s Tasks - %s Nodes' % (ss[sid]['ntask'],
```

(continues on next page)

(continued from previous page)

```
                    int(ss[sid]['nnodes'])))

        # Format axes
        axarr[k][i].set_xlim([x['min'],     x['max']])
        axarr[k][i].set_ylim([y['min'], int(y['max'])])
        axarr[k][i].yaxis.set_major_locator(mticker.MaxNLocator(4))
        axarr[k][i].xaxis.set_major_locator(mticker.MaxNLocator(4))

        if rtype == 'cpu':
            # Specific to Summit when using SMT=4 (default)
            axarr[k][i].yaxis.set_major_formatter(
                    mticker.FuncFormatter(lambda z, pos: int(z/4)))

        # Y label per subplot. We keep only the 1st for each raw.
        if i == 0:
            axarr[k][i].set_ylabel('%ss' % rtype.upper())

        # Set x labels to letters for references in the paper.
        # Set them only for the bottom-most subplot
        if rtype == 'gpu':
            axarr[k][i].set_xlabel('(%s)' % j, labelpad=10)

        # update session id and raw identifier letter
        j = chr(ord(j) + 1)


# Add legend
fig.legend(legend, [m[0] for m in metrics],
           loc='upper center', bbox_to_anchor=(0.5, 1.25), ncol=6)

# Add axes labels
fig.text( 0.5 , -0.2, 'Time (s)', ha='center')
```

[12]:
```
Text(0.5, -0.2, 'Time (s)')
```



6.3 Aggregated Resource Utilization

This method is still under development and, as such, it requires to explicitly define the durations for each metric. Defaults will be included in `rp.utils` as done with the detailed plotting.

6.3.1 Metrics

The definition of metrics needs to be accompanied by the explicit definition of the event transitions represented by each metric. RP transitions are documented here but default values will be made available at a later time.

[13]:
```
# pick and choose what resources to plot (one sub-plot per resource)
resrc = ['cpu', 'gpu']

# pick and choose what contributions to plot
#            metric     ,  line color,  alpha, fill color,  alpha
metrics  = [['bootstrap', ['#c6dbef' ,  0.0  , '#c6dbef' ,  1  ]],
            ['exec_cmd' , ['#e31a1c' ,  0.0  , '#e31a1c' ,  1  ]],
            ['schedule' , ['#c994c7' ,  0.0  , '#c994c7' ,  1  ]],
            ['exec_rp'  , ['#fdbb84' ,  0.0  , '#fdbb84' ,  1  ]],
            ['term'     , ['#addd8e' ,  0.0  , '#addd8e' ,  1  ]],
            ['idle'     , ['#f0f0f0' ,  0.0  , '#f0f0f0' ,  1  ]] ]

# transition events for pilot, task, master, worker, request
#
# event  : resource transitions from : resource transitions to
#
p_trans = [[{1: 'bootstrap_0_start'}     , 'system'     , 'bootstrap'  ],
           [{5: 'PMGR_ACTIVE'}           , 'bootstrap'  , 'idle'       ],
           [{1: 'cmd', 6: 'cancel_pilot'}, 'idle'       , 'term'       ],
           [{1: 'bootstrap_0_stop'}      , 'term'       , 'system'     ],
           [{1: 'sub_agent_start'}       , 'idle'       , 'agent'      ],
           [{1: 'sub_agent_stop'}        , 'agent'      , 'term'       ] ]

t_trans = [[{1: 'schedule_ok'}           , 'idle'       , 'schedule'   ],
           [{1: 'exec_start'}            , 'schedule'   , 'exec_rp'    ],
           [{1: 'task_exec_start'}       , 'exec_rp'    , 'exec_cmd'   ],
           [{1: 'unschedule_stop'}       , 'exec_cmd'   , 'idle'       ] ]

m_trans = [[{1: 'schedule_ok'}           , 'idle'       , 'schedule'   ],
           [{1: 'exec_start'}            , 'schedule'   , 'exec_rp'    ],
           [{1: 'task_exec_start'}       , 'exec_rp'    , 'exec_master'],
           [{1: 'unschedule_stop'}       , 'exec_master', 'idle'       ] ]

w_trans = [[{1: 'schedule_ok'}           , 'idle'       , 'schedule'   ],
           [{1: 'exec_start'}            , 'schedule'   , 'exec_rp'    ],
           [{1: 'task_exec_start'}       , 'exec_rp'    , 'exec_worker'],
           [{1: 'unschedule_stop'}       , 'exec_worker', 'idle'       ] ]

r_trans = [[{1: 'req_start'}             , 'exec_worker', 'workload'   ],
           [{1: 'req_stop'}              , 'workload'   , 'exec_worker'] ]

# what entity maps to what transition table
tmap = {'pilot'  : p_trans,
        'task'   : t_trans,
        'master' : m_trans,
        'worker' : w_trans,
        'request': r_trans}
```

6.3.2 Session

Pick a session to plot and use the `ra.Session` object already stored in memory. Also use the `ra.Entity` object for the pilot of that session. Here we assume we have a session with a single pilot.

[15]:
```
uid = sids[0]
session = ss[uid]['s']
pilot   = ss[uid]['p'].get()[0]
```

6.3.3 Plotting CPU/GPU Utilization

Stack two plots for the chosen session, one for CPU and one for GPU resources.

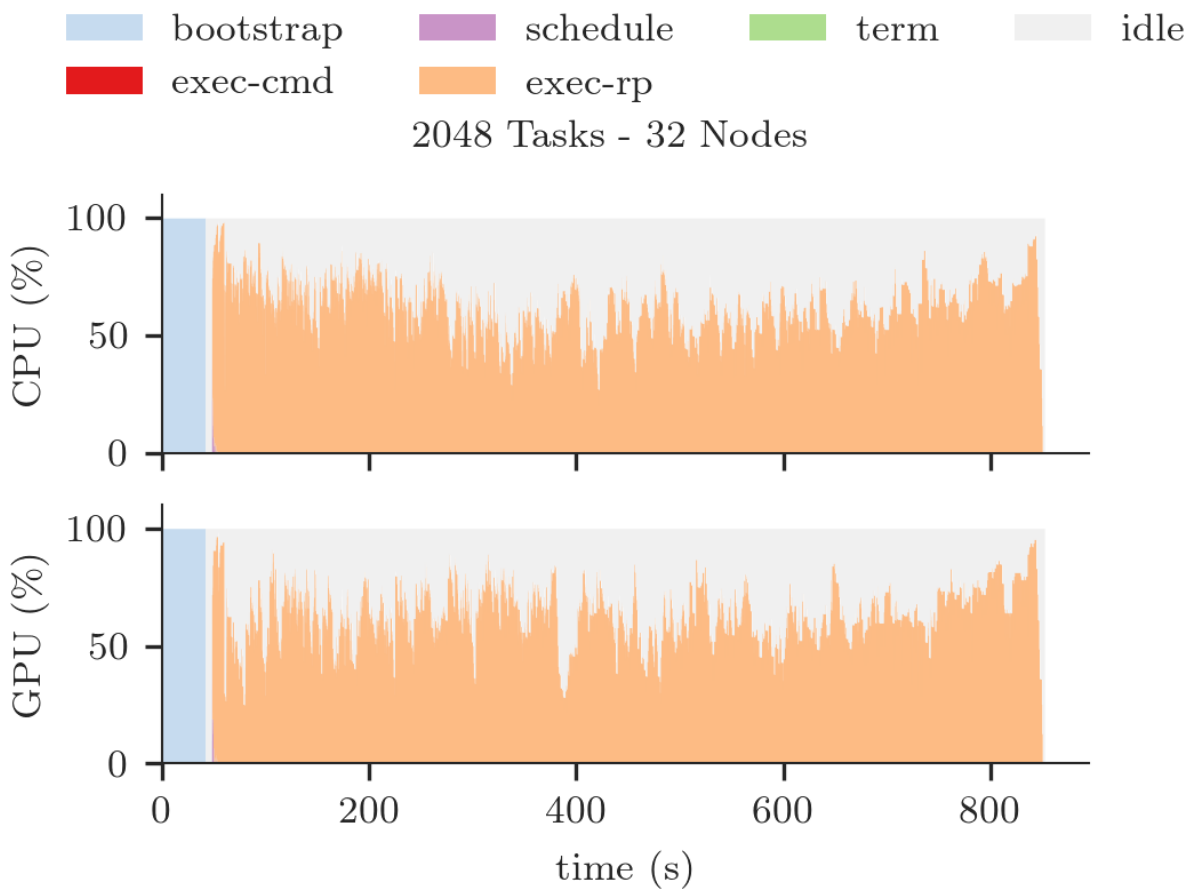
[16]:
```
# metrics to stack and to plot
to_stack = [m[0] for m in metrics]
to_plot  = {m[0]: m[1] for m in metrics}

