UNIVERSITY OF NOTRE DAME

Resource Management for Dynamic Function Distribution with Parsl and Work Queue

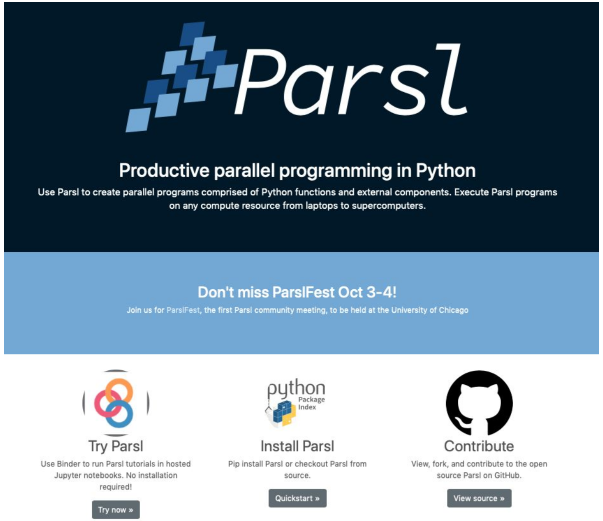
Douglas Thain, Ben Tovar,
Thanh Son Phung, and Barry Sly-Delgado
Parsl / FuncX Workshop, 27 October 2021

CCTools

Parsl + Work Queue for Scalable Apps

CCTools

http://parsl-project.org



Parsl

Productive parallel programming in Python

Use Parsl to create parallel programs comprised of Python functions and external components. Execute Parsl programs on any compute resource from laptops to supercomputers.

Don't miss ParslFest Oct 3-4!

Join us for ParslFest, the first Parsl community meeting, to be held at the University of Chicago

Try Parsl

Use Binder to run Parsl tutorials in hosted Jupyter notebooks. No installation required!

Try now »

Install Parsl

Pip install Parsl or checkout Parsl from source.

Quickstart »

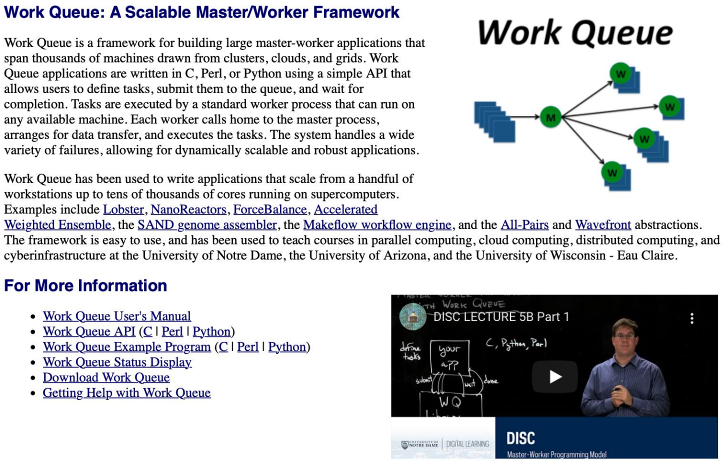
Contribute

View, fork, and contribute to the open source Parsl on GitHub.

View source »

Powerful Pythonic Workflow Programming Model

http://ccl.cse.nd.edu



Work Queue: A Scalable Master/Worker Framework

Work Queue is a framework for building large master-worker applications that span thousands of machines drawn from clusters, clouds, and grids. Work Queue applications are written in C, Perl, or Python using a simple API that allows users to define tasks, submit them to the queue, and wait for completion. Tasks are executed by a standard worker process that can run on any available machine. Each worker calls home to the master process, arranges for data transfer, and executes the tasks. The system handles a wide variety of failures, allowing for dynamically scalable and robust applications.

Work Queue has been used to write applications that scale from a handful of workstations up to tens of thousands of cores running on supercomputers. Examples include Lobster, NanoReactors, ForceBalance, Accelerated Weighted Ensemble, the SAND genome assembler, the Makeflow workflow engine, and the All-Pairs and Wavefront abstractions. The framework is easy to use, and has been used to teach courses in parallel computing, cloud computing, distributed computing, and cyberinfrastructure at the University of Notre Dame, the University of Arizona, and the University of Wisconsin - Eau Claire.

For More Information

- Work Queue User's Manual
- Work Queue API (C | Perl | Python)
- Work Queue Example Program (C | Perl | Python)
- Work Queue Status Display
- Download Work Queue
- Getting Help with Work Queue

Scalable, Portable, Robust Distributed Execution

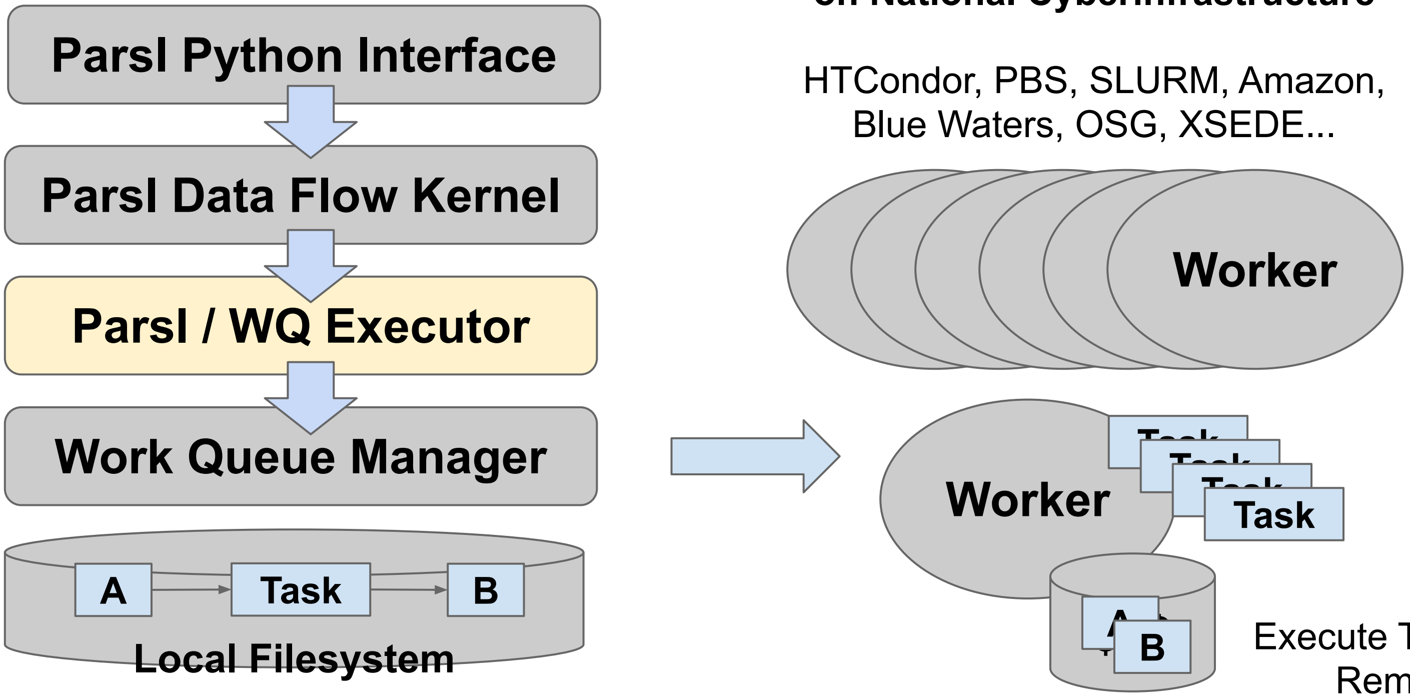
System Architecture

Generate Funcs

Order Functions (Futures)

