HTCondor Week 2019

master-worker applications with work queue

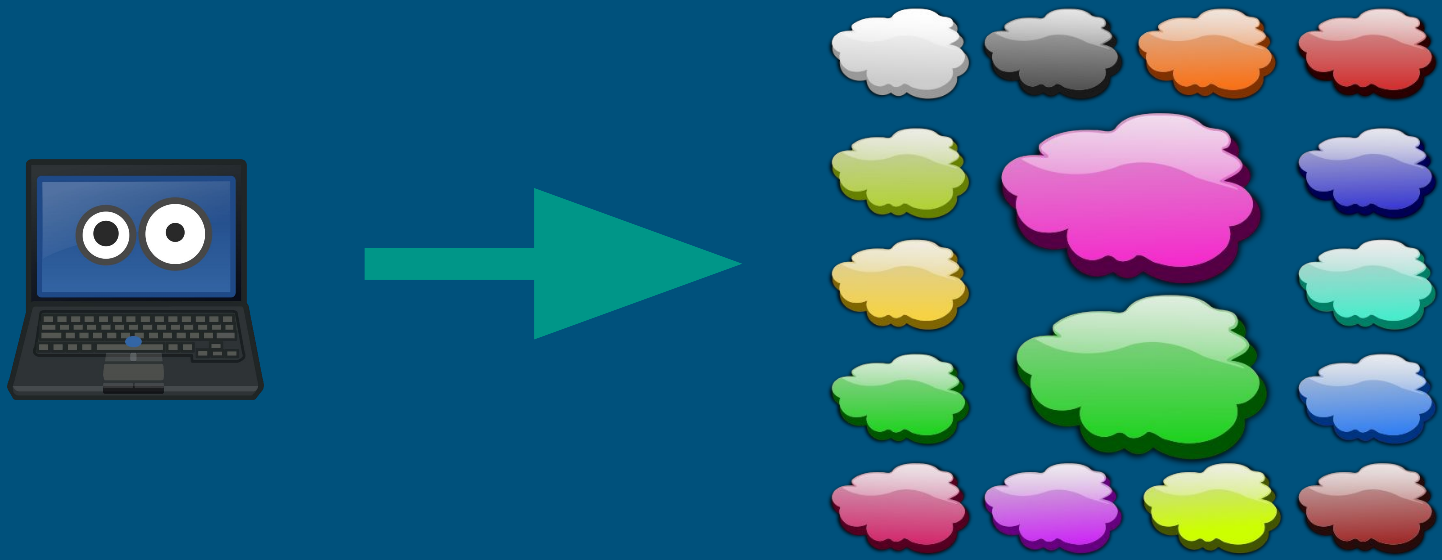
Ben Tovar
btovar@nd.edu

UNIVERSITY OF NOTRE DAME

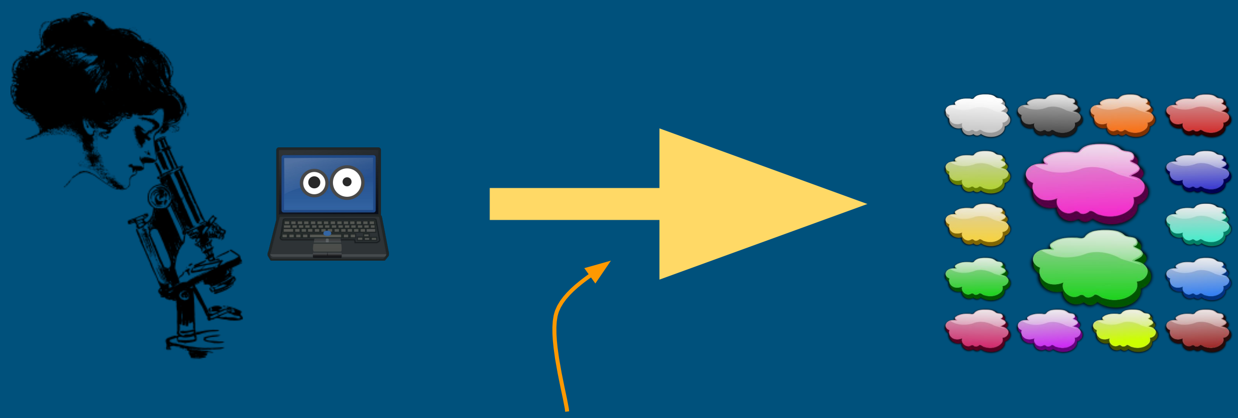
CCTools

where you start

"This demo task runs on my laptop, but I need much more for the real application. It would be great if we can run O(25K) tasks like this on this cloud/grid/cluster I have heard so much about."



who we are

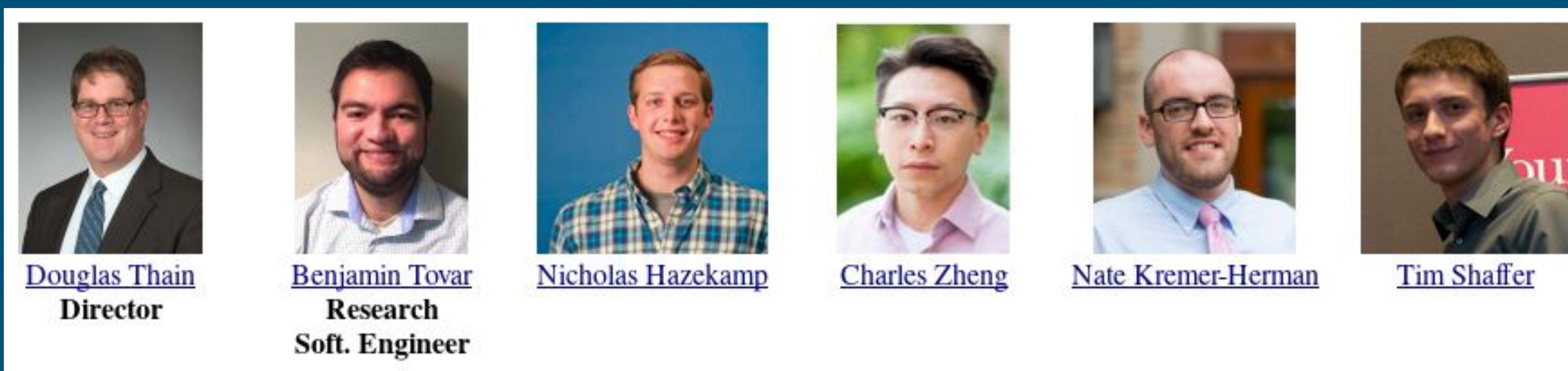


The Cooperative Computing Lab
Computer Science and Engineering
University of Notre Dame

CCL Objectives

- Harness all the resources that are available: desktops, clusters, clouds, and grids.
- Make it easy to scale up from one desktop to national scale infrastructure.
- Provide familiar interfaces that make it easy to connect existing apps together.
- Allow portability across operating systems, storage systems, middleware…
- Make simple things easy, and complex things possible.
- No special privileges required.

Cooperative Computing Lab



CCTools

- Open source, GNU General Public License.
- Compiles in 1-2 minutes, installs in $HOME.
- Runs on Linux, Solaris, MacOS, Cygwin, FreeBSD, …
- Interoperates with many distributed computing systems.
  - Condor, SGE, Torque, Globus, iRODS, Hadoop…

most used components

Makeflow: A portable workflow manager

What to run?

Work Queue: A lightweight distributed execution system

What to run and where to run it?

Chirp: A user-level distributed filesystem

Where to get/put the data?

Parrot: A personal user-level virtual file system

How to read/write the data?

agenda

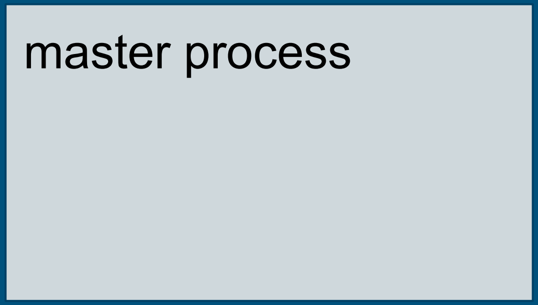
Introduction to master-worker applications
    5min