# Use to set Y-axes to % of resource utilization
use_percent = True

# Derive pilot and task timeseries of a session for each metric
p_resrc, series, x = ra.get_pilot_series(session, pilot, tmap, resrc, use_percent)

# #plots = # of resource types (e.g., CPU/GPU = 2 resource types = 2 plots)
n_plots = 0
for r in p_resrc:
    if p_resrc[r]:
        n_plots += 1

# sub-plots for each resource type, legend on first, x-axis shared
fig = plt.figure(figsize=(ra.get_plotsize(252)))
gs  = mpl.gridspec.GridSpec(n_plots, 1)

for plot_id, r in enumerate(resrc):

    if not p_resrc[r]:
        continue

    # create sub-plot
    ax = plt.subplot(gs[plot_id])

    # stack timeseries for each metrics into areas
    areas = ra.stack_transitions(series, r, to_stack)

    # plot individual metrics
    prev_m  = None
    lines   = list()
    patches = list()
    legend  = list()
    for num, m in enumerate(areas.keys()):

        if m not in to_plot:
            if m != 'time':
                print('skip', m)
            continue

        lcol   = to_plot[m][0]
        lalpha = to_plot[m][1]
        pcol   = to_plot[m][2]
        palpha = to_plot[m][3]
```

(continues on next page)

(continued from previous page)

```
        # plot the (stacked) areas
        line, = ax.step(areas['time'], areas[m], where='post', label=m,
                        color=lcol, alpha=lalpha, linewidth=1.0)

        # fill first metric toward 0, all others towards previous line
        if not prev_m:
            patch = ax.fill_between(areas['time'], areas[m],
                                    step='post', label=m, linewidth=0.0,
                                    color=pcol, alpha=palpha)

        else:
            patch = ax.fill_between(areas['time'], areas[m], areas[prev_m],
                                    step='post', label=m, linewidth=0.0,
                                    color=pcol, alpha=palpha)

        # remember lines and patches for legend
        legend.append(m.replace('_', '-'))
        patches.append(patch)

        # remember this line to fill against
        prev_m = m

    ax.set_xlim([x['min'], x['max']])
    if use_percent:
        ax.set_ylim([0, 110])
    else:
        ax.set_ylim([0, p_resrc[r]])

    ax.set_xlabel('time (s)')
    ax.set_ylabel('%s (%s)' % (r.upper(), '\%'))


    # first sub-plot gets legend
    if plot_id == 0:
        ax.legend(patches, legend, loc='upper center', ncol=4,
                  bbox_to_anchor=(0.5, 1.8), fancybox=True, shadow=True)

for ax in fig.get_axes():
    ax.label_outer()

# Title of the plot
fig.suptitle('%s Tasks - %s Nodes' % (ss[uid]['ntask'], ss[uid]['nnodes']))
```

[16]:
```
Text(0.5, 0.98, '2048 Tasks - 32 Nodes')
```


bootstrap
schedule
term
idle
exec-cmd
exec-rp
2048 Tasks - 32 Nodes
CPU (%)
100
50
0
GPU (%)
100
50
0
0
200
400
600
800
time (s)

[ ]:

CHAPTER 7

Timestamps

RADICAL-Analytics (RA) enables event-based analyses in which the timestamps recorded in a RADICAL-Cybertools (RCT) session are studied as timeseries instead of durations. Those analyses are low-level and, most of the time, useful to 'visualize' the process of execution as it happens in one or more components of the stack.

Warning: Sessions with 100,000+ tasks and resoruces may generate traces with 1M+ events. Depending on the quantity of available memory, plotting that amount of timestamps with RA could not be feasable.

7.1 Prologue

Load the Python modules needed to profile and plot a RCT session.

[1]:
```
import tarfile

import numpy as np
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker

import radical.utils as ru
import radical.pilot as rp
import radical.analytics as ra

from radical.pilot import states as rps
```

Load the RADICAL Matplotlib style to obtain viasually consistent and publishable-qality plots.

[2]:
```
plt.style.use(ra.get_mplstyle('radical_mpl'))
```

Usually, it is useful to record the stack used for the analysis.

Note: The analysis stack might be different from the stack used to create the session to analyze. Usually, the two stacks must have the same minor release number (Major.Minor.Patch) in order to be compatible.

[3]:
```
! radical-stack
```

```
  python               : /home/docs/checkouts/readthedocs.org/user_builds/
↪radicalanalytics/envs/stable/bin/python3
  pythonpath           :
  version              : 3.9.15
  virtualenv           :

  radical.analytics    : 1.34.0-v1.34.0@HEAD-detached-at-0b58be0
  radical.entk         : 1.33.0
  radical.gtod         : 1.20.1
  radical.pilot        : 1.34.0
  radical.saga         : 1.34.0
  radical.utils        : 1.33.0
```

7.2 Event Model

RCT components have each a well-defined event model:

- RADICAL-Pilot (RP) event model
- RADICAL-EnsembleToolkit (EnTK) event model

Note: RA does not support RADICAL-SAGA.

Each event belongs to an entity and is timestamped within a component. The succession of the same event over time constitutes a time series. For example, in RP the event `schedule_ok` belongs to a `task` and is timestamped by `AgentSchedulingComponent`. The timeseries of that event indicates the rate at which tasks are scheduled by RP.

7.3 Timestamps analysis

We use RA to derive the timeseries for one or more events of interest. We then plot each time series singularly or together in the same plot. When plotting the time series of multiple events together, they must all be ordered in the same way. Typically, we sort the entities by the timestamp of their first event.

Here is the RA workflow for a timestamps analysis:

1. Go at RADICAL-Pilot (RP) event model, RP state model or RADICAL-EnsembleToolkit (EnTK) event model and derive the list of events of interest.
2. Convert events and states in RP/RA dict notation.

E.g., a scheduling event and state in RP:

- AGENT_SCHEDULING - picked up by agent scheduler, attempts to assign cores for execution
- AGENT_EXECUTING - picked up by the agent executor and ready to be launched

[4]:
```
state_sched = {ru.STATE: rps.AGENT_SCHEDULING}
state_exec = {ru.STATE: rps.AGENT_EXECUTING}
```

3. Filter a RCT session for the entity to which the selected event/state belong.
4. use `ra.entity.timestamps()` and the defined event/state to derive the time series for that event/state.

7.3.1 Session

Name and location of the session we profile.

[5]:
```
sidsbz2 = !find sessions -maxdepth 1 -type f -exec basename {} \;
sids = [s[:-8] for s in sidsbz2]
sdir = 'sessions/'
```

Unbzip and untar the session.

[6]:
```
sidbz2 = sidsbz2[0]
sid = sidbz2[:-8]
sp  = sdir + sidbz2

tar = tarfile.open(sp, mode='r:bz2')
tar.extractall(path=sdir)
tar.close()
```

Create a `ra.Session` object for the session. We do not need EnTK-specific traces so load only the RP traces contained in the EnTK session. Thus, we pass the `'radical.pilot'` session type to `ra.Session`.

Warning: We already know we need information about pilots and tasks. Thus, we save in memory two session objects filtered for pilots and tasks. This might be too expensive with large sessions, depending on the amount of memory available.

Note: We save the ouput of `ra.Session` in `capt` to avoid polluting the notebook with warning messages.

[7]:
```
%%capture capt

sp = sdir + sid

session = ra.Session(sp, 'radical.pilot')
pilots  = session.filter(etype='pilot', inplace=False)
tasks   = session.filter(etype='task' , inplace=False)
```

We usually want to collect some information about the sessions we are going to analyze. That information is used for bookeeping while performing the analysis (especially when having multiple sessions) and to add meaningful titles to (sub)plots.

[8]:
```
sinfo = {}

sinfo.update({
    'cores_node': session.get(etype='pilot')[0].cfg['resource_details']['rm_info'][
↪'cores_per_node'],
    'pid'       : pilots.list('uid'),
```

(continues on next page)

(continued from previous page)