Functions -> Tasks

Schedule Tasks



Parsl Python Interface

Parsl Data Flow Kernel

Parsl / WQ Executor

Work Queue Manager

A

Task

B

Local Filesystem

Worker

Worker

Task

B

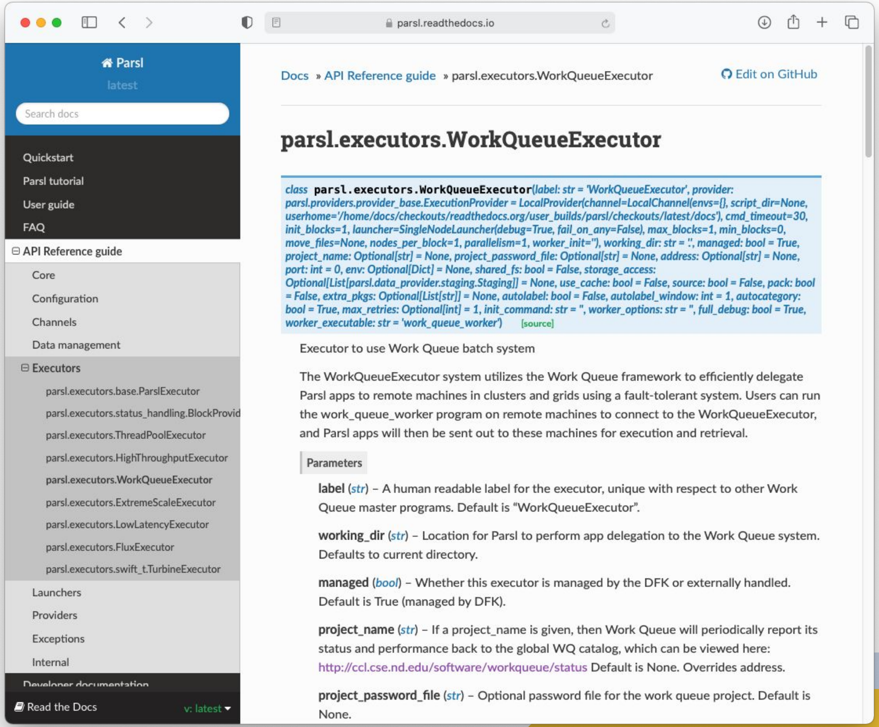
Thousands of Workers on National Cyberinfrastructure

HTCondor, PBS, SLURM, Amazon, Blue Waters, OSG, XSEDE...

Execute Tasks Remotely on Local Disk

Configuring Parsl + WQ

```
import parsl

from parsl.executors import WorkQueueExecutor


config = parsl.config.Config(

    executors=[

        WorkQueueExecutor(

            label="wq-parsl-app",

            port=9123,

            project_name="wq-parsl-app",

            shared_fs=False,

            full_debug = True,

) ] )
```


parsl.readthedocs.io
Parsl
latest
Search docs
Quickstart
Parsl tutorial
User guide
FAQ
⊟ API Reference guide
Core
Configuration
Channels
Data management
⊟ Executors
parsl.executors.base.ParslExecutor
parsl.executors.status_handling.BlockProvid
parsl.executors.ThreadPoolExecutor
parsl.executors.HighThroughputExecutor
parsl.executors.WorkQueueExecutor
parsl.executors.ExtremeScaleExecutor
parsl.executors.LowLatencyExecutor
parsl.executors.FluxExecutor
parsl.executors.swift_t.TurbineExecutor
Launchers
Providers
Exceptions
Internal
Developer documentation
Read the Docs
v: latest
Docs » API Reference guide » parsl.executors.WorkQueueExecutor
Edit on GitHub

parsl.executors.WorkQueueExecutor

class **parsl.executors.WorkQueueExecutor**(*label: str = 'WorkQueueExecutor', provider: parsl.providers.provider_base.ExecutionProvider = LocalProvider(channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/docs'), cmd_timeout=30, init_blocks=1, launcher=SingleNodeLauncher(debug=True, fail_on_any=False), max_blocks=1, min_blocks=0, move_files=None, nodes_per_block=1, parallelism=1, worker_init=''), working_dir: str = '.', managed: bool = True, project_name: Optional[str] = None, project_password_file: Optional[str] = None, address: Optional[str] = None, port: int = 0, env: Optional[Dict] = None, shared_fs: bool = False, storage_access: Optional[List[parsl.data_provider.staging.Staging]] = None, use_cache: bool = False, source: bool = False, pack: bool = False, extra_pkgs: Optional[List[str]] = None, autolabel: bool = False, autolabel_window: int = 1, autocategory: bool = True, max_retries: Optional[int] = 1, init_command: str = '', worker_options: str = '', full_debug: bool = True, worker_executable: str = 'work_queue_worker'*) [source]

Executor to use Work Queue batch system

The WorkQueueExecutor system utilizes the Work Queue framework to efficiently delegate Parsl apps to remote machines in clusters and grids using a fault-tolerant system. Users can run the work_queue_worker program on remote machines to connect to the WorkQueueExecutor, and Parsl apps will then be sent out to these machines for execution and retrieval.

Parameters

- **label** (*str*) – A human readable label for the executor, unique with respect to other Work Queue master programs. Default is "WorkQueueExecutor".
- **working_dir** (*str*) – Location for Parsl to perform app delegation to the Work Queue system. Defaults to current directory.
- **managed** (*bool*) – Whether this executor is managed by the DFK or externally handled. Default is True (managed by DFK).
- **project_name** (*str*) – If a project_name is given, then Work Queue will periodically report its status and performance back to the global WQ catalog, which can be viewed here: http://ccl.cse.nd.edu/software/workqueue/status Default is None. Overrides address.
- **project_password_file** (*str*) – Optional password file for the work queue project. Default is None.

Common Challenges

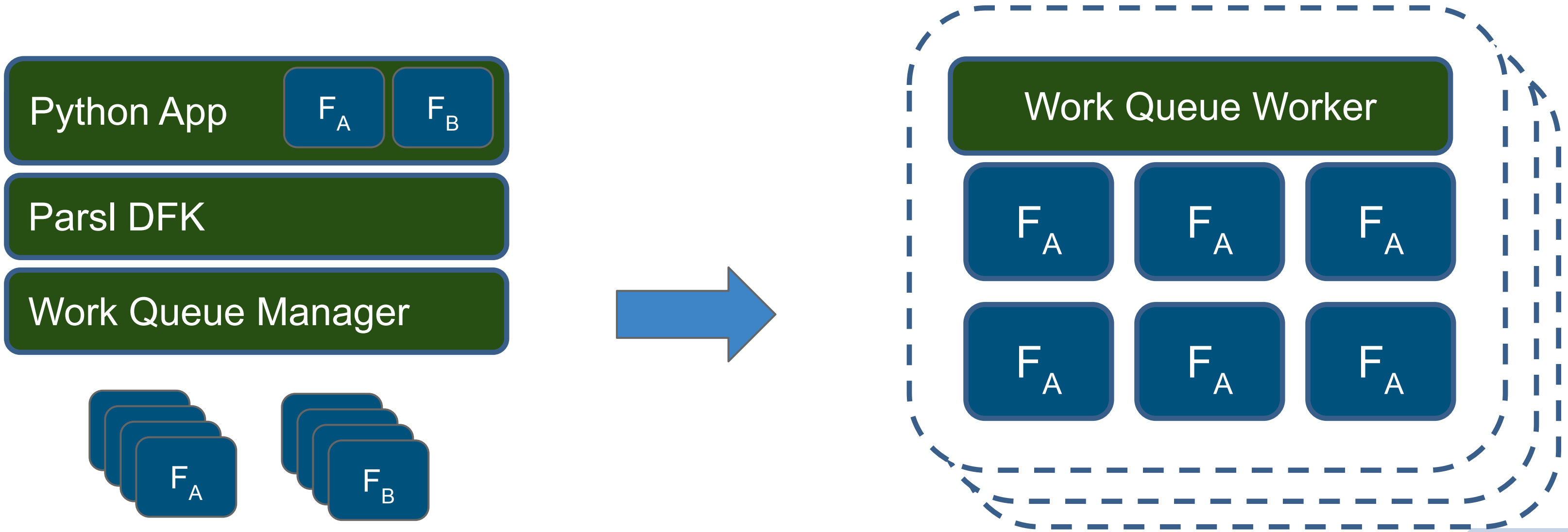
Two common problems of scaling up:

- What resources should be assigned to a function call?
- What software dependencies does this function need?