Writing master-worker applications with work queue
    40 min

Setting-up CCTools
    5min

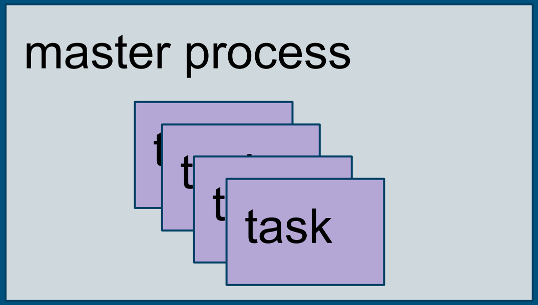
master-worker applications

master-worker application



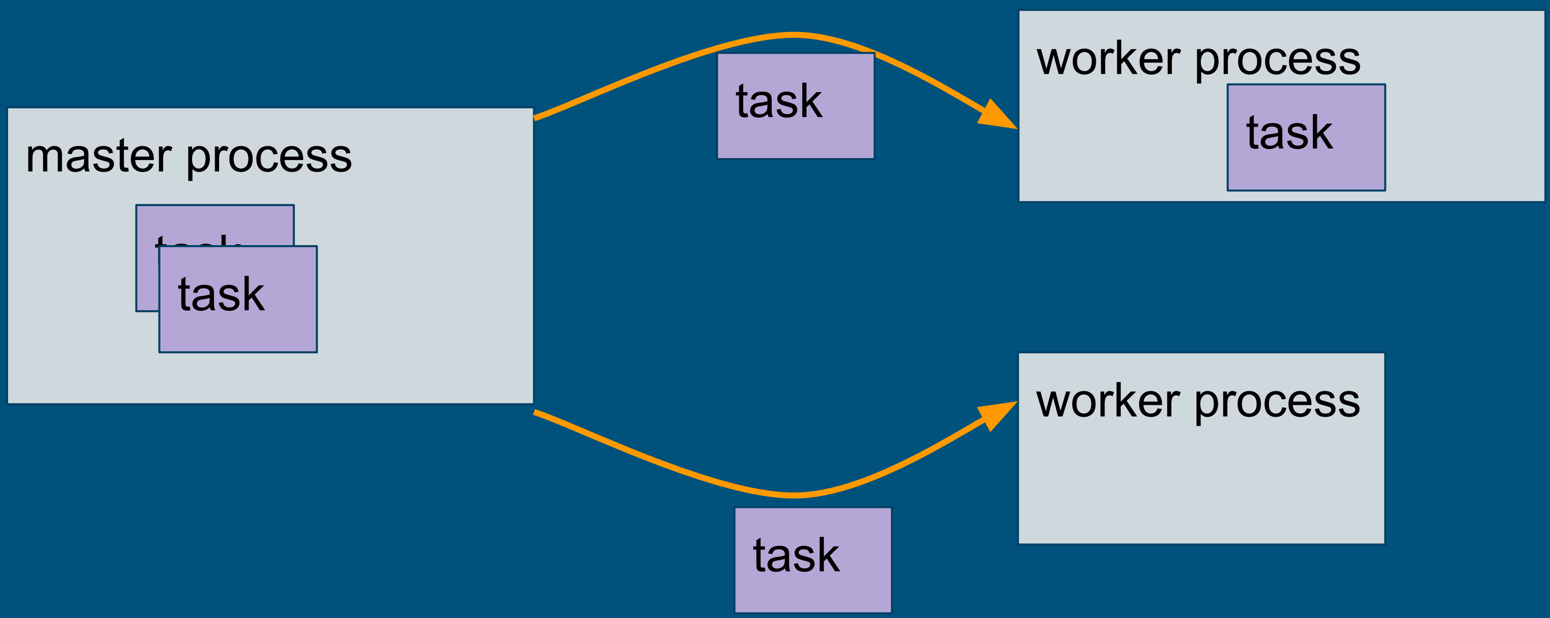
In a master worker application...

master-worker application



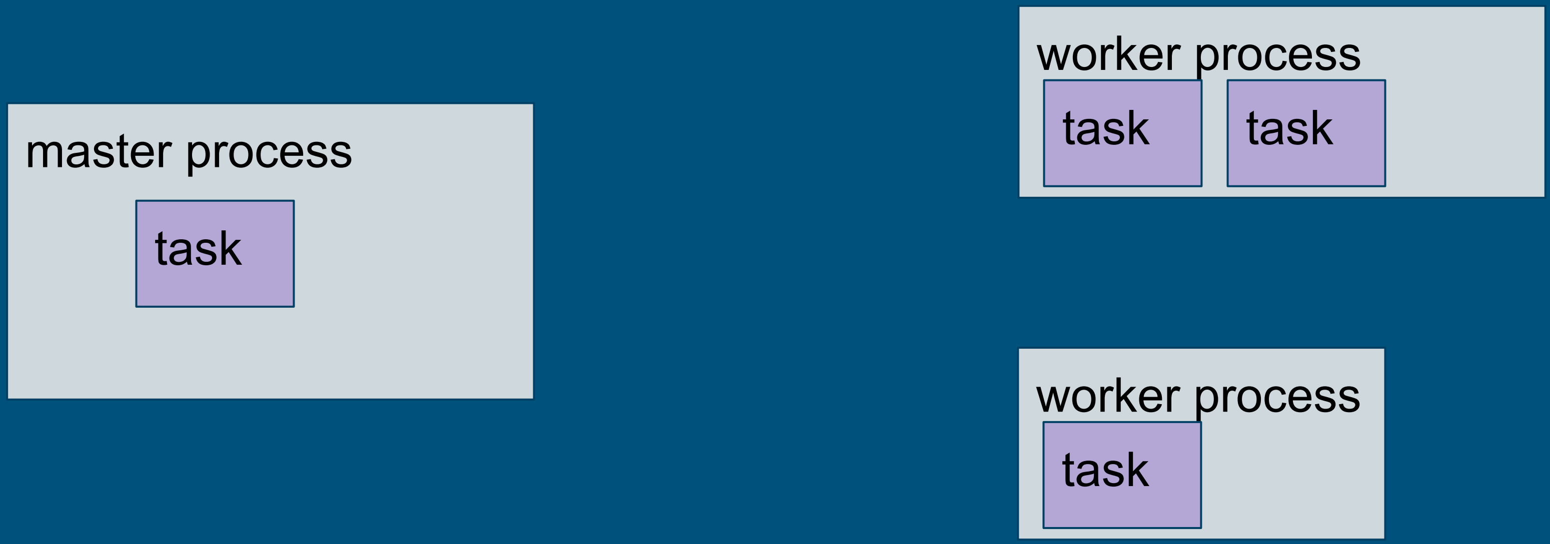
the master process generates tasks, puts them in a queue...

master-worker application



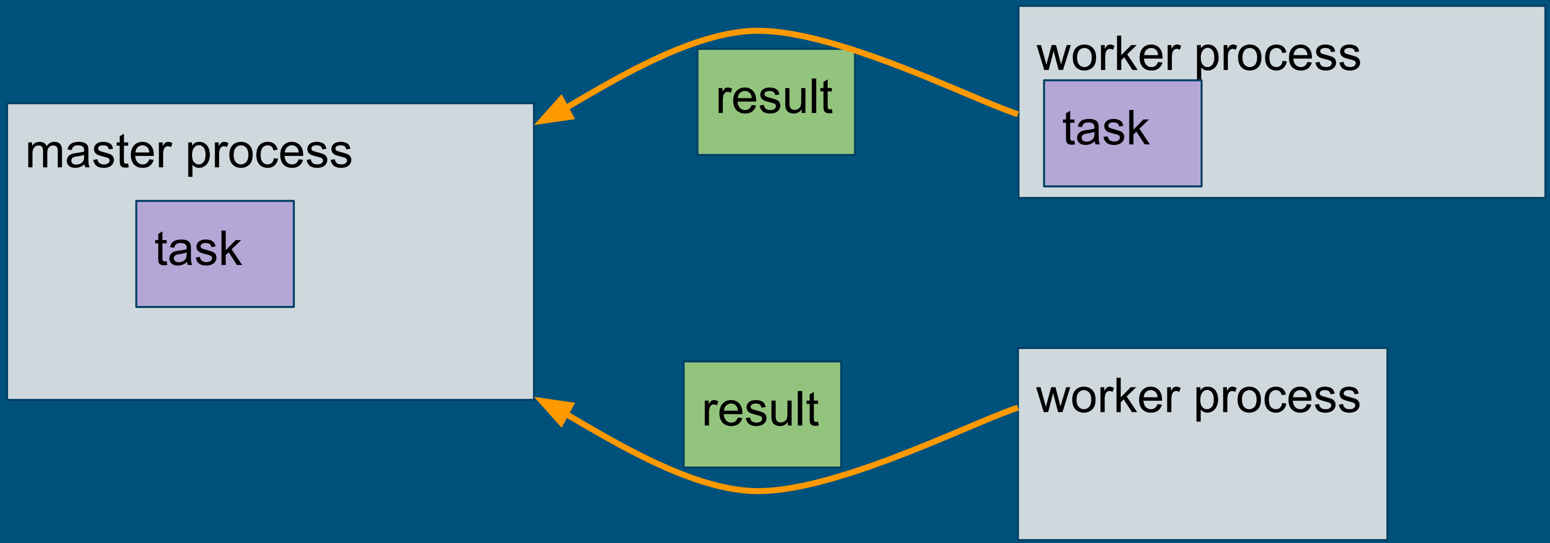
… delivers them to worker processes to execute...

master-worker application



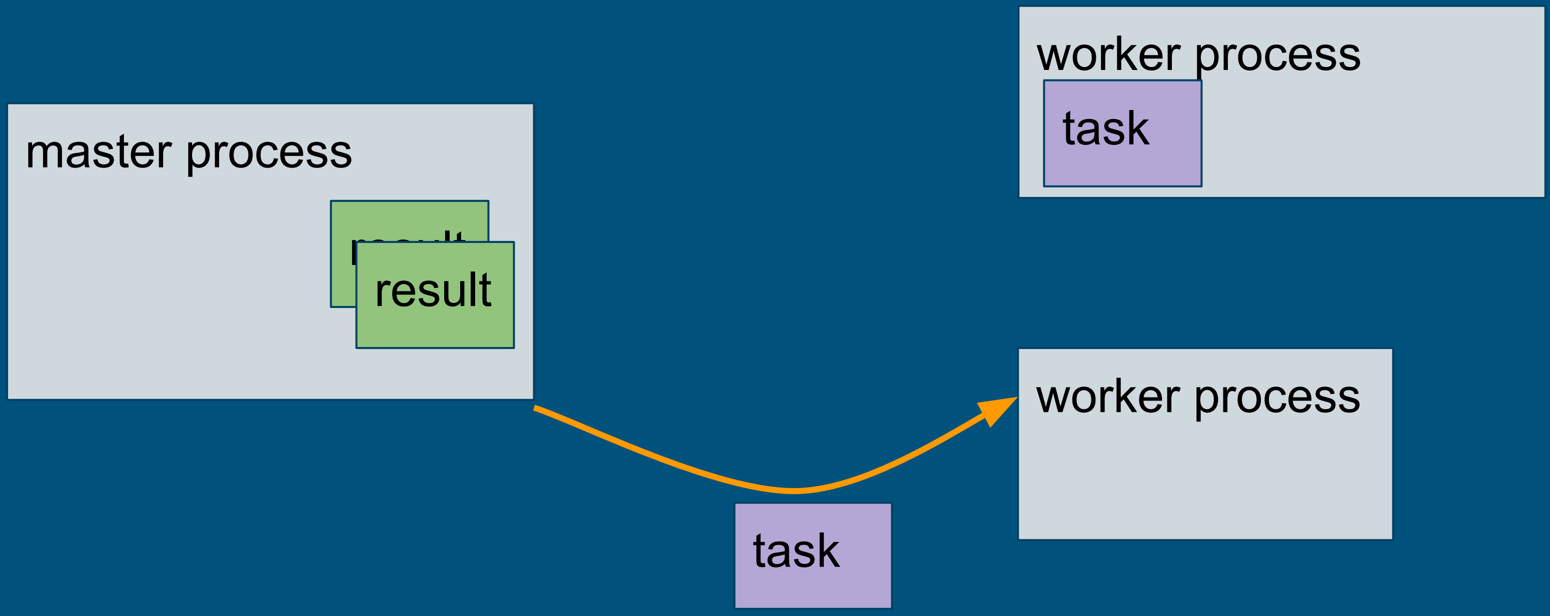
… waits for workers to execute tasks ...

master-worker application



and gathers the results on completion.

master-worker application



and on and on until no more tasks are generated.