```
    'ntask'     : len(tasks.get())
})

sinfo.update({
    'ncores'    : session.get(uid=sinfo['pid'])[0].description['cores'],
    'ngpus'     : pilots.get(uid=sinfo['pid'])[0].description['gpus']
})

sinfo.update({
    'nnodes'    : int(sinfo['ncores']/sinfo['cores_node'])
})
```

Use `ra.session.get()` on the filtered session objects that contains only task entities. Then use `ra.entity.timestamps()` to derive the time series for each event/state of interest. We put the time series into a pandas DataFrame to make plotting easier.

[9]:
```
tseries = {'AGENT_SCHEDULING': [],
           'AGENT_EXECUTING': []}

for task in tasks.get():
    ts_sched = task.timestamps(event=state_sched)[0]
    ts_exec = task.timestamps(event=state_exec)[0]
    tseries['AGENT_SCHEDULING'].append(ts_sched)
    tseries['AGENT_EXECUTING'].append(ts_exec)

time_series = pd.DataFrame.from_dict(tseries)
time_series
```

[9]:

	AGENT_SCHEDULING	AGENT_EXECUTING
0	57.458064	1132.290056
1	57.458064	696.452034
2	57.458064	779.445003
3	57.458064	865.469337
4	57.458064	732.306301
...	...	...
2043	57.722614	3084.190474
2044	57.722614	377.477711
2045	57.722614	732.306301
2046	57.722614	1137.071288
2047	57.722614	2164.869668

[2048 rows x 2 columns]

Usually, time series are plotted as lineplots but, in our case, we want to plot just the time stamps without a 'line' connecting those events, a potentially misleading artefact. Thus, we use a scatterplot in which the X axes are the number of tasks and the Y axes time in seconds. This somehow 'stretches' the meaning of a scatterplot as we do not use it to represent a correlation.

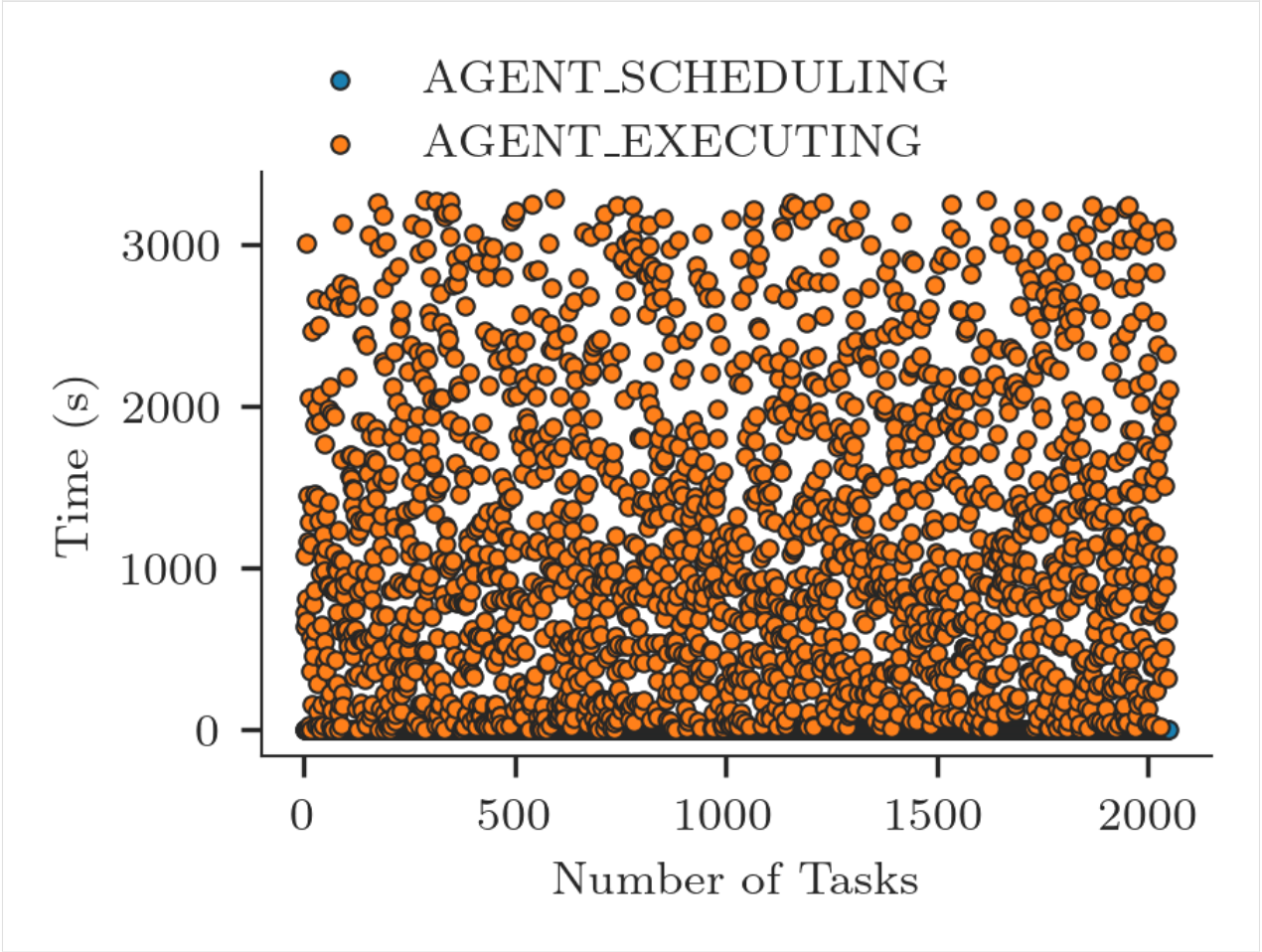
Note: We need to zero the Y axes as the timestamps are taken starting from the first timestamp of the session. The event/state we choose are much later down the execution. Here we select the event/state that has to appen first, based on our knowledge of RP's architecture. Alternatively, we could find the min among all the time stamps we have in the dataframe and use that as the zero point.

Note: Once we have found the zero point in time (`zero`) we subtract it to the whole time series. Pandas' dataframe

make that easy. We also add 1 to the index we use for the X axes so to start to count tasks from 1 instead of 0.

[10]:
```
fig, ax = plt.subplots(figsize=(ra.get_plotsize(212)))

# Find the min timestamp of the first event/state timeseries and use it to zero
# the Y axes.
zero = time_series['AGENT_SCHEDULING'].min()

ax.scatter(time_series['AGENT_SCHEDULING'].index + 1,
           time_series['AGENT_SCHEDULING'] - zero,
           marker = '.',
           label = ra.to_latex('AGENT_SCHEDULING'))
ax.scatter(time_series['AGENT_EXECUTING'].index + 1,
           time_series['AGENT_EXECUTING'] - zero,
           marker = '.',
           label = ra.to_latex('AGENT_EXECUTING'))