How can we solve these problems automatically at runtime, without requiring the user to make advance declarations?

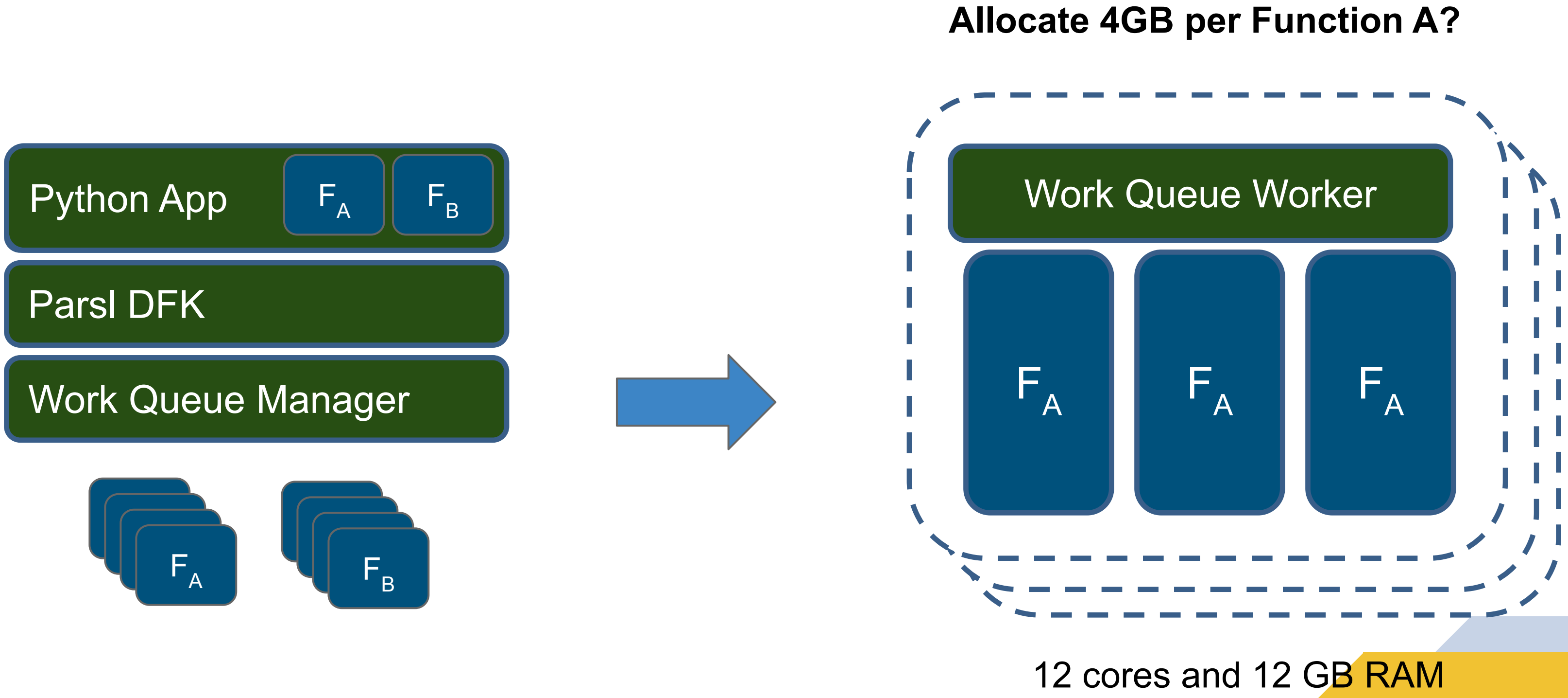
Packing Functions Into Manycore Nodes

Allocate 2GB per Function A?