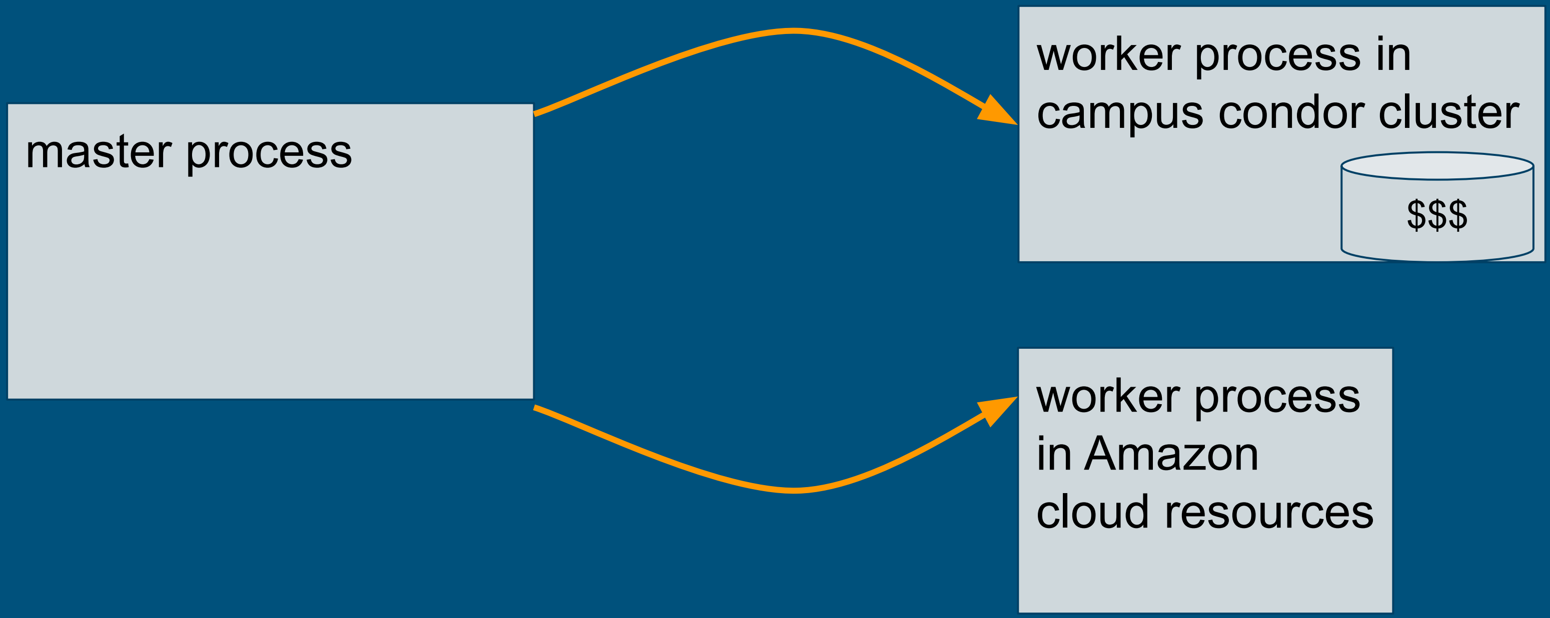
pure condor vs wq master-worker

one condor job per task vs. one condor job per worker

When is it most beneficial?

- Lots of small tasks:
  - Wait time in the condor queue proportional to the number of workers, not the number of tasks.
- Workers can cache common input files, reducing transfer times.
- Workers may run in any pool, or resource you have access (including non-condor resources).

master-worker application



pure condor vs wq master-worker

one condor job per task vs. one condor job per worker

When it is **not** beneficial?

- Tasks are not easily described in terms of input-outputs.
  - (e.g. streaming)

- You need to use an advanced feature of condor.

- You like to write highly customized condor submit files.

- The worker process interferes with your task. (Wrappers all the way down.)

writing master-worker applications with work queue

work queue when describing workflows

work queue:

- submit-wait programming model
- workflow structure can be decided at run time
- when a task is declared, it is assumed to be ready to run
- bindings in C, python2, python3, and perl

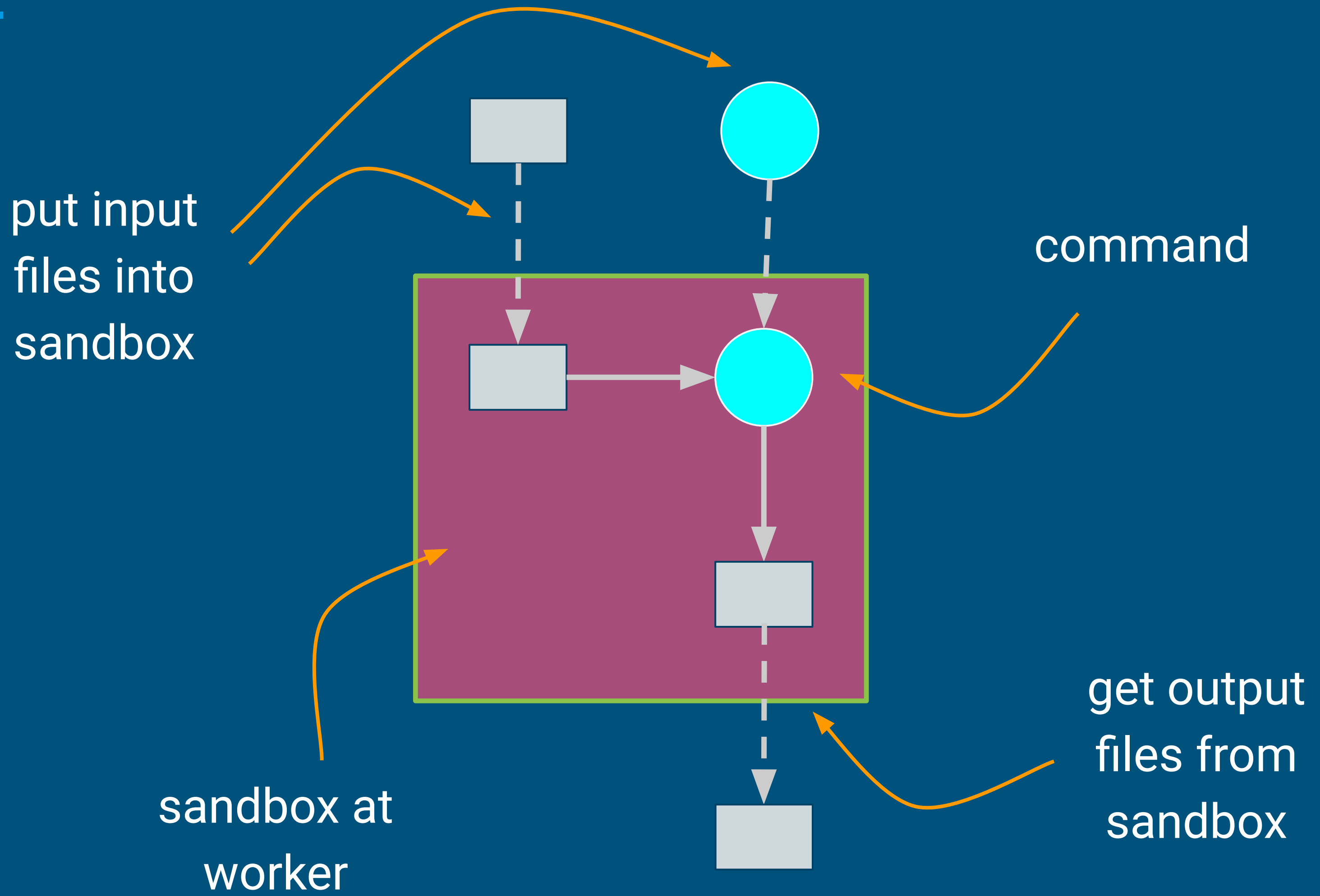
describing tasks

Consider a command '**sim.exe**', that takes input file **A**, and produces outfile **X**.