ax.legend(ncol=1, loc='upper left', bbox_to_anchor=(0,1.25))
ax.set_xlabel('Number of Tasks')
ax.set_ylabel('Time (s)')
```

[10]:
```
Text(0, 0.5, 'Time (s)')
```



The plot above shows that all the tasks arrive at RP's Scheduler together (AGENT_SCHEDULING state). That is expected as tasks are transferred in bulk from RP Client's Task Manager to RP Agent's Staging In component.

The plot shows that tasks are continously scheduled across the whole duration of the execution (schedule_ok event). That is expected as we have more tasks than available resurces and task wait in the scheduler queue to be scheduled until enough resource are available. Every time one of the task terminates, a certain amount of resources become available. When enough resources become available to execute a new task, the scheduler schedule the task that, then executes on those resources.

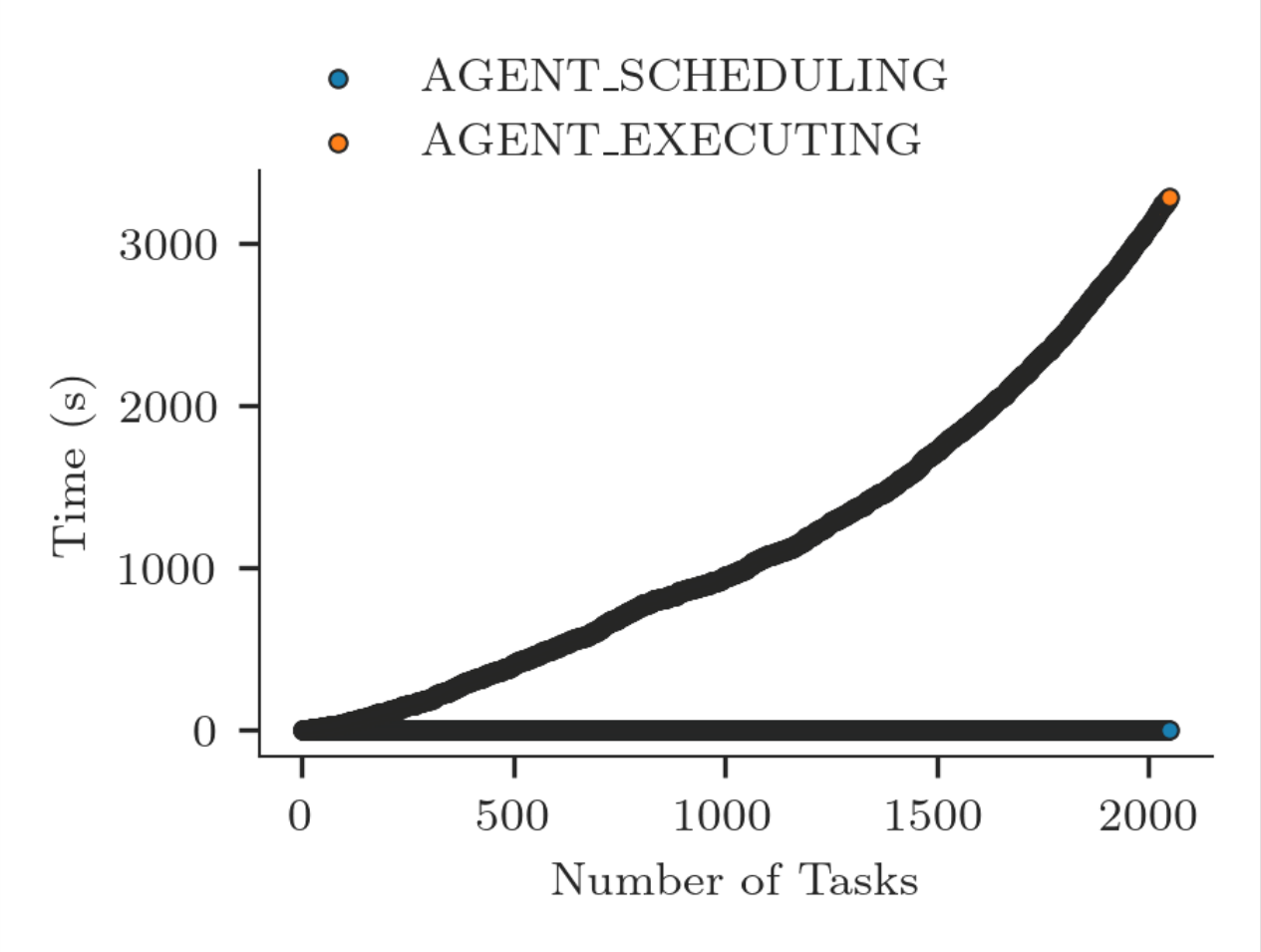
The plot above might be confusing because tasks are not ordered by the time in which they were scheduled. We sort `time_series` by `AGENT_EXECUTING` and then we plot the scatterplot again,

[11]:
```
ts_sorted = time_series.sort_values(by=['AGENT_EXECUTING']).reset_index(drop=True)
ts_sorted
```

[11]:
```
      AGENT_SCHEDULING  AGENT_EXECUTING
0             57.458064        57.669098
1             57.458064        57.676626
2             57.458064        57.684323
3             57.458064        57.691968
4             57.458064        57.717011
...                 ...              ...
2043          57.458064      3325.463749
2044          57.458064      3325.463749
2045          57.458064      3333.839660
2046          57.722614      3333.839660
2047          57.458064      3338.267327

[2048 rows x 2 columns]
```

[12]:
```
fig, ax = plt.subplots(figsize=(ra.get_plotsize(212)))

# Find the min timestamp of the first event/state timeseries and use it to zero
# the Y axes.
zero = ts_sorted['AGENT_SCHEDULING'].min()

ax.scatter(ts_sorted['AGENT_SCHEDULING'].index + 1,
           ts_sorted['AGENT_SCHEDULING'] - zero,
           marker = '.',
           label = ra.to_latex('AGENT_SCHEDULING'))
ax.scatter(ts_sorted['AGENT_EXECUTING'].index + 1,
           ts_sorted['AGENT_EXECUTING'] - zero,
           marker = '.',
           label = ra.to_latex('AGENT_EXECUTING'))

ax.legend(ncol=1, loc='upper left', bbox_to_anchor=(0,1.25))
ax.set_xlabel('Number of Tasks')
ax.set_ylabel('Time (s)')
```

[12]:
```
Text(0, 0.5, 'Time (s)')
```



Unsurprisingly, the resulting plot is consistent with the plot shown in *Concurrency*.

Adding execution events to our timestamps analysis should confirm the *duration distributions* of the time taken by RP's Executor launch method to launch tasks. We add the relevant events/states to the `time_series` dataframe and we sort it again for the `AGENT_EXECUTING` event.

[13]:
```
executor = {
    'rank_start'     : {ru.EVENT: 'rank_start'},
    'rank_stop'      : {ru.EVENT: 'rank_stop'}
}

for name, event in executor.items():

    tseries = []
    for task in tasks.get():
        ts_state = task.timestamps(event=event)[0]
        tseries.append(ts_state)

    time_series[name] = tseries

ts_sorted = time_series.sort_values(by=['AGENT_EXECUTING']).reset_index(drop=True)
ts_sorted
```

[13]:

	AGENT_SCHEDULING	AGENT_EXECUTING	rank_start	rank_stop
0	57.458064	57.669098	57.848533	68.974868

(continues on next page)

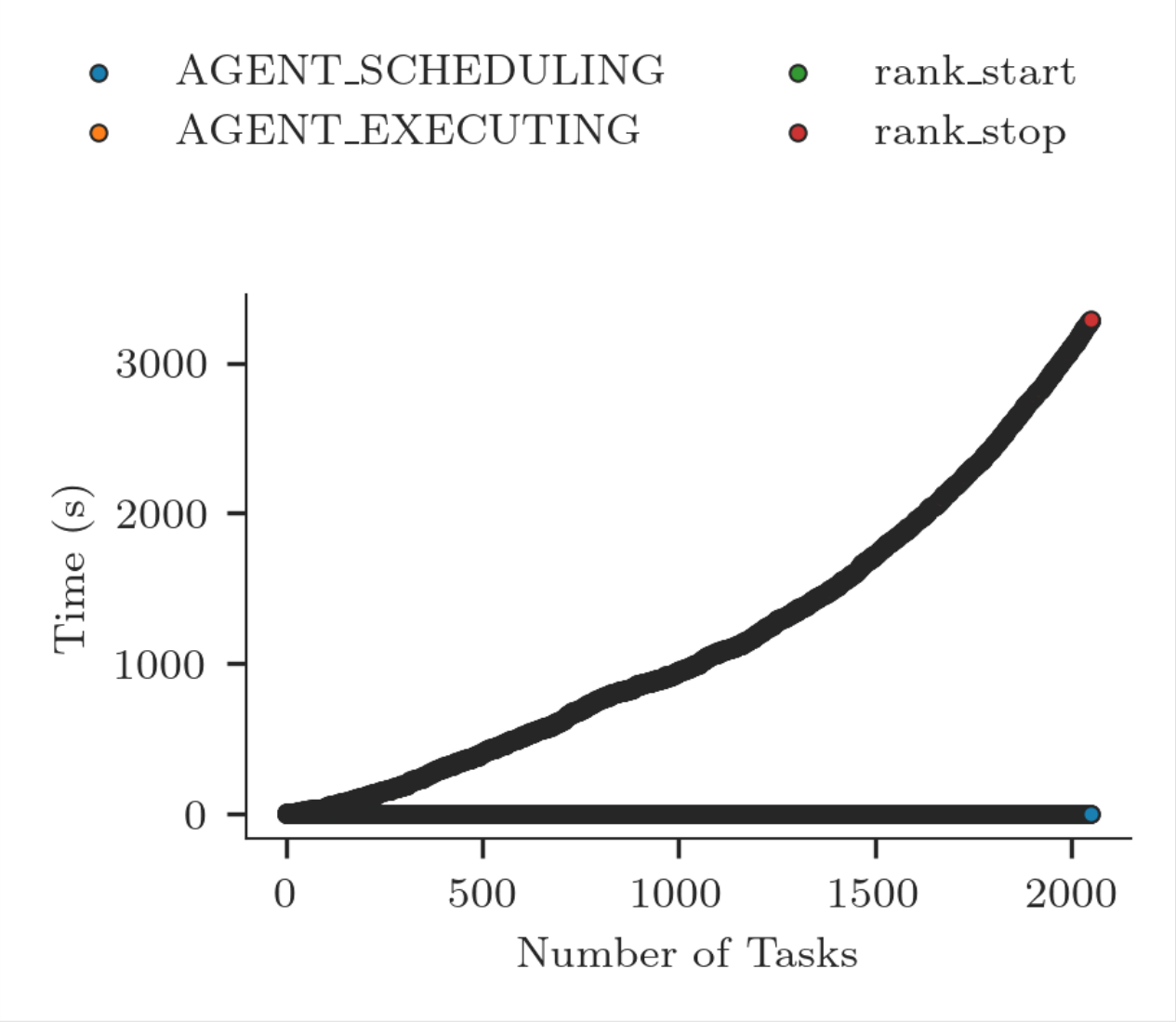
(continued from previous page)