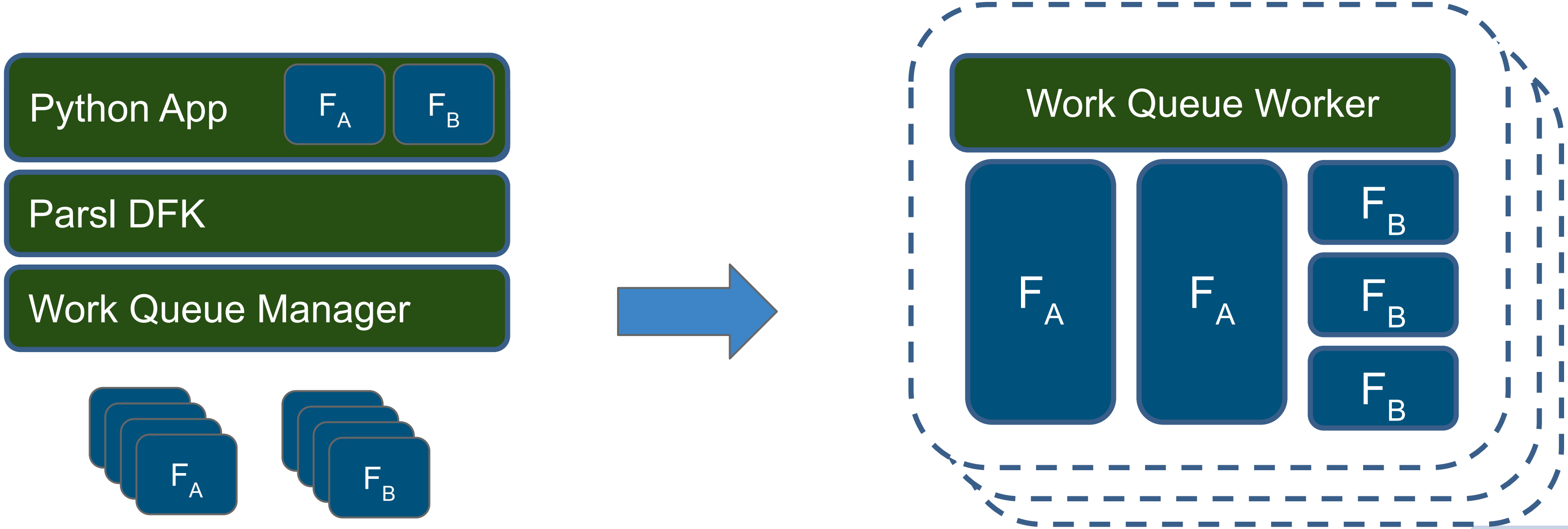


12 cores and 12 GB RAM

Packing Functions Into Manycore Nodes



Packing Functions Into Manycore Nodes

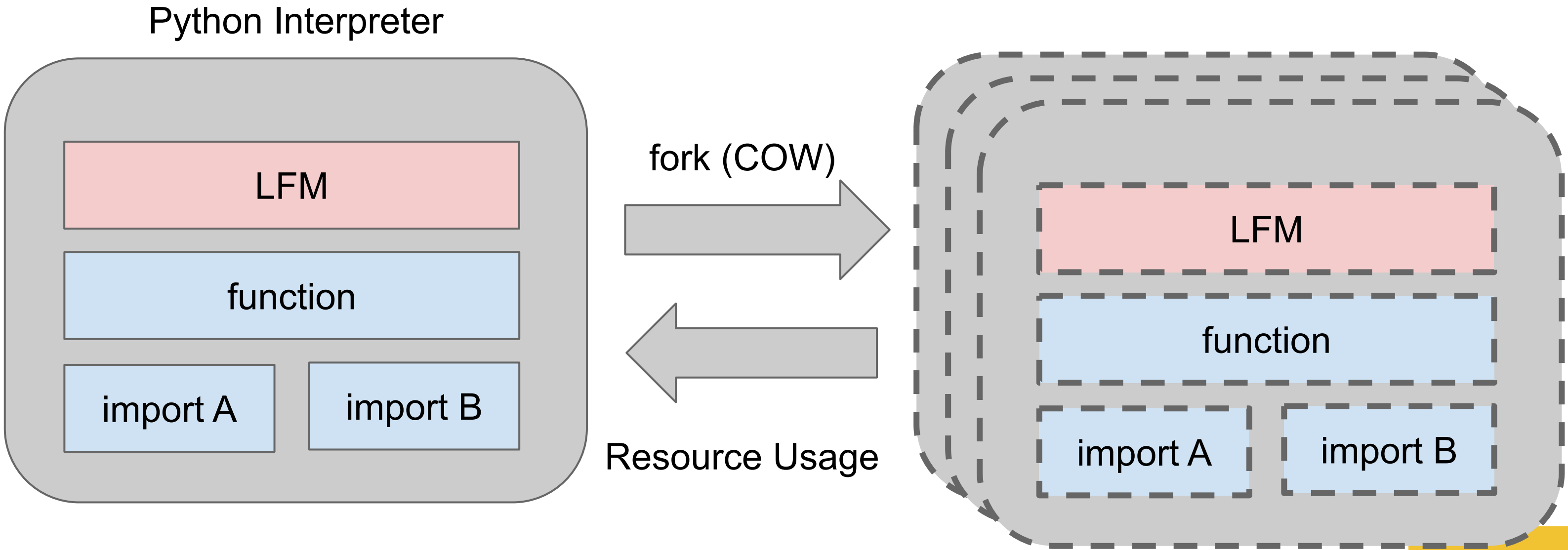


12 cores and 12 GB RAM

How to measure a single function call?

Ben Tovar

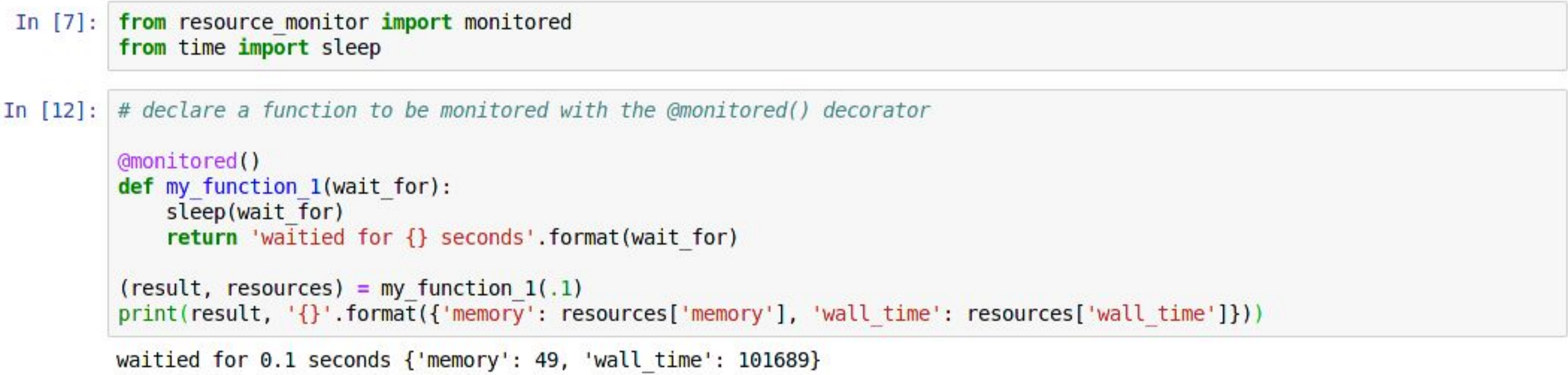
LFM - Lightweight Function Monitor



Tim Shaffer, Zhuozhao Li, Ben Tovar, Yadu Babuji, TJ Dasso, Zoe Surma, Kyle Chard, Ian Foster, and Douglas Thain, **Lightweight Function Monitors for Fine-Grained Management in Large Scale Python Applications**, IEEE International Parallel & Distributed Processing Symposium, May, 2021. DOI: 10.1109/IPDPS49936.2021.00088

Lightweight Function Monitors (LFMs)

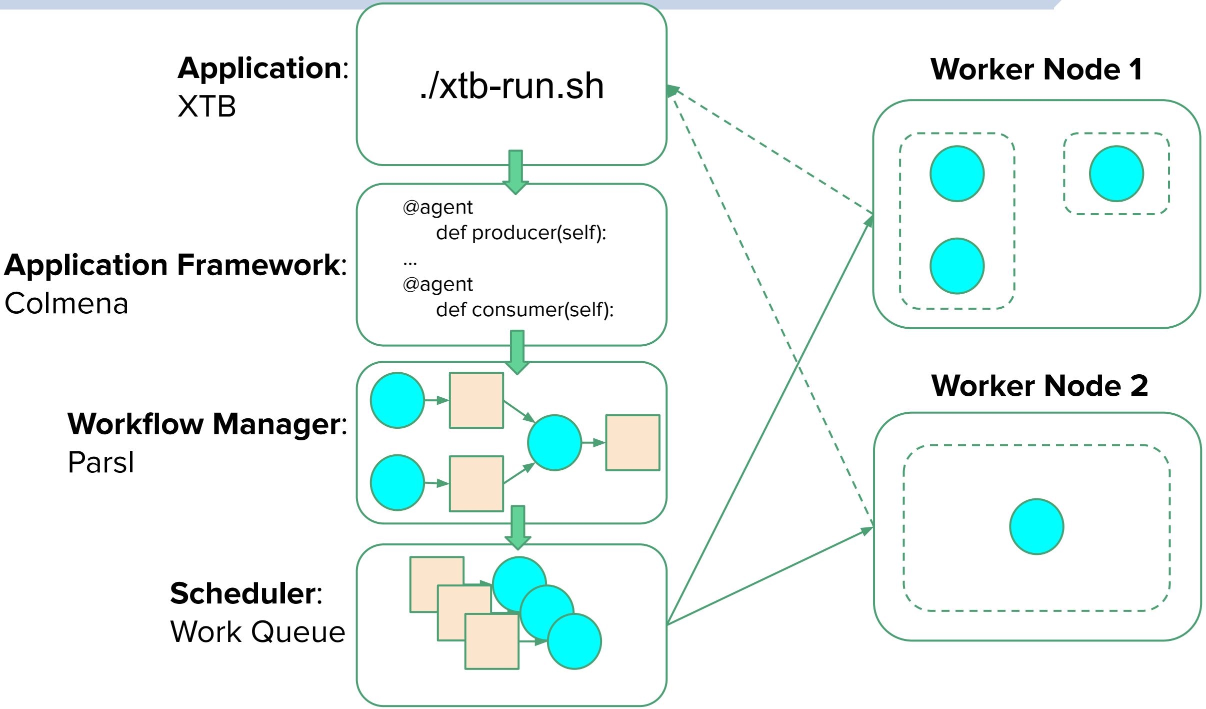
Activate LFMs with an import and the @monitored keyword