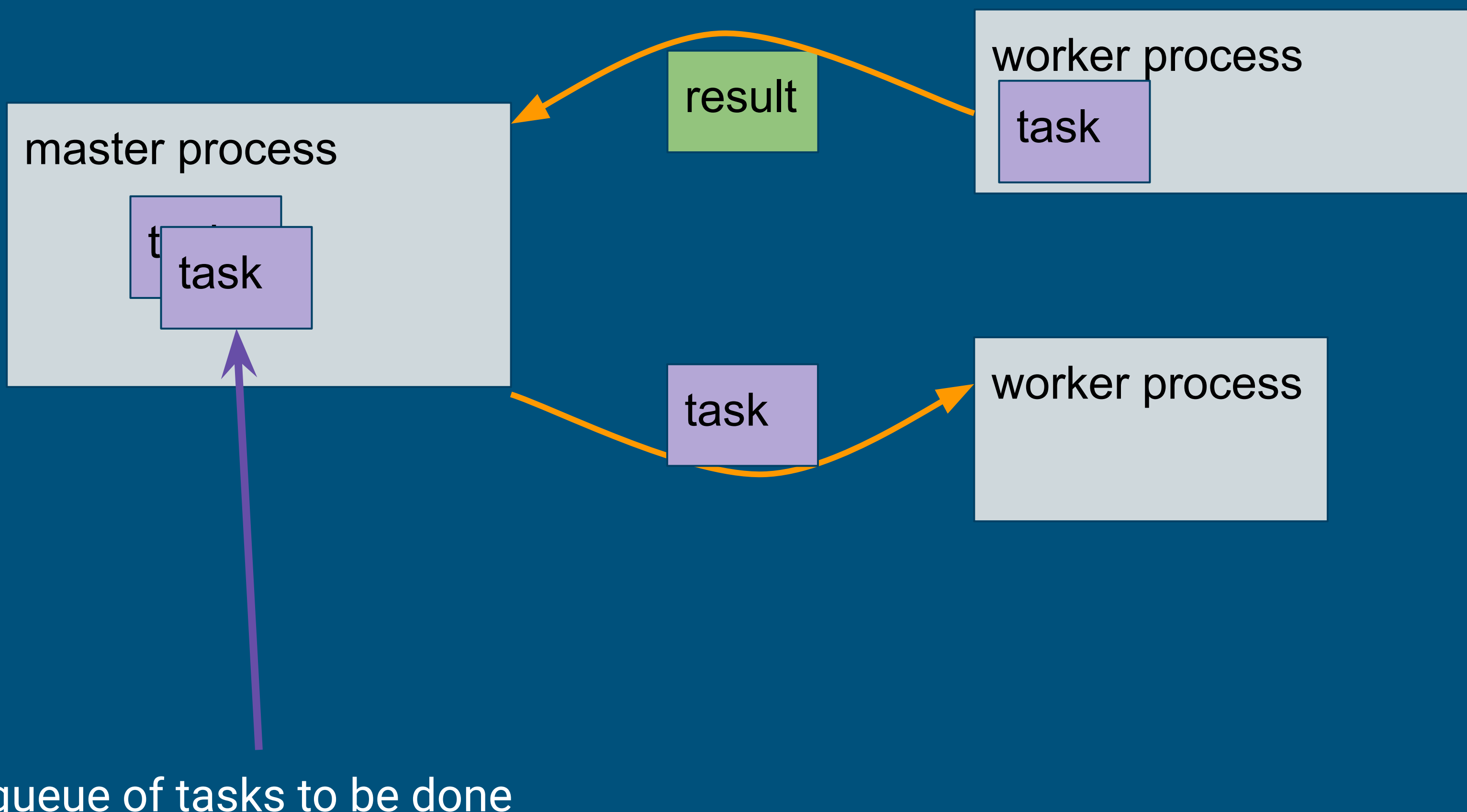
what is the set of input files?   what is the set of output files?

```
$ ls
sim.exe A

$ ./sim.exe A X

$ ls
sim.exe A X
```

Task Execution Model



master-worker application



skeleton of a work queue application

1. create and configure a queue
2. create and configure tasks
3. submit tasks to the queue
4. wait for tasks to complete
    a. if no new tasks to submit, terminate
    b. otherwise go to 2

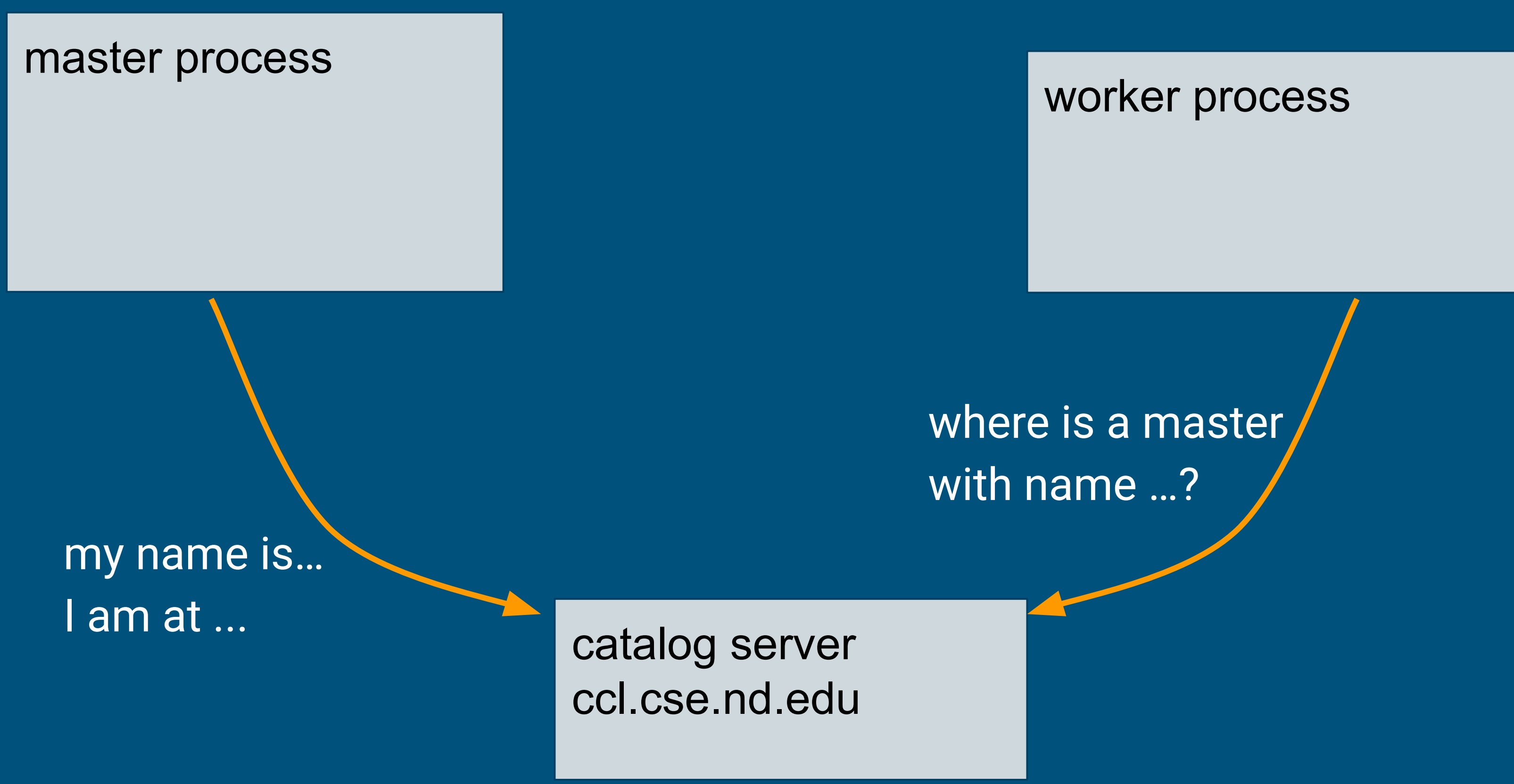
minimal work queue application

```
import work_queue as WQ

# 1. master named: 'my-master-name', run at some port at random
q = WQ.WorkQueue(name='my-master-name', port=0)

# 2. create a tasks that runs a command remotely, and ...
t = WQ.Task('./sim.exe A X')

# ...specify the name of input and output files
t.specify_input_file('sim.exe', cache=True)
t.specify_input_file('A')
t.specify_output_file('X')

# 3. submit the task to the queue
q.submit(t)

# 4. wait for all tasks to finish, 5 second timeout:
while not q.empty():
    t = q.wait(5)
    if t.result == WQ.WORK_QUEUE_RESULT_SUCCESS:
        print 'task {} finished'.format(t.id)
```

running work queue

```
$ python example_01.py

# in some other terminal, launch a worker for that master
# workers don't need PYTHONPATH set.
# -M my-master-name to serve masters with that name
# it could be a regexp.

# --single-shot to terminate after serving one master
# In general workers may serve many masters in their
# lifetime, but only one at a time.

$ work_queue_worker -M my-master-name --single-shot
```

how do workers find the master?



work_queue_status

```
cclws16  ~ > work_queue_status
PROJECT            HOST                   PORT WAITING RUNNING COMPLETE WORKERS
shadho-thermalize  45.55.219.221          9123     400       0    60438       0
reopt              chem9164.ucdavis.edu   7324      49      31       20      31
lobster_rgoldouz_w earth.crc.nd.edu       9000       2       0        0       1
lobster_rgoldouz_w earth.crc.nd.edu       9001       2       0        0       1
forcebalance       entropy.ucsd.edu      20001       9       0      117       0
forcebalance       entropy.ucsd.edu      20000       3       0      105       0
forcebalance       entropy.ucsd.edu      40002       4       0       59       0
forcebalance       entropy.ucsd.edu      30001       1       0       75       0
ForceBalance       entropy.ucsd.edu       6868       0       1    66516       1
forcebalance       entropy.ucsd.edu      40001       9       0        0       0
forcebalance       entropy.ucsd.edu      20009       9       0       72       0
forcebalance       entropy.ucsd.edu      20005       4       0      104       0
dihedral           libra.ucdavis.edu      8246      55       8     3346       8
my-master-name     submit2.chtc.wisc.edu 10010      10       0        0       0
forcebalance       tscc-gpu-9-2.sdsc.edu  7680       4       6        6       6
```

chosen master name

if the defaults don't work for you

Before launching the master, specify the range of ports available

default range is 9000-9999

at U. of Wisconsin you need:

```
export TCP_LOW_PORT=10000
export TCP_HIGH_PORT=10999
```

or: `WQ.WorkQueue(name='my-master-name', port=[10000,10999])`

Instead of -M:

use --port at the master to specify a port to listen
specify address of master and port at the worker

If you must, you can also run your own cctools/bin/catalog_server (-C option)

create a worker in condor

```
# using \ to break the command in multiple lines
# you can omit the \ and put everything in one line

# run 3 workers in condor, each of size 1 cores, 2048 MB
# of memory and 4096 MB of disk,
# to serve ${USER}-my-makeflow
# and which timeout after 60s of being idle.

$ condor_submit_worker  --cores 1              \
                        --memory 2048          \
                        --disk   4096          \
                        -M my-master-name      \
                        --timeout 60           \
                        3
```