```
1            57.458064        57.676626    57.884947    64.122034
2            57.458064        57.684323    57.883325    63.967687
3            57.458064        57.691968    57.989414    63.107628
4            57.458064        57.717011    58.008192    66.338753
...                ...              ...          ...          ...
2043         57.458064      3325.463749  3325.541283  3333.749098
2044         57.458064      3325.463749  3325.562549  3327.760062
2045         57.458064      3333.839660  3333.918936  3335.097386
2046         57.722614      3333.839660  3333.904375  3338.089182
2047         57.458064      3338.267327  3338.346752  3343.485991

[2048 rows x 4 columns]
```

We plot the new time series alongside the previous ones.

[14]:
```
fig, ax = plt.subplots(figsize=(ra.get_plotsize(212)))

zero = ts_sorted['AGENT_SCHEDULING'].min()

for ts in ts_sorted.columns:
    ax.scatter(ts_sorted[ts].index + 1,
               ts_sorted[ts] - zero,
               marker = '.',
               label = ra.to_latex(ts))

ax.legend(ncol=2, loc='upper left', bbox_to_anchor=(-0.25,1.5))
ax.set_xlabel('Number of Tasks')
ax.set_ylabel('Time (s)')
```

[14]:
```
Text(0, 0.5, 'Time (s)')
```



At the resolution of this plot, all the states and events `AGENT_SCHEDULING`, `AGENT_EXECUTING`, `rank_start` and `rank_stop` overlap. That indicates that the duration of each task is very short and, thus, the scheduling turnover is very rapid.

In presence of a large amount of tasks, we can slice the time stamps to plot one or more of their subsets.

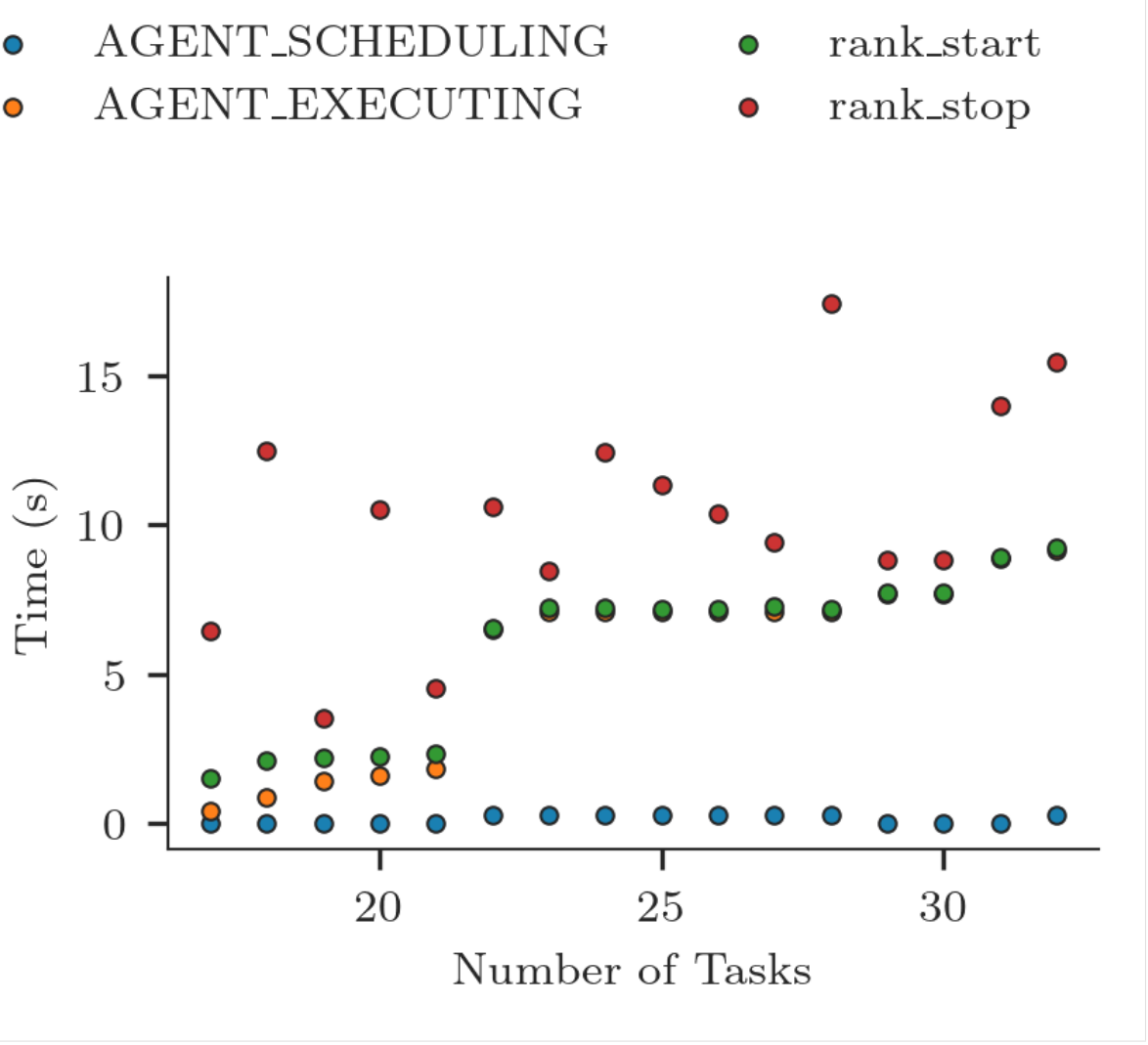
[15]:
```
fig, ax = plt.subplots(figsize=(ra.get_plotsize(212)))

# Slice time series to plot only one of their subsets
ts_sorted = ts_sorted.reset_index(drop=True)
ts_sorted = ts_sorted.iloc[16:32]

zero = ts_sorted['AGENT_SCHEDULING'].min()

for ts in ts_sorted.columns:
    ax.scatter(ts_sorted[ts].index + 1,
               ts_sorted[ts] - zero,
               marker = '.',
               label = ra.to_latex(ts))
```

(continues on next page)

(continued from previous page)

```
ax.legend(ncol=2, loc='upper left', bbox_to_anchor=(-0.25,1.5))
ax.set_xlabel('Number of Tasks')
ax.set_ylabel('Time (s)')
```

[15]: `Text(0, 0.5, 'Time (s)')`



CHAPTER 8

Concurrency

RADICAL-Analytics (RA) offers a method `ra.session.concurrency` that returns a time series, counting the number of tasks which are matching a given pair of timestamps at any point in time. For example, a time series can show the number of concurrent tasks that were scheduled, executed or staging in/out at every point of time, during the execution of the workload.

We plot concurrency time series as a canonical line plot. We can add to the same plot multiple timeseries, showing the relation among diverse components of each RADICAL-Cybertool (RCT) system.

8.1 Prologue

Load the Python modules needed to profile and plot a RCT session.

[1]:
```
import os
import tarfile

import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker

import radical.utils as ru
import radical.pilot as rp
import radical.entk as re
import radical.analytics as ra
```

Load the RADICAL Matplotlib style to obtain viasually consistent and publishable-qality plots.

[2]:
```
plt.style.use(ra.get_mplstyle('radical_mpl'))
```

Usually, it is useful to record the stack used for the analysis.

Note: The analysis stack might be different from the stack used to create the session to analyze. Usually, the two stacks must have the same minor release number (Major.Minor.Patch) in order to be compatible.

[3]:
```
! radical-stack
```