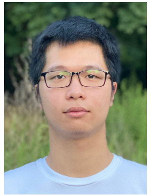


```
In [7]:  from resource_monitor import monitored
         from time import sleep
```

```
In [12]: # declare a function to be monitored with the @monitored() decorator

         @monitored()
         def my_function_1(wait_for):
             sleep(wait_for)
             return 'waitied for {} seconds'.format(wait_for)

         (result, resources) = my_function_1(.1)
         print(result, '{}'.format({'memory': resources['memory'], 'wall_time': resources['wall_time']}))

         waitied for 0.1 seconds {'memory': 49, 'wall_time': 101689}
```

Example: Colmena-XTB Application





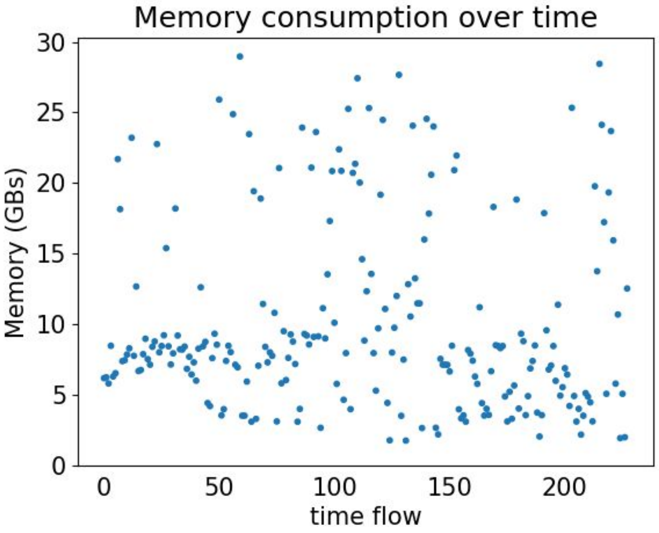
Thanh Son Phung

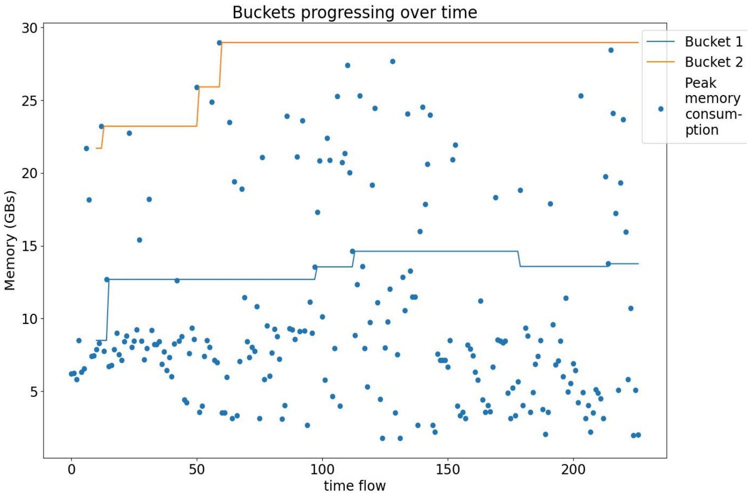
Tasks:

- Are of two types: inference and simulation
- Display significant differences in resource consumption

Thanh Son Phung, Logan Ward, Kyle Chard, Douglas Thain, "**Not All Tasks are Created Equal: Adaptive Resource Allocation for Hetergeneous Tasks in Dynamic Workflows"**, WORKS Workshop at Supercomputing 2021.

Memory Consumption of Colmena-XTB's Tasks





Problem

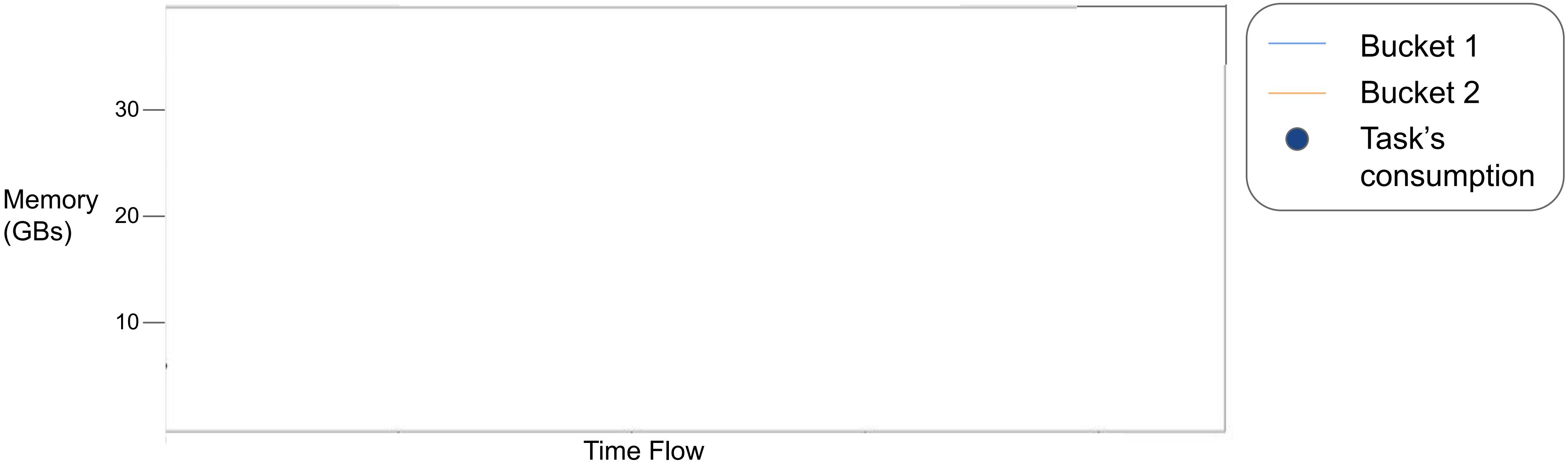
Tasks can consume as low as 2 GBs or as high as 30 GBs of RAM!

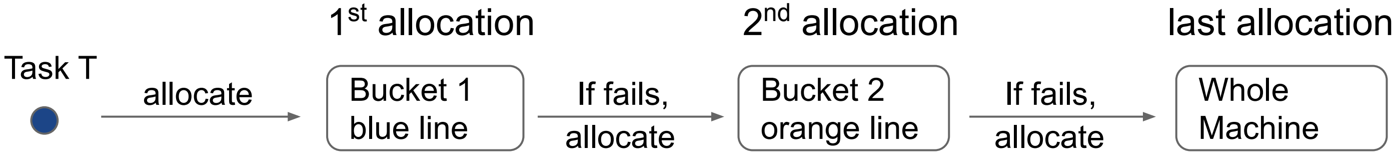
Solution

Bucket tasks with similar consumption and allocate new tasks accordingly.