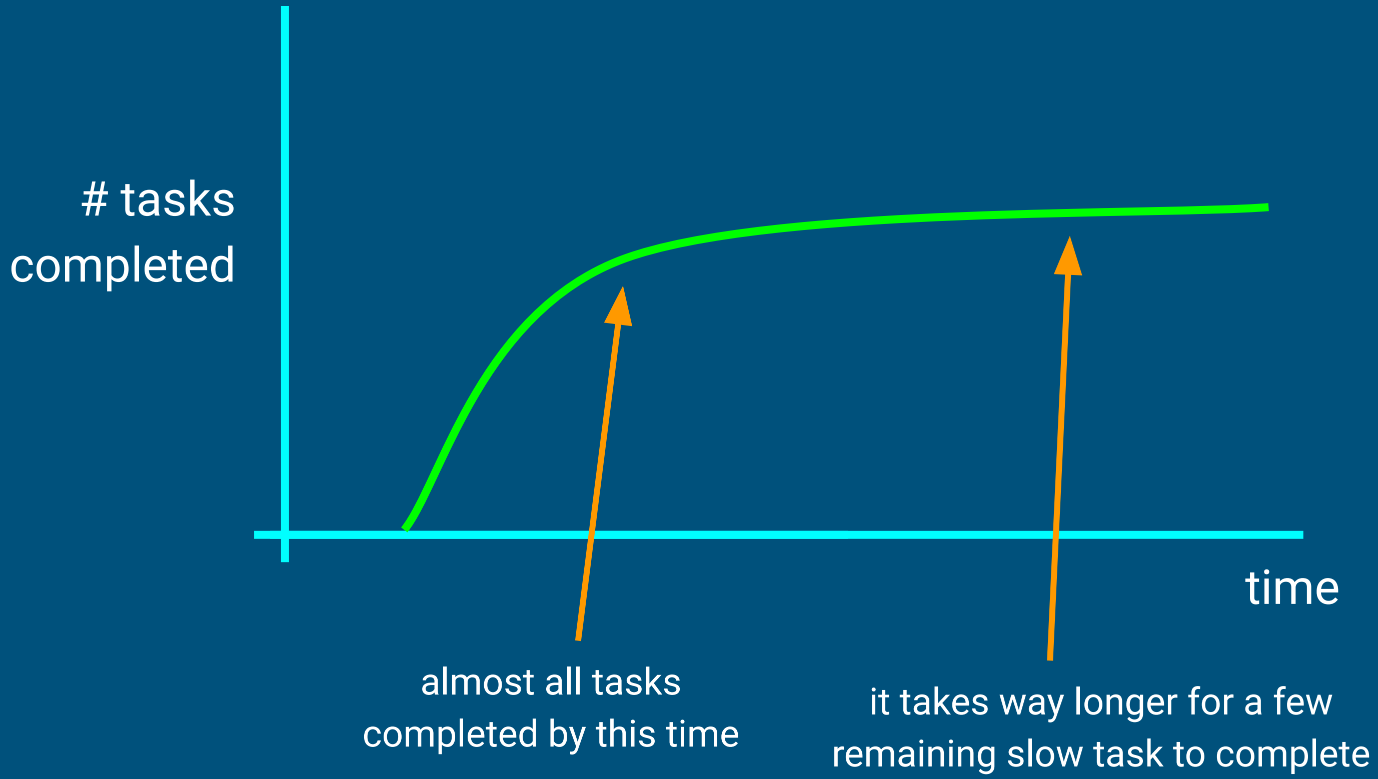
parameter sweep example

```
$ ls
my-cmd

# my-cmd takes the value of one parameter and produces an
# output file:
$ ./my-cmd   -parameter 1  -output out.1

$ ls
my-cmd out.1

# we want to try 1000 values of the parameter
$ ./my-cmd   -parameter 2   -output out.2
$ ./my-cmd   -parameter 3   -output out.3
$ ./my-cmd   -parameter 4   -output out.4
…
$ ./my-cmd   -parameter 1000  -output out.1000
```

parameter sweep example

```
from work_queue import WorkQueue, Task

# 1. create the queue
q = WorkQueue(name='my-parameter-sweep', port=0)


for i in range(1..1000):
    # 2. create a task
    t = Task('./my-cmd -output out.{1} -parameter {1}'.format(i))
    t.specify_input_file('cmd', cache=True)
    t.specify_output_file('out.{}'.format(i))
    t.specify_tag(str(i))  # arbitrary string to identify the task

    # 3. submit the task to the queue
    q.submit(t)

# 4. wait for all tasks to finish, 5 second timeout:
while not q.empty():
     t = q.wait(5)
     if t:
         if t.return_status == 0:
             with open('out.{}'.format(t.tag) as f:
                 ...
```

dealing with long tails
on opportunistic resources



if you suspect there are workers in slow machines

```
# strategy one:
# tell work queue to automatically shutdown slow workers:


q.activate_fast_abort(multiplier)


# if the average completion time of all tasks is AVG
# shutdown workers with average completion time
# larger than multiplier*AVG


# i.e: multiplier = (1 + n*STD_DEV)
```

if you suspect there are workers in slow machines

```
# strategy 2:
# submit the same task multiple times
# keep the result of the one that terminates the fastest.


t = Task(...)
t.specify_tag('some_identifying_tag')

for n in range(0..5):
    t_copy = t.clone()
    q.submit(t_copy)


while not q.empty():
    t_fastest = q.wait(5)
    if t_fastest:
        q.cancel_by_tasktag('some_identifying_tag')
        break
```

work queue
resource management

resources contract:
running several tasks in a worker concurrently

```
Worker has
available:

i cores
j MB of memory
k MB of disk
```

```
Task needs:



m cores
n MB of memory
o MB of disk
```

Task runs only if it fits in the currently available worker resources.

resources contract example

```
Worker has
available:

8 cores
512 MB of memory
512 MB of disk
```

```
Task a:

4 cores
100 MB of memory
100 MB of disk
```

```
Task b:

3 cores
100 MB of memory
100 MB of disk
```

Tasks a and b may run in worker at the same time.
(Work could still run another 1 core task.)

Beware!
tasks use all worker on missing declarations

```
Worker has
available:

8 cores
512 MB of memory
500 TB of disk
```

```
Task a:

4 cores
100 MB of memory
```

```
Task b:

3 cores
100 MB of memory
```

Tasks a and b may NOT run in worker at the same time.
(disk resource is not specified.)

specifying tasks resources

```
# categories are groups of tasks with the same
# resource requirements

# specify resources per category
q.specify_category_max_resources('my_category',
{
   'cores'  : 1,
   'memory': 1024,
   'disk'   : 1014
})

# assign the task to the category
t = Task('...')
t.specify_category('my_category')
```

managing resources

Do nothing (default if tasks don't declare cores, memory or disk):

One task per worker, task occupies the whole worker.

Honor contract (default if tasks declare resources):

Task declares cores, memory, and disk (the three of them!)

Worker runs as many concurrent tasks as they fit.

Tasks **may** use more resources than declared.

Monitoring and Enforcement:

Tasks fail (permanently) if they go above the resources declared.

Automatic resource labeling:

Tasks are retried with resources that maximize throughput, or minimize waste.

Monitoring and enforcement

Tasks fail (permanently) if they go above the resources declared.

```
q.enable_monitoring()

t = q.wait(...)

# resources assigned to the task
# .cores, .memory, .disk
t.resources_allocated.cores

# resource really used
t.resources_measured.memory

# which limit was broken?
if t.result == WORK_QUEUE_RESULT_RESOURCE_EXHAUSTION:
  if t.limits_exceeded.disk > -1:
       ...
```

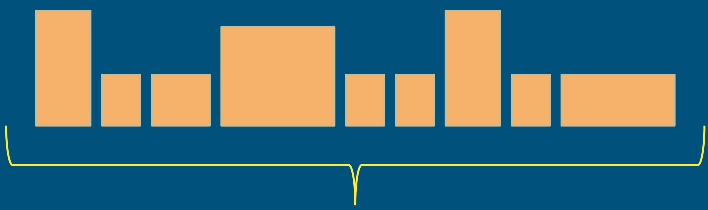
Monitoring and enforcement

Workers and tasks are matched using **only cores, memory, and disk**.

However, limits can be set and monitored in many other resources:

```
q.specify_category_max_resources('my_category',
{
   'cores': n,       'memory': MB,    'disk': MB,
   'wall_time': us, 'cpu_time': us,  'end': us,
   'swap_memory': MB,
   'bytes_read': B,          'bytes_written': B,
   'bytes_received': B,   'bytes_sent': B,
   'bandwidth': B/s
   'work_dir_num_files': n
… }
```

automatic resource labeling
when you don't know how big your tasks are

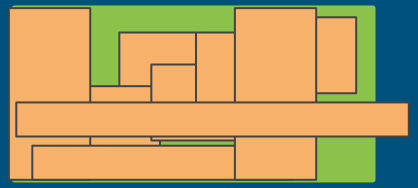


Tasks which size
(e.g., cores, memory, and disk)
is not known until runtime.

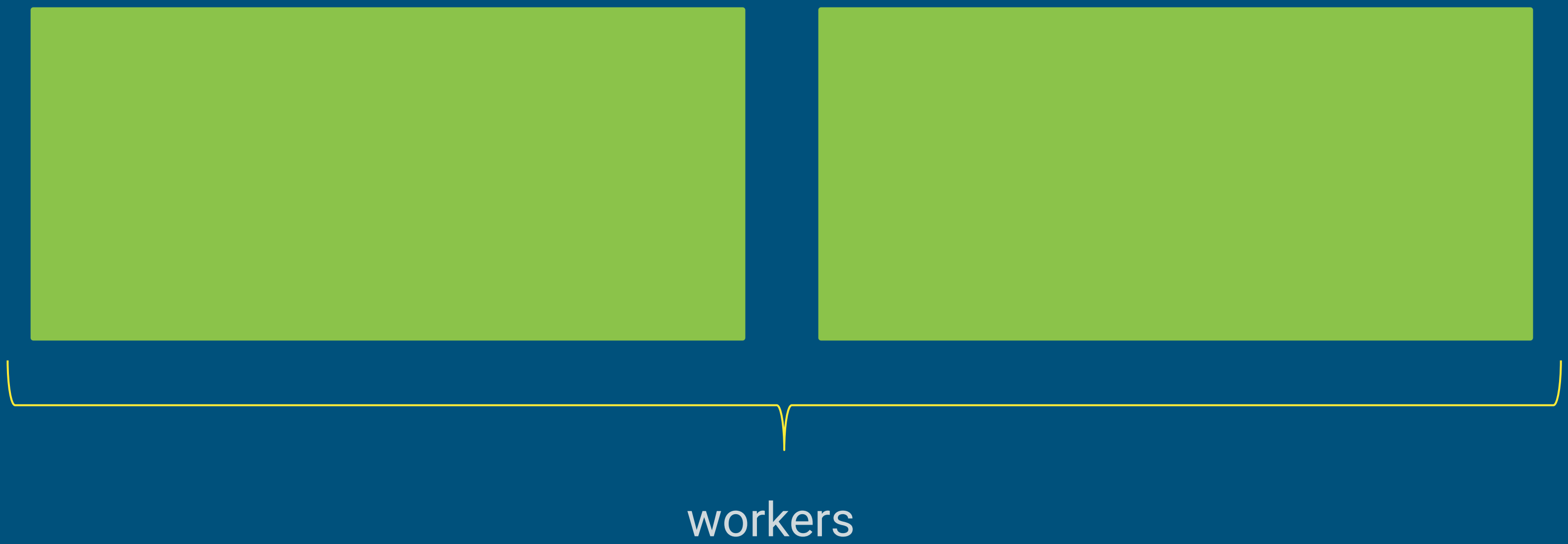
workers

One task per worker:
Wasted resources, reduced throughput.