```
  python               : /home/docs/checkouts/readthedocs.org/user_builds/
↪radicalanalytics/envs/stable/bin/python3
  pythonpath           :
  version              : 3.9.15
  virtualenv           :

  radical.analytics    : 1.34.0-v1.34.0@HEAD-detached-at-0b58be0
  radical.entk         : 1.33.0
  radical.gtod         : 1.20.1
  radical.pilot        : 1.34.0
  radical.saga         : 1.34.0
  radical.utils        : 1.33.0
```

8.2 Session

Name and location of the session we profile.

[4]:
```
sidsbz2 = !find sessions -maxdepth 1 -type f -exec basename {} \;
sids = [s[:-8] for s in sidsbz2]
sdir = 'sessions/'
```

Unbzip and untar the session.

[5]:
```
sidbz2 = sidsbz2[0]
sid = sidbz2[:-8]
sp  = sdir + sidbz2

tar = tarfile.open(sp, mode='r:bz2')
tar.extractall(path=sdir)
tar.close()
```

Create a `ra.Session` object for the session. We do not need EnTK-specific traces so load only the RP traces contained in the EnTK session. Thus, we pass the `'radical.pilot'` session type to `ra.Session`.

Warning: We already know we need information about pilots and tasks. Thus, we save in memory two session objects filtered for pilots and tasks. This might be too expensive with large sessions, depending on the amount of memory available.

Note: We save the ouput of `ra.Session` in `capt` to avoid polluting the notebook with warning messages.

[6]:
```
%%capture capt
```

(continues on next page)

(continued from previous page)

```
sp = sdir + sid

session = ra.Session(sp, 'radical.pilot')
pilots  = session.filter(etype='pilot', inplace=False)
tasks   = session.filter(etype='task' , inplace=False)
```

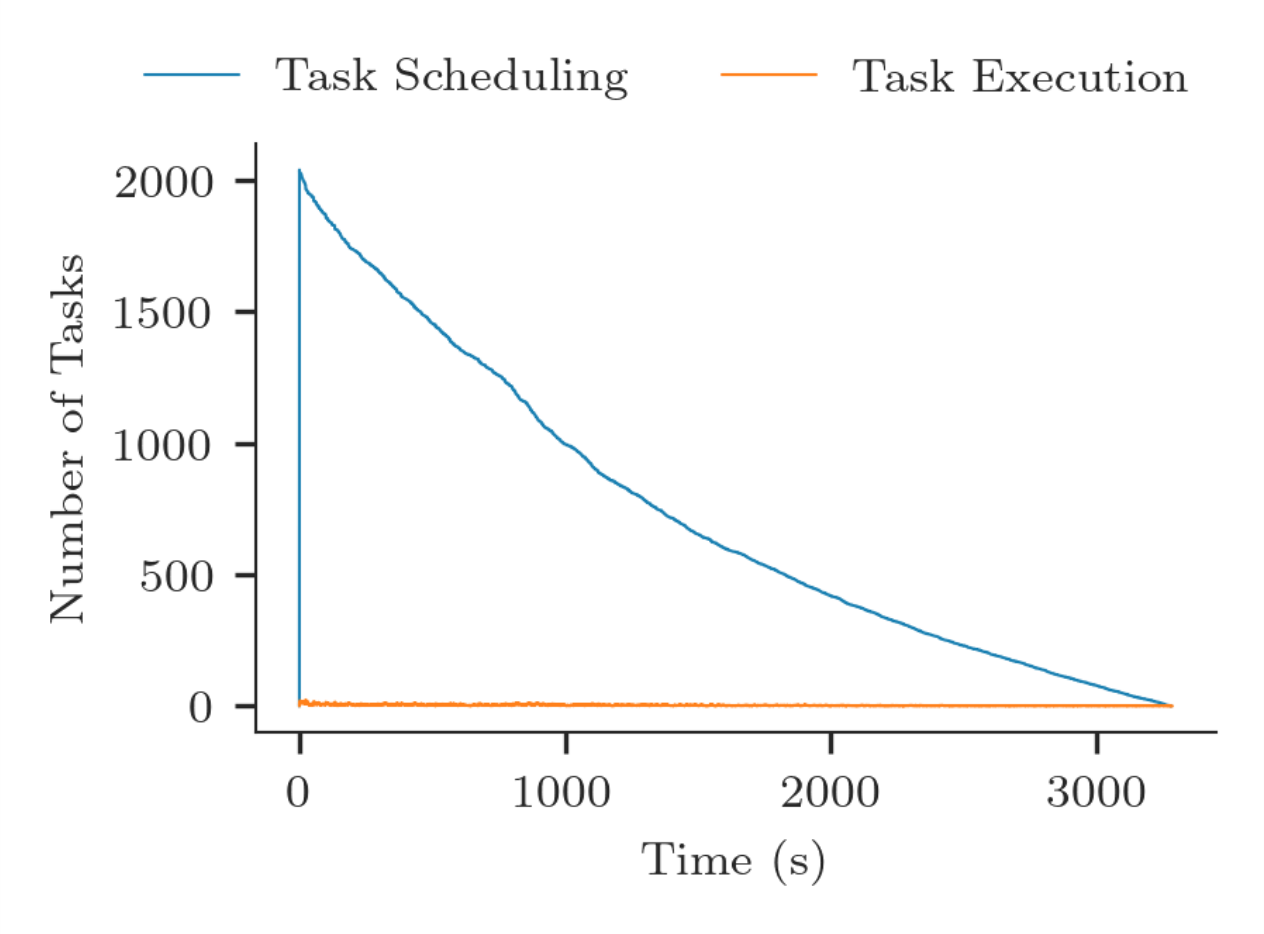
8.3 Plotting

We name some pairs of events we want to use for concurrency analysis. We use the ra.session's concurrency method to compute the number of tasks which match the given pair of timestamps at every point in time. We zero the time of the X axes.

[7]:
```
pairs = {'Task Scheduling'  : [{ru.STATE: 'AGENT_SCHEDULING'},
                               {ru.EVENT: 'schedule_ok'     } ],
         'Task Execution'   : [{ru.EVENT: 'rank_start'      },
                               {ru.EVENT: 'rank_stop'       } ]}

time_series = {pair: session.concurrency(event=pairs[pair]) for pair in pairs}
```

[8]:
```
fig, ax = plt.subplots(figsize=(ra.get_plotsize(212)))

for name in time_series:

    zero = min([e[0] for e in time_series[name]])
    x = [e[0]-zero for e in time_series[name]]

    y = [e[1] for e in time_series[name]]
    ax.plot(x, y, label=ra.to_latex(name))

ax.legend(ncol=2, loc='upper left', bbox_to_anchor=(-0.15,1.2))
ax.set_ylabel('Number of Tasks')
ax.set_xlabel('Time (s)')
```

[8]:
```
Text(0.5, 0, 'Time (s)')
```



The plot above shows that tasks are between 'AGENT_SCHEDULING' and 'schedule_ok' at the beginning of the execution (dark blue). Few seconds later, tasks start to be between 'rank_start' and 'rank_stop', i.e., they are scheduled and start executing. Tasks appear to have a relatively heterogeneous duration, consistent with the task runtime distribution measured in *duration analysis*.

Task as scheduled as soon as new resources become available, across the whole duration of the workload execution. Consistently, the total number of tasks waiting to be scheduled progressively decreases, represented by the slope of the blue line. Consistently, the number of executed tasks remain relatively constant across all the workload duration, represented by the orange line.

CHAPTER 9

API Reference

9.1 Session

class radical.analytics.**Session**(*src*, *stype*, *sid=None*, *_entities=None*, *_init=True*)

__init__(*src*, *stype*, *sid=None*, *_entities=None*, *_init=True*)

Create a radical.analytics session for analysis.

The session is created from a set of traces, which usually have been produced by a Session object in the RCT stack, such as radical.pilot or radical.entk. Profiles are accepted in two forms: in a directory, or in a tarball (of such a directory). In the latter case, the tarball are extracted into *$TMP*, and then handled just as the directory case.

If no *sid* (session ID) is specified, that ID is derived from the directory name.

concurrency(*state=None*, *event=None*, *time=None*, *sampling=None*)

This method accepts the same set of parameters as the *ranges()* method, and will use the *ranges()* method to obtain a set of ranges. It will return a time series, counting the number of tasks which are concurrently matching the ranges filter at any point in time.

The additional parameter *sampling* determines the exact points in time for which the concurrency is computed, and thus determines the sampling rate for the returned time series. If not specified, the time series will contain all points at which the concurrency changed. If specified, it is interpreted as second (float) interval at which, after the starting point (begin of first event matching the filters) the concurrency is computed.

Returned is an ordered list of tuples:

```
[ [time_0, concurrency_0],
  [time_1, concurrency_1],
  ...
  [time_n, concurrency_n] ]
```

where *time_n* is represented as *float*, and *concurrency_n* as *int*.

Example:

```
session.filter(etype='task').concurrency(state=[rp.AGENT_EXECUTING,
    rp.AGENT_STAGING_OUTPUT_PENDING])
```

consistency(*mode=None*)

Performs a number of data consistency checks, and returns a set of UIDs for entities which have been found to be inconsistent. The method accepts a single parameter *mode* which can be a list of strings defining what consistency checks are to be performed. Valid strings are:

- state_model: check if all entity states are in adherence to the respective entity state model
- event_model: check if all entity events are in adherence to the respective entity event model
- timestamps: check if events and states are recorded with correct ordering in time.

If not specified, the method will execute all three checks.

After this method has been run, each checked entity will have more detailed consistency information available via:

```
entity.consistency['state_model'] (bool)
entity.consistency['event_model'] (bool)
entity.consistency['timestamps' ] (bool)
entity.consistency['log' ]        (list of strings)
```

The boolean values each indicate consistency of the respective test, the *log* will contain human readable information about specific consistency violations.

duration(*state=None*, *event=None*, *time=None*, *ranges=None*)

This method accepts the same set of parameters as the *ranges()* method, and will use the *ranges()* method to obtain a set of ranges. It will return the sum of the durations for all resulting & collapsed ranges.

Example:

```
session.duration(state=[rp.NEW, rp.FINAL]))
```

where *rp.FINAL* is a list of final task states.

ranges(*state=None*, *event=None*, *time=None*, *collapse=True*)

Gets a set of initial and final conditions, and computes time ranges in accordance to those conditions from all session entities. The resulting set of ranges is then collapsed to the minimal equivalent set of ranges covering the same set of times.

Please refer to the *Entity.ranges* documentation on detail on the constrain parameters.

Setting 'collapse' to 'True' (default) will prompt the method to collapse the resulting set of ranges.

rate(*state=None*, *event=None*, *time=None*, *sampling=None*, *first=False*)

This method accepts the same parameters as the *timestamps()* method: it will count all matching events and state transitions as given, and will return a time series of the rate of how many of those events and/or transitions occurred per second.

The additional parameter *sampling* determines the exact points in time for which the rate is computed, and thus determines the sampling rate for the returned time series. If not specified, the time series will contain all points at which and event occurred, and the rate value will only be determined by the time passed between two consecutive events. If specified, it is interpreted as second (float) interval at which, after the starting point (begin of first event matching the filters) the rate is computed.

Returned is an ordered list of tuples:

```
[ [time_0, rate_0] ,
  [time_1, rate_1] ,
  ...
  [time_n, rate_n] ]
```

where *time_n* is represented as *float*, and *rate_n* as *int*.

The *time* parameter is expected to be a single tuple, or a list of tuples, each defining a pair of start and end time which are used to constrain the resulting time series.

The 'first' is defined, only the first matching event fir the selected entities is considered viable.

Example:

```
session.filter(etype='task').rate(state=[rp.AGENT_EXECUTING])
```

timestamps (*state=None*, *event=None*, *time=None*, *first=False*)

This method accepts a set of conditions, and returns the list of timestamps for which those conditions applied, i.e., for which state transitions or events are known which match the given 'state' or 'event' parameter. If no match is found, an empty list is returned.

Both *state* and *event* can be lists, in which case the union of all timestamps are returned.

The *time* parameter is expected to be a single tuple, or a list of tuples, each defining a pair of start and end time which are used to constrain the matching timestamps.

If *first* is set to *True*, only the timestamps for the first matching events (per entity) are returned.

The returned list will be sorted.

tzero (*t*)

Setting a *tzero* timestamp will shift all timestamps for all entities in this session by that amount. This simplifies the alignment of multiple sessions, or the focus on specific events.

usage (*alloc_entity*, *alloc_events*, *block_entity*, *block_events*, *use_entity*, *use_events*)

This method creates a dict with three entries: *alloc*, *block*, *use*. Those three dict entries in turn have a a dict of entity IDs for all entities which have blocks in the respective category, and foreach of those entity IDs the dict values will be a list of rectangles.

A resource is considered:

- *alloc* (allocated) when it is owned by the RCT application;
- *block* (blocked) when it is reserveed for a specific task;
- *use* (used) when it is utilized by that task.

Each of the rectangles represents a continuous block of resources which is alloced/blocked/used:

- x_0 time when *alloc/block/usage* begins;
- x_1 time when *alloc/block/usage* ends;
- y_0 lowest index of a continuous block of resource IDs;
- y_1 highest index of a continuous block of resource IDs.

Any specific entity (pilot, task) can have a **set** of such resource blocks, for example, a task might be placed over multiple, non-consecutive nodes:

- gpu and cpu resources are rendered as separate blocks (rectangles).

Parameters

- **alloc_entity** (Entity) – *Entity* instance which allocates resources
- **alloc_events** (*list*) – event tuples which specify allocation time
- **block_entity** (Entity) – *Entity* instance which blocks resources
- **block_events** (*list*) – event tuples which specify blocking time
- **use_entity** (Entity) – *Entity* instance which uses resources
- **use_events** (*list*) – event tuples which specify usage time

Example:

```
usage('pilot', [{ru.STATE: None, ru.EVENT: 'bootstrap_0_start'},
                {ru.STATE: None, ru.EVENT: 'bootstrap_0_stop' }],
      'task' , [{ru.STATE: None, ru.EVENT: 'schedule_ok'      },
                {ru.STATE: None, ru.EVENT: 'unschedule_stop'  }],
      'task' , [{ru.STATE: None, ru.EVENT: 'exec_start'       },
                {ru.STATE: None, ru.EVENT: 'exec_stop'        }])
```

9.2 Entity

class radical.analytics.**Entity**(*_uid*, *_profile*, *_details*)

__init__(*_uid*, *_profile*, *_details*)

Parameters

- **uid** (*str*) – an ID assumed to be unique in the scope of an RA Session
- **profile** – a list of profile events for this entity
- **details** – a dictionary of complementary information on this entity

duration(*state=None*, *event=None*, *time=None*, *ranges=None*)
This method accepts a set of initial and final conditions, interprets them as documented in the *ranges()* method (which has the same signature), and then returns the difference between the resulting timestamps.

ranges(*state=None*, *event=None*, *time=None*, *expand=False*, *collapse=True*)
This method accepts a set of initial and final conditions, in the form of range of state and or event specifiers:

```
entity.ranges(state=[['INITIAL_STATE_1', 'INITIAL_STATE_2'],
                     'FINAL_STATE_1',   'FINAL_STATE_2'  ]],
              event=[[ initial_event_1,   initial_event_2 ]
                     [ final_event_1,     final_event_2   ]],
              time =[[2.0, 2.5], [3.0, 3.5]])
```

More specifically, the *state* and *event* parameter are expected to be a tuple, where the first element defines the initial condition, and the second element defines the final condition. The *time* parameter is expected to be a single tuple, or a list of tuples, each defining a pair of start and end time which are used to constrain the resulting ranges. States are expected as strings, events as full event tuples:

```
[ru.TIME,  ru.NAME, ru.UID,  ru.STATE, ru.EVENT, ru.MSG,  ru.ENTITY]
```

where empty fields are not applied in the filtering - all other fields must match exactly. The events can also be specified as dictionaries, which then don't need to have all fields set.

The method will:

- determine the *earliest* timestamp when any of the given initial conditions have been met, which can be either an event or a state;
- determine the *next* timestamp when any of the given final conditions have been met (when *expand* is set to *False* [default]) OR
- determine the *last* timestamp when any of the given final conditions have been met (when *expand* is set to *True*)

From that final point in time the search for the next initial condition applies again, which may result in another time range to be found. The method returns the set of found ranges, as a list of *[start, end]* time tuples.

The resulting ranges are constrained by the *time* constraints, if such are given.

Note that with *expand=True*, at most one range will be found.

Setting 'collapse' to 'True' (default) will prompt the method to collapse the resulting set of ranges.

The returned ranges are time-sorted

Example:

```
task.ranges(state=[rp.NEW, rp.FINAL]))
task.ranges(event=[{ru.NAME : 'exec_start'},
                   {ru.NAME : 'exec_ok'}])
```

timestamps(*state=None*, *event=None*, *time=None*)

This method accepts a set of conditions, and returns the list of timestamps for which those conditions applied, i.e., for which state transitions or events are known which match the given 'state' or 'event' parameter. If no match is found, an empty list is returned.

Both *state* and *event* can be lists, in which case the union of all timestamps are returned.

The *time* parameter is expected to be a single tuple, or a list of tuples, each defining a pair of start and end time which are used to constrain the matching timestamps.

The returned list will be sorted.

9.3 Experiment

class radical.analytics.**Experiment**(*sources*, *stype*)

__init__(*sources*, *stype*)

This class represents an RCT experiment, i.e., a series of RA sessions which are collectively analyzed.

sources is expected to be a list of tuples of session source paths pointing to tarballs or session directories. The order of tuples in the list determines the default order used in plots etc.

The session type *stype* will be uniformely applied to all sessions.

utilization(*metrics*, *rtype='cpu'*, *udurations=None*)

return five dictionaries:

- provided resources
- consumed resources
- absolute stats
- relative stats

- information about resource utilization

The resource dictionaries have the following structures:

```
provided = {
    <session_id> : {
        'metric_1' : {
            'uid_1'         : [float, list],
            'uid_2'         : [float, list],
            ...
        },
        'metric_2' : {
            'uid_1'         : [float, list],
            'uid_2'         : [float, list],
            ...
        },
        ...
    },
    ...
}
consumed = {
    <session_id> : {
        'metric_1' : {
            'uid_1'          : [float, list]
            'uid_2'          : [float, list],
            ...
        },
        'metric_2' :          {
            'uid_1'          : [float, list],
            'uid_2'          : [float, list],
            ...
        },
        ...
    },
    ...
}
```

float is always in tasks of *resource * time*, (think *core-hours*), *list* is a list of 4-tuples *[t0, t1, r0, r1]* which signify at what specific time interval (*t0 to t1*) what specific resources (*r0 to r1*) have been used. The task of the resources are here dependent on the session type: only RP sessions are supported at the moment where those resource values are indexes in to the list of cores used in that specific session (offset over multiple pilots, if needed).

9.4 utils

radical.analytics.**get_plotsize**(*width*, *fraction=1*, *subplots=(1, 1)*)

Sets aesthetic figure dimensions to avoid scaling in latex.

Parameters

- **width** (*float*) – Width in points (pts).
- **fraction** (*float*) – Fraction of the width which you wish the figure to occupy.
- **subplots** (*tuple*) – Number of raws and number of columns of the plot.

Returns **fig_dim** – Dimensions of figure in inches.

Return type tuple

radical.analytics.**get_mplstyle**(*name*)
Returns the installation path of a Matplotlib style.

Parameters **name** (*string*) – Filename ending in .txt.

Returns **path** – Normalized path.

Return type string

radical.analytics.**stack_transitions**(*series*, *tresource*, *to_stack*)
Creates data frames for each metric and combines them into one data frame for alignment. Since transitions obviously happen at arbitrary times, the timestamps for metric A may see no transitions for metric B. When using a combined timeline, we end up with NaN entries for some metrics on most timestamp, which in turn leads to gaps when plotting. So we fill the NaN values with the previous valid value, which in our case holds until the next transition happens.

Parameters

- **series** (*dict*) – Pairs of timestamps for each metric of each type of resource. E.g. series['cpu']['term'] = [[0.0, 0.0], [302.4374113082886, 100.0], [304.6761999130249, 0.0]].
- **tresource** (*string*) – Type of resource. E.g., 'cpu' or 'gpu'.
- **to_stack** (*list*) – List of metrics to stack. E.g., ['bootstrap', 'exec_cmd', 'schedule', 'exec_rp', 'term', 'idle'].

Returns **stacked** – Columns: time and one for each metric. Rows: timestamp and percentage / amount of resource utilization for each metric at that point in time.

Return type pandas.DataFrame

radical.analytics.**get_pilot_series**(*session*, *pilot*, *tmap*, *resrc*, *percent=True*)
Derives the series of pilot resource transition points from the metrics.

Parameters

- **session** (*ra.Session*) – The Session object of RADICAL-Analytics created from a RCT sandbox.
- **pilot** (*ra.Entity*) – The pilot object of session.
- **tmap** (*dict*) – Map events to transition points in which a metric changes its owner. E.g., [{1: 'bootstrap_0_start'}, 'system', 'bootstrap'] defines bootstrap_0_start as the event in which resources pass from the system to the bootstrapper.
- **resrc** (*list*) – Type of resources. E.g., ['cpu', 'gpu'].
- **percent** (*bool*) – Whether we want to return resource utilization as percentage of the total resources available or as count of a type of resource.

Returns

- **p_resrc** (*dict*) – Amount of resources in the pilot.
- **series** (*dict*) – List of time series per metric and resource type. E.g., series['cpu']['term'] = [[0.0, 0.0], [302.4374113082886, 100.0], [304.6761999130249, 0.0]].
- **x** (*dict*) – Mix and max value of the X-axes.

radical.analytics.**get_plot_utilization**(*metrics*, *consumed*, *t_zero*, *sid*)
Calculates the resources utilized by a set of metrics. Utilization is calculated for each resource without stacking and aggregation. May take hours or days with >100K tasks, 100K resource items. Use get_pilot_series and stack_transitions instead.

Parameters

- **metrics** (*list*) – Each element is a list with name, metrics and color. E.g., ['Bootstrap', ['boot', 'setup_1'], '#c6dbef'].
- **consumed** (*dict*) – min-max timestamp and resource id range for each metric and pilot. E.g., {'boot': {'pilot.0000': [[2347.582849740982, 2365.6164498329163, 0, 167]}.
- **t_zero** (*float*) – Start timestamp for the pilot.
- **sid** (*string*) – Identifier of a ra.Session object.

Returns

- **legend** (*dict*) – keys: Type of resource ('cpu', 'gpu'); values: list of matplotlib.lines.Line2D objects for the plot's legend.
- **patches** (*dict*) – keys: Type of resource ('cpu', 'gpu'); values: list of matplotlib.patches.Rectangle. Each rectangle represents the utilization for a set of resources.
- **x** (*dict*) – Mix and max value of the X-axes.
- **y** (*dict*) – Mix and max value of the Y-axes.

radical.analytics.**get_pilots_zeros**(*ra_exp_obj*)

Calculates when a set of pilots become available.

Parameters **ra_exp_obj** (*ra.Experiment*) – RADICAL-Analytics Experiment object with all the pilot entity objects for which to calculate the starting timestamp.

Returns **p_zeros** – Session ID, pilot ID and starting timestamp. E.g., {'re.session.login1.lei.018775.0005': {'pilot.0000': 2347.582849740982}}.

Return type dict

radical.analytics.**to_latex**(*data*)

Transforms the input string(s) so that it can be used as latex compiled plot label, title etc. Escapes special characters with a slash.

Parameters **data** (*list or str*) – An individual string or a list of strings to transform.

Returns **data** – Transformed data.

Return type list of str

radical.analytics.**tabulate_durations**(*durations*)

Takes a dict of durations as defined in rp.utils (e.g., *rp.utils.PILOT_DURATIONS_DEBUG*) and returns a list of durations with their start and stop timestamps. That list can be directly converted to a panda.df.

Parameters **durations** (*dict*) – Dict of lists of dicts/lists of dicts. It contains details about states and events.

Returns **data** – List of dicts, each dict containing 'Duration Name', 'Start Timestamp' and 'Stop Timestamp'.

Return type list

Index

Symbols

C

D

E

G

R

S

T

U