K-means Bucketing

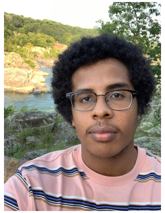
Buckets progressing over time



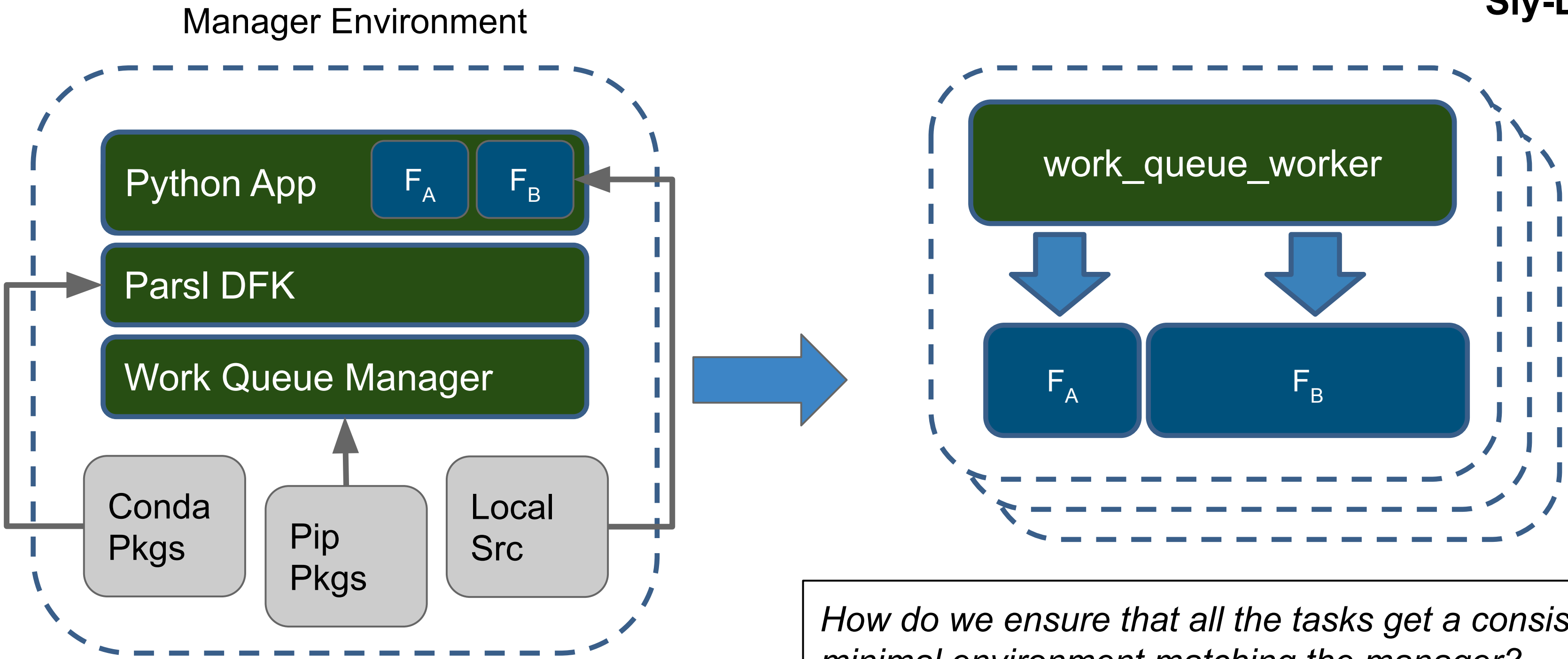


Task T ● allocate → 1st allocation: Bucket 1 blue line → If fails, allocate → 2nd allocation: Bucket 2 orange line → If fails, allocate → last allocation: Whole Machine

Run Time Dependency Management



Barry Sly-Delgado



How do we ensure that all the tasks get a consistent, minimal environment matching the manager?

Poncho Toolkit

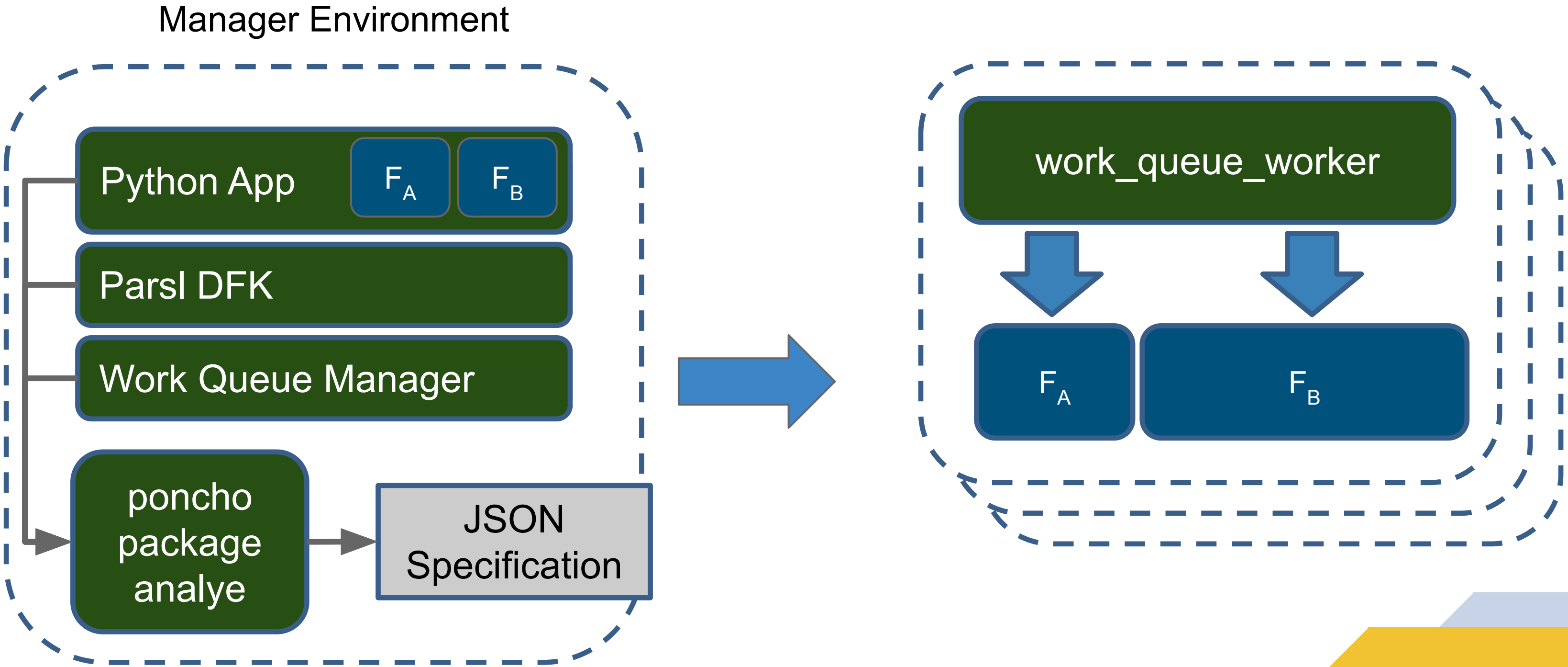
The Poncho Toolkit allows users to create and deploy self contained Python environments at user level in arbitrary distributed systems via a JSON specification file.