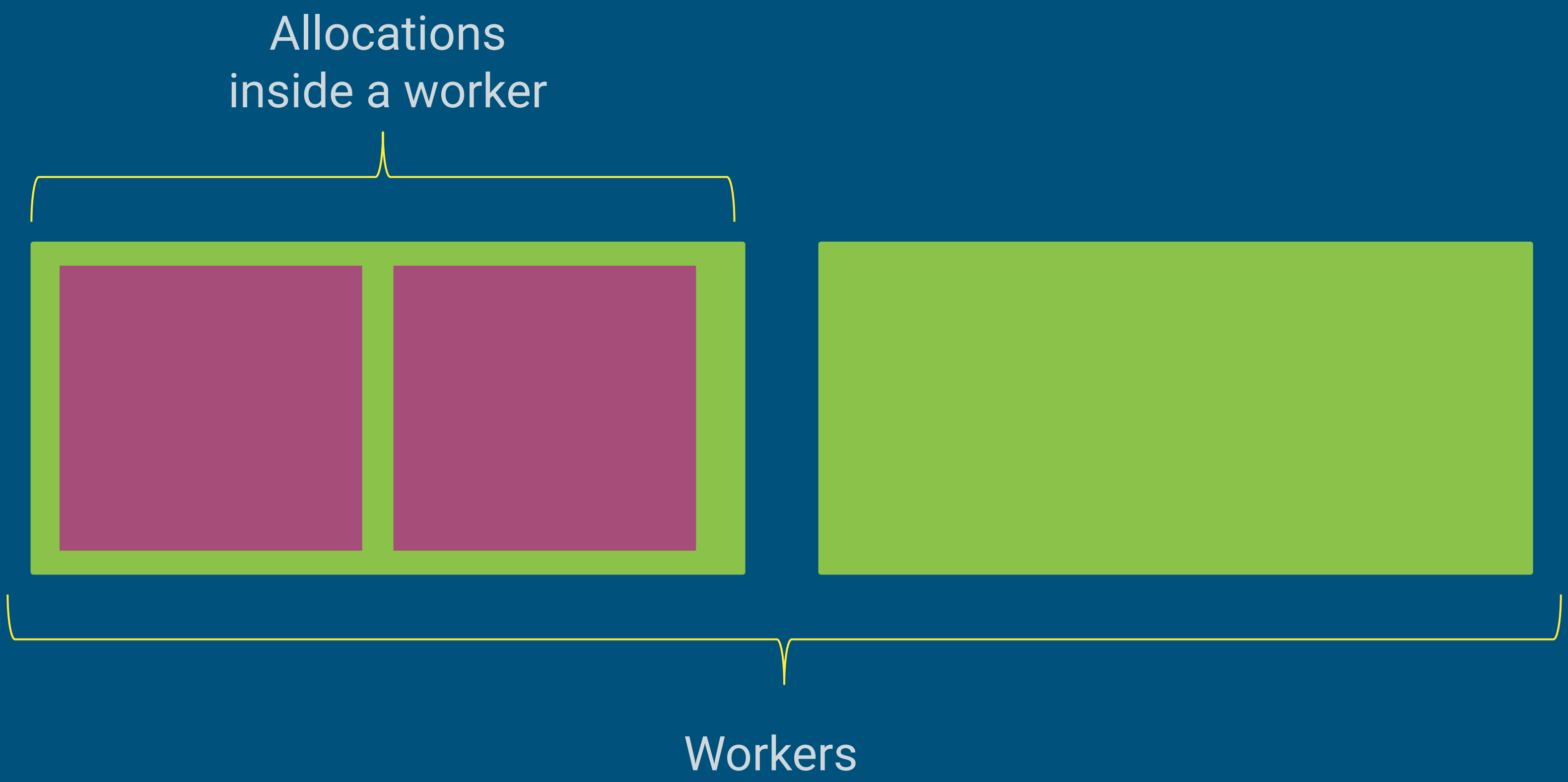
Many tasks per worker:
Resource contention/exhaustion, reduce throughput