- poncho_package_analyze
- poncho_package_create
- poncho_package_run

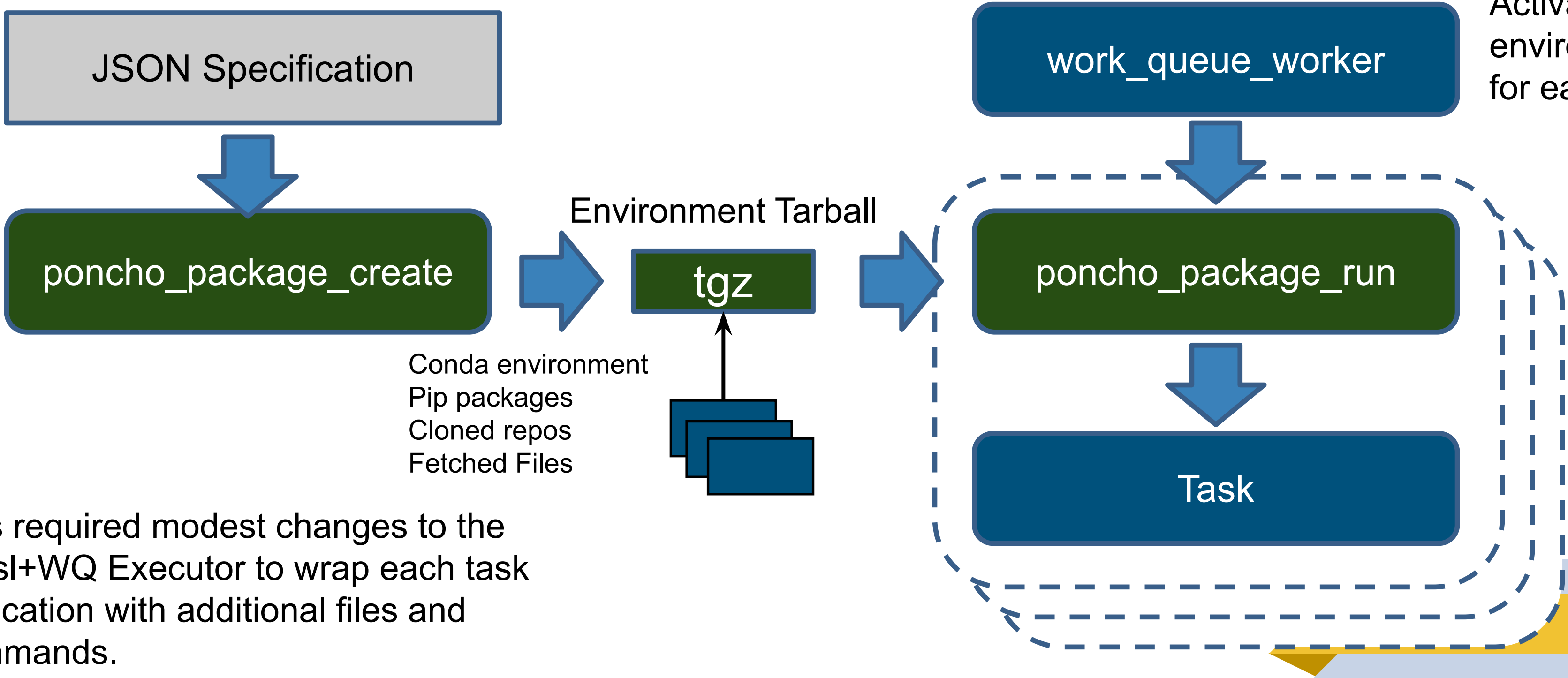
https://cctools.readthedocs.io/en/latest/poncho/

```
"conda":{
        "channels":[
                "defaults",
                "conda-forge"
        ],
        "packages":[
                "ndcctools=7.3.0",
                "parsl=1.1.0",
        ]
},
"pip": [
        "topcoffea"
]
"git": {
         "DATA_DIR": {
                 "remote": "http://.../repo.git"
         }
```

Run Time Dependency Management



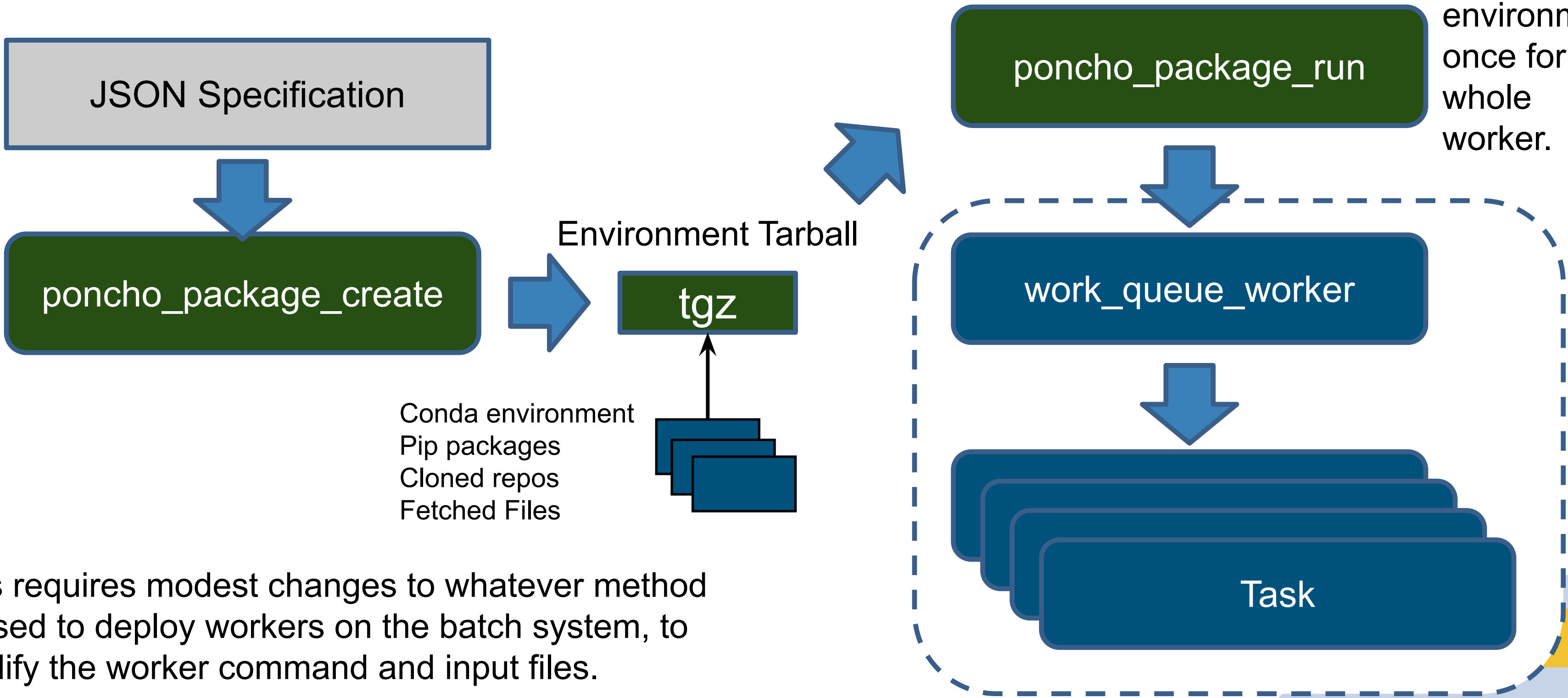
Package per Task



Activate environment for each task.

This required modest changes to the Parsl+WQ Executor to wrap each task invocation with additional files and commands.

Package per Worker



Activate environment once for whole worker.

This requires modest changes to whatever method is used to deploy workers on the batch system, to modify the worker command and input files.