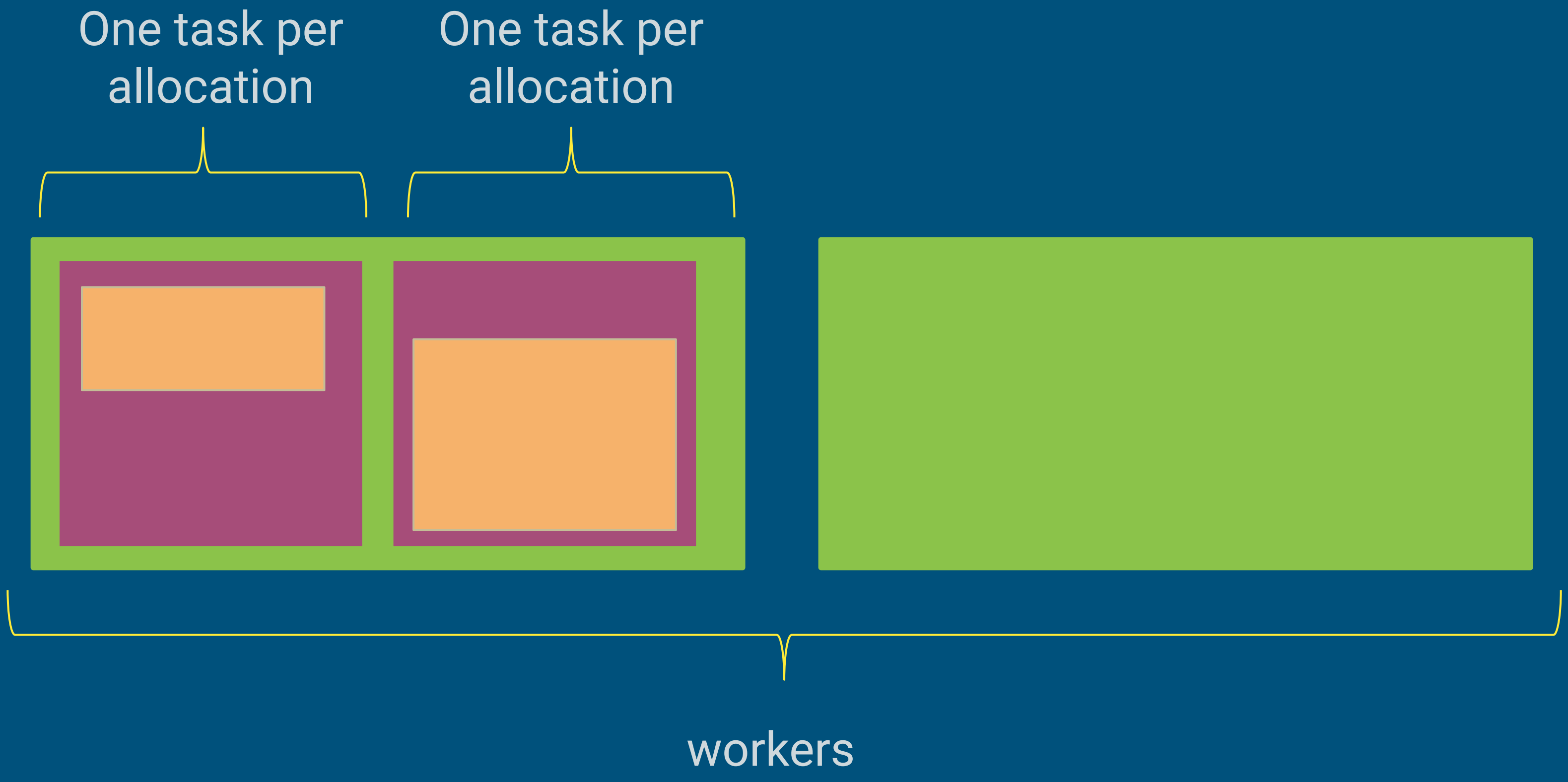
Task-in-the-Box



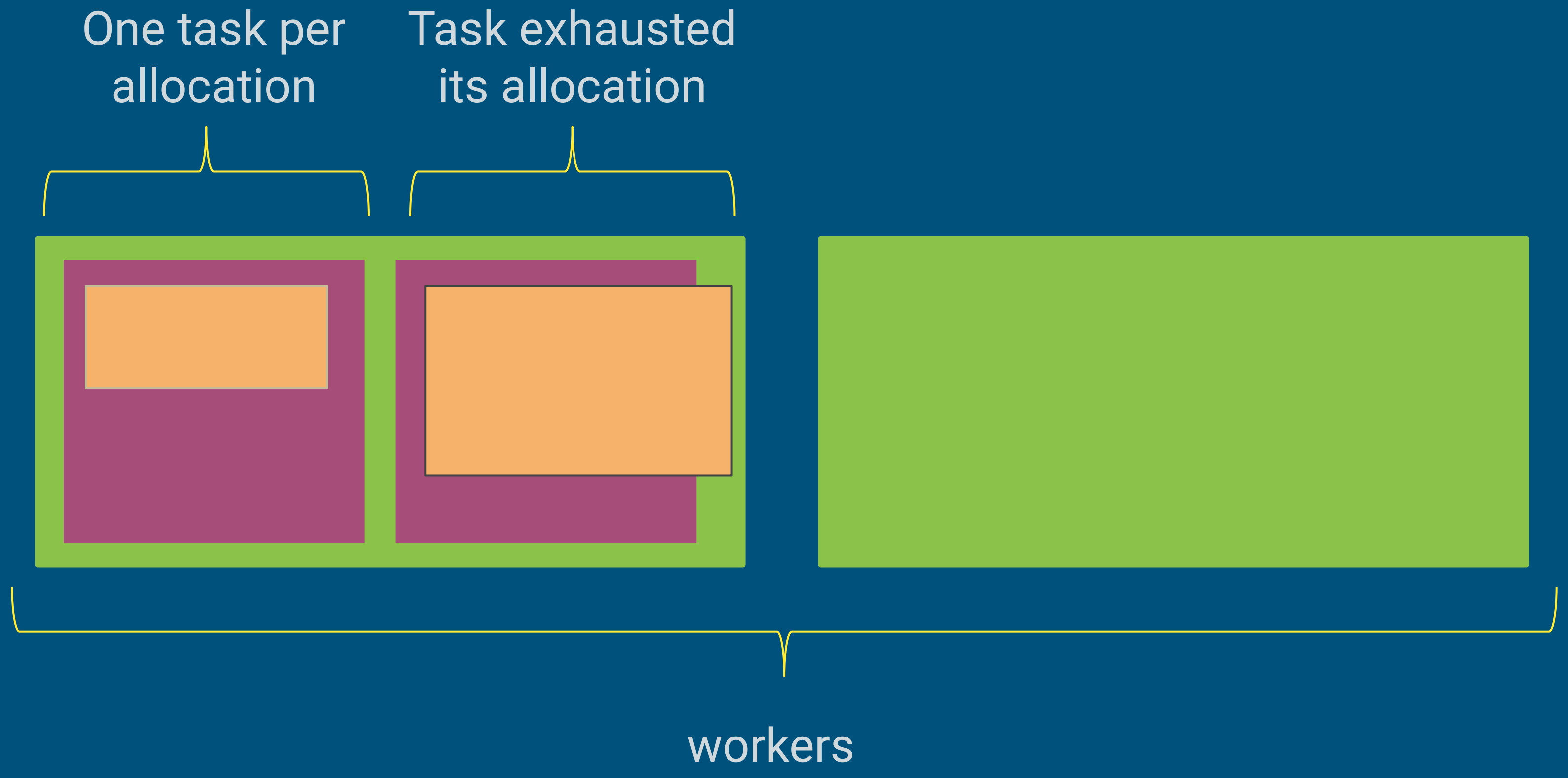
Task-in-the-Box



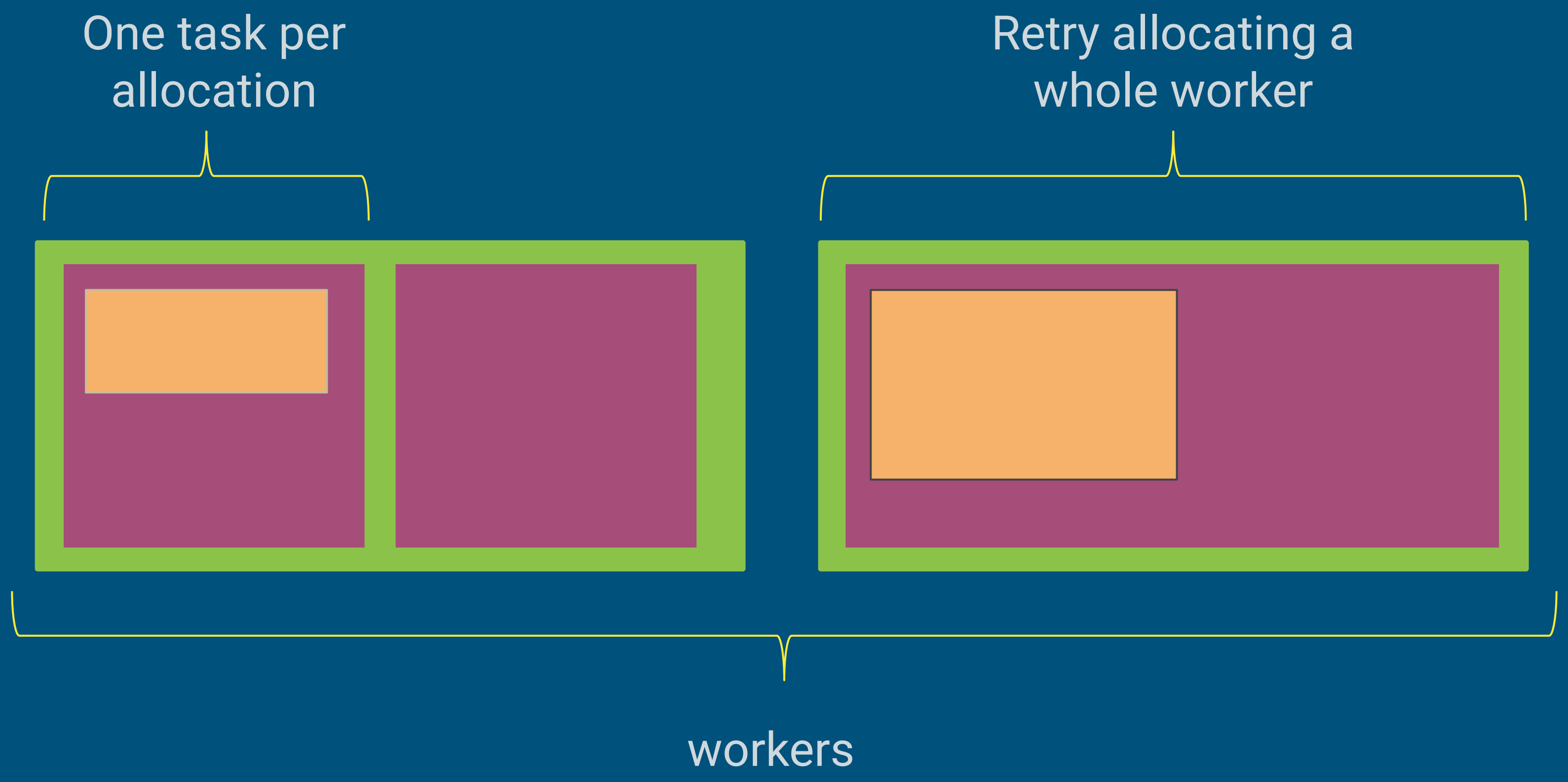
Task-in-the-Box



Task-in-the-Box



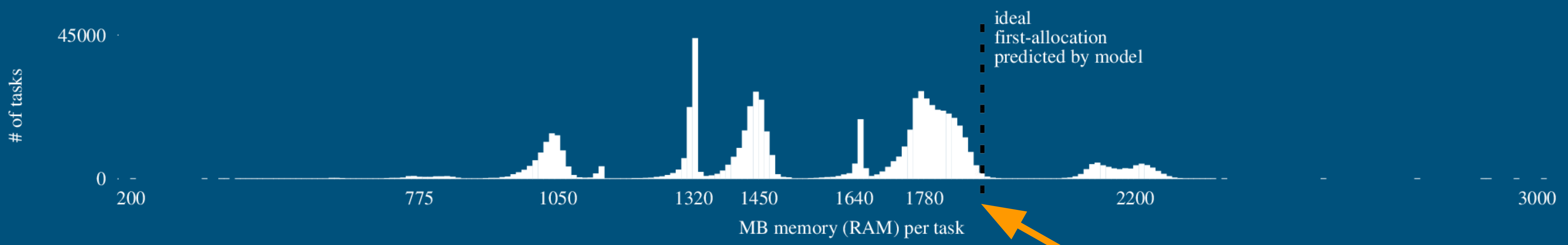
Task-in-the-Box



ND CMS example

Real result from a production High-Energy Physics CMS analysis (Lobster NDCMS)

Histogram Peak Memory vs Number of Tasks
O(700K) tasks that ran in O(26K) cores managed by WorkQueue/Condor.



First-allocation that maximizes expected throughput
(increase of %40 w.r.t. no task is retried)

Tovar, et.al
DOI:10.1109/TPDS.2017.2762310

automatic resource labeling

```
# compute retries for maximum throughput
q.specify_category_mode('my_category',
      work_queue.WORK_QUEUE_ALLOCATION_MODE_MAX_THROUGHPUT)

# compute retries for minimum waste
q.specify_category_mode('my_category',
      work_queue.WORK_QUEUE_ALLOCATION_MODE_MIN_WASTE)

# task fails at first resource exhaustion (default)
q.specify_category_mode('my_category',
      work_queue.WORK_QUEUE_ALLOCATION_MODE_FIXED)

# task is retried at bigger workers when available
q.specify_category_mode('my_category',
      work_queue.WORK_QUEUE_ALLOCATION_MODE_MAX)
```

when do task retries stop?

```
# an explicit hard limit is reached...
q.specify_category_max_resources('my_category', …)

# or maximum number of retries is reached:
# (default 1)
t.specify_max_retries(n)


# note that you can define categories for which
# no hard limit is reached, then only max retries
# is relevant.
```

what work queue does behind the scenes

1. Some tasks are run using full workers.
2. Statistics are collected.
3. Allocations computed to maximize throughput, or minimize waste.
   a. Run task using guessed size.
   b. If task exhausts guessed size, keep retrying on full (bigger) workers, or a specified `specify_category_max_resources` is reached.
4. When statistics become out-of-date, go to 1.