First Look: Orders of Magnitude

poncho_package_create

Application	No Versions Specified	All Versions Specified
TopEFT	2940s	170s
SHADHO	257s	159s

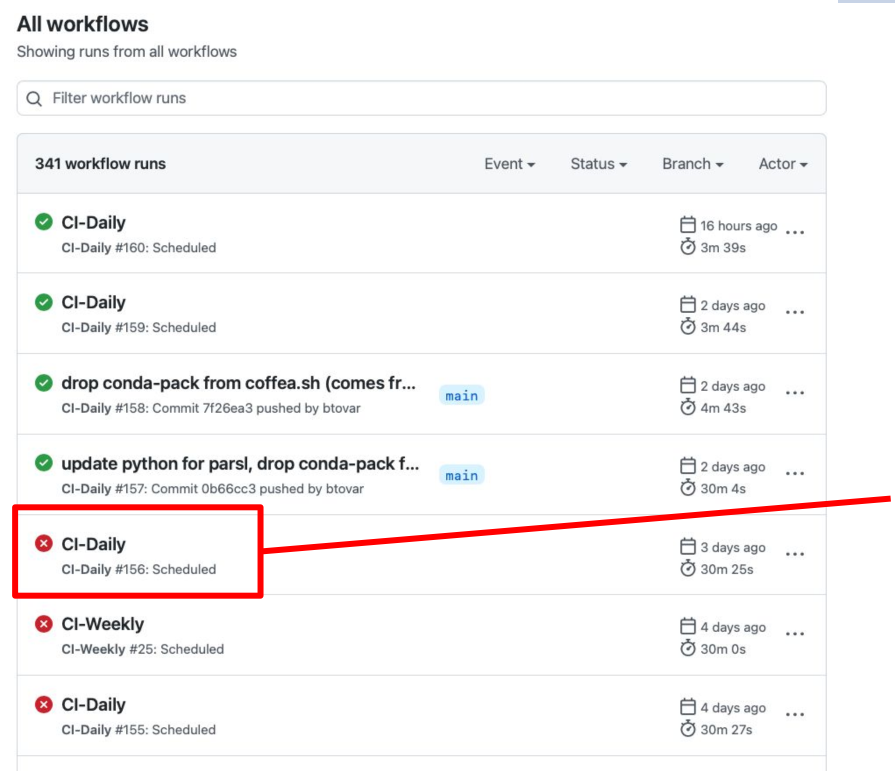
poncho_package_run

Application	Size compressed	Size unpacked	Unpack Time
SHADHO	438MB	1.4GB	12s
TopEFT	594MB	2GB	21s
Colmena-XTB	1.4GB	4.8GB	46s

Next Steps...

- Do we really need all this code just to run a function? (maybe)
- Understanding the dependencies ***actually used*** by a function execution, and how they evolve over time.
- Extending dependency detection to other kinds of resources: databases, executables, file system resources...
- Closing the loop on application configuration: capture discovered resource configurations from multiple runs and use to predict future runs.
- Conveying known application categories from top to bottom through software stacks.

End to End Integration Testing



All workflows

Showing runs from all workflows

Filter workflow runs

341 workflow runs — Event ▾ Status ▾ Branch ▾ Actor ▾

✓ **CI-Daily**
CI-Daily #160: Scheduled — 16 hours ago — 3m 39s

✓ **CI-Daily**
CI-Daily #159: Scheduled — 2 days ago — 3m 44s

✓ **drop conda-pack from coffea.sh (comes fr...** `main`
CI-Daily #158: Commit 7f26ea3 pushed by btovar — 2 days ago — 4m 43s

✓ **update python for parsl, drop conda-pack f...** `main`
CI-Daily #157: Commit 0b66cc3 pushed by btovar — 2 days ago — 30m 4s

✗ **CI-Daily**
CI-Daily #156: Scheduled — 3 days ago — 30m 25s

✗ **CI-Weekly**
CI-Weekly #25: Scheduled — 4 days ago — 30m 0s

✗ **CI-Daily**
CI-Daily #155: Scheduled — 4 days ago — 30m 27s

End-to-end daily test that simply installs parsl+workqueue and runs a trivial example out of the manual to see if it gets the right result.

conda-forge dropped support for python 3.6, resulting in attempts to install taking forever while conda tries to solve an unsolvable dependency problem!

For More Information…

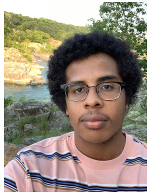
Quick Start:

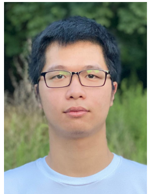
```
conda install -c conda-forge python=3.9 ndcctools parsl
```

https://cctools.readthedocs.io
https://ccl.cse.nd.edu/software/workqueue

btovar@nd.edu

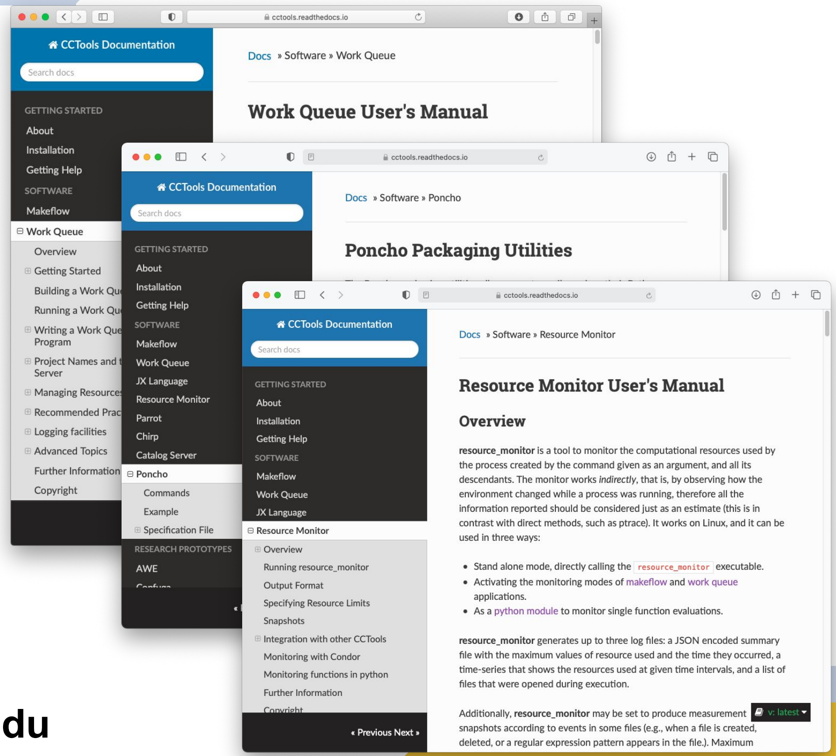
tphung@nd.edu





bslydelg@nd.edu

dthain@nd.edu



Docs » Software » Work Queue

Work Queue User's Manual

Docs » Software » Poncho

Poncho Packaging Utilities

Docs » Software » Resource Monitor

Resource Monitor User's Manual

Overview

resource_monitor is a tool to monitor the computational resources used by the process created by the command given as an argument, and all its descendants. The monitor works *indirectly*, that is, by observing how the environment changed while a process was running, therefore all the information reported should be considered just as an estimate (this is in contrast with direct methods, such as ptrace). It works on Linux, and it can be used in three ways:

- Stand alone mode, directly calling the `resource_monitor` executable.
- Activating the monitoring modes of makeflow and work queue applications.
- As a python module to monitor single function evaluations.

resource_monitor generates up to three log files: a JSON encoded summary file with the maximum values of resource used and the time they occurred, a time-series that shows the resources used at given time intervals, and a list of files that were opened during execution.

Additionally, **resource_monitor** may be set to produce measurement snapshots according to events in some files (e.g., when a file is created, deleted, or a regular expression pattern appears in the file.). Maximum