resources example

```
q.enable_monitoring()

# create a category for all tasks
q.specify_category_max_resources('my-tasks', {'cores': 1, 'disk': 500})
q.specify_category_mode('my-tasks',
WQ.WORK_QUEUE_ALLOCATION_MODE_MAX_THROUGHPUT)

# create 30 tasks. A task creates a 1000MB file, using 10MB of memory buffer.
for i in range(0,30):
    t = WQ.Task('python task.py 1000')
    t.specify_input_file('task.py', cache = True)
    t.specify_category('my-tasks')
    t.specify_max_retries(2)
    q.submit(t)

# create a task that will break the limits set
t = WQ.Task('python task.py 1000')
t.specify_input_file('task.py', cache = True)
t.specify_category('my-tasks')
t.specify_max_retries(2)
q.submit(t)

while not q.empty():
    t = q.wait(60)
    ...
```

resources example

```
$ source ~/cctools-tutorial/etc/uofwm-env
$ cd ~/cctools-tutorial/example_02
$ python example_02.py
…
WorkQueue on port: NNNN

# in another terminal, create a worker:
# (-dall -o:stdout to send debug output to stdout)
$ work_queue_worker -M ${USER}-master --disk 2000 -dall  -o:stdout |
grep 'Limit'
… cctools-monitor[8837] error: Limit disk broken.

# ^C to kill the worker
# check resources statistics
$ work_queue_status -A localhost NNNN
CATEGORY RUNNING WAITING FIT-WORKERS MAX-CORES MAX-MEMORY MAX-DISK
my-tasks       0      50            0         1        ~10     >500
```

work_queue_status -A HOST PORT

information about waiting tasks and resources

CATEGORY	RUNNING	WAITING	FIT-WORKERS	MAX-CORES	MAX-MEM	MAX-DISK
my-cat-a	2	20	2	1	~1024	~2000
my-cat-b	0	15	0	1	>3000	~1000
my-cat-c	0	0	0	???	???	???

fixed resource

No info on tasks waiting.

no fixed resource set, and all tasks have run under this value

> At least one task that is now waiting, failed exhausting these much of the resource.

other work queue capabilities

the work queue factory

Factory creates workers as needed by the master:

```
$ work_queue_factory -Tcondor \
 -M some-master-name
 --min-workers 5
 --max-workers 200
 --cores 1 --memory 4096 --disk 10000
 --tasks-per-worker 4
```

the work queue factory -- conf file

to make adjustments the configuration file can be modified once the factory is running

```
$ work_queue_factory -Tcondor -C my-conf.json
$ cat my-conf.json
{
    "master-name": "some-master-name",
    "max-workers": 200,
    "min-workers": 5,
    "workers-per-cycle": 5,
    "cores":  1,
    "disk":   10000,
    "memory": 4096,
    "timeout":   900,
    "tasks-per-worker": 4
}
```

using condor docker universe

```
# launch three workers to serve my-master-name
# workers will run inside docker-image-name

$ condor_submit_workers
          -M my-master-name                     \
          --docker-universe  docker-image-name  \
          3
```

configuring runtime logs

We recommend to always enable all the logs.

```
import work_queue as WQ

# record of the states of tasks and workers
# specially useful when tracking tasks resource
# usage and retries
q.specify_transactions_log('my_transactions.log')


# workers joined, tasks completed, etc. per time step
q.specify_log('my_stats.log')
```

transactions log

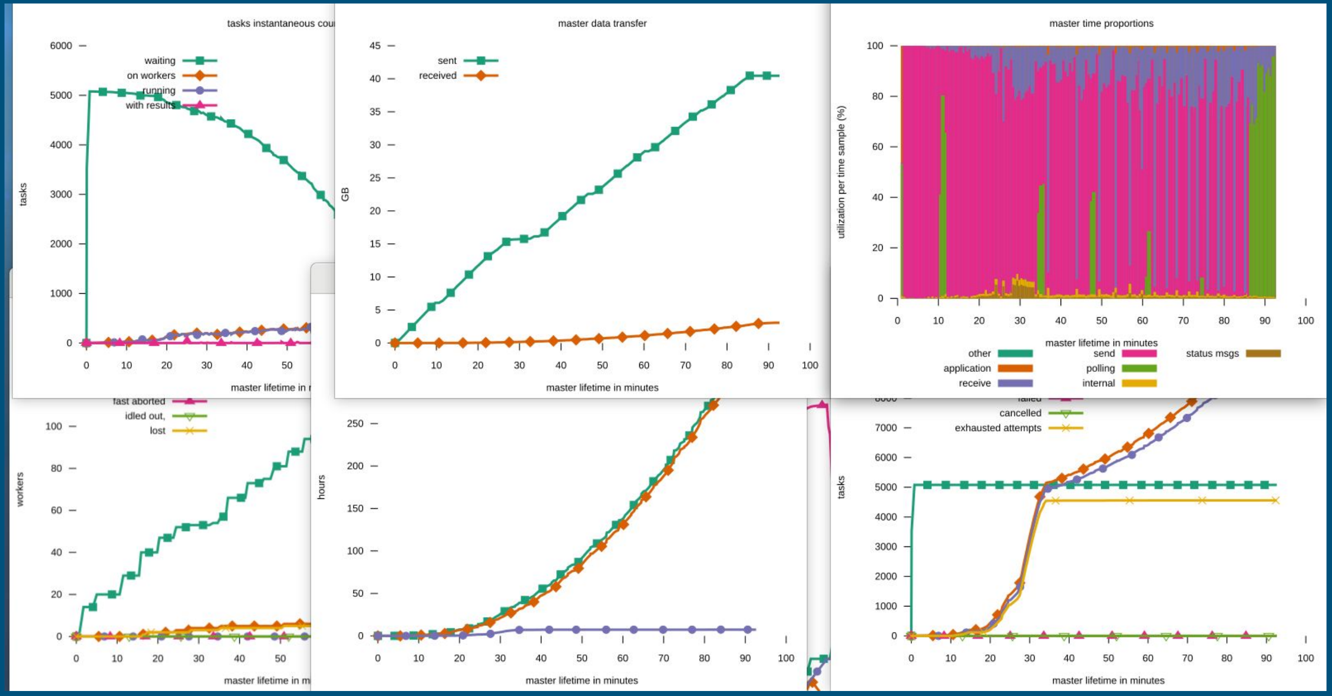
```
$ grep '\<TASK 1\>' example_02.tr
```

```
1550697985850270 9374 TASK 1 WAITING my-tasks FIRST_RESOURCES {"cores":[1,"cores"]}
1550698004105770 9374 TASK 1 RUNNING 127.0.0.1:40730  FIRST_RESOURCES {"cores":[1,"cores"],"memory
,"MB"]}
1550698004473367 9374 TASK 1 WAITING_RETRIEVAL 127.0.0.1:40730
1550698004475215 9374 TASK 1 RETRIEVED RESOURCE_EXHAUSTION {"disk":[20,"MB"]} {"start":[155069800
698004259680,"us"],"cores_avg":[0.989,"cores"],"cores":[1,"cores"],"wall_time":[0.14619,"s"],"cpu_
x_concurrent_processes":[1,"procs"],"total_processes":[1,"procs"],"memory":[1,"MB"],"virtual_memo
":[0,"MB"],"bytes_read":[0.00138569,"MB"],"bytes_written":[0,"MB"],"bytes_received":[0,"MB"],"byt
dth":[0,"Mbps"],"total_files":[7,"files"],"disk":[201,"MB"],"machine_cpus":[8,"cores"],"machine_l
1550698004475384 9374 TASK 1 WAITING my-tasks MAX_RESOURCES {"cores":[1,"cores"],"memory":[1,"MB"
1550698046053626 9374 TASK 1 RUNNING 127.0.0.1:40734  MAX_RESOURCES {"cores":[1,"cores"],"memory"
"MB"]}
1550698046444043 9374 TASK 1 WAITING_RETRIEVAL 127.0.0.1:40734
1550698046445440 9374 TASK 1 RETRIEVED SUCCESS {"start":[1550698046079981,"us"],"end":[155069804622
[0.989,"cores"],"cores":[1,"cores"],"wall_time":[0.146097,"s"],"cpu_time":[0.144457,"s"],"max_con
ocs"],"total_processes":[1,"procs"],"memory":[1,"MB"],"virtual_memory":[6,"MB"],"swap_memory":[0,
38569,"MB"],"bytes_written":[0,"MB"],"bytes_received":[0,"MB"],"bytes_sent":[0,"MB"],"bandwidth":
[7,"files"],"disk":[201,"MB"],"machine_cpus":[8,"cores"],"machine_load":[0.31,"procs"]}
1550698046445762 9374 TASK 1 DONE SUCCESS {"start":[1550698046079981,"us"],"end":[155069804622607
9,"cores"],"cores":[1,"cores"],"wall_time":[0.146097,"s"],"cpu_time":[0.144457,"s"],"max_concurre
,"total_processes":[1,"procs"],"memory":[1,"MB"],"virtual_memory":[6,"MB"],"swap_memory":[0,"MB"]
,"MB"],"bytes_written":[0,"MB"],"bytes_received":[0,"MB"],"bytes_sent":[0,"MB"],"bandwidth":[0,"M
iles"],"disk":[201,"MB"],"machine_cpus":[8,"cores"],"machine_load":[0.31,"procs"]}
```

statistics log

Use work_queue_graph_log to visualize the statistics log:

```
$ work_queue_graph_log my_stats.log
$ display my_stats.*.svn
```



other ways to access statistics

```
$ work_queue_status -l HOST PORT
{"name":"cclws16.cse.nd.edu","address":"129.74.153.171","tasks_total_disk":0,...
```

```
# current stats counts (e.g., q.stats.workers_idle)
s = q.stats
s = q.stats_by_category('my-category'))

# available stats
# http://ccl.cse.nd.edu/software/manuals/api/html/structwork__queue__stats.html
```

all workers can talk to all masters, unless...

```
# put a passphrase in a text file, say my.password.txt

# tell master to use the password:
q.specify_password('my.password.txt')

# tell workers to use the password:
$ work_queue_worker … --password my.password.txt
```

other miscellaneous work queue calls

```
# blacklist a worker
q.blacklist(hostname)

# remove cached file from workers
q.invalidate_cache_file(filename)

# remote name of files
q.specify_{in|out}put_file(name-at-master, name-at-worker,...)

# if directory name, send/receive recursively
t.specify_directory('some/dir',
                      recursive=True,
                      type=work_queue.WORK_QUEUE_INPUT)
                      # or type=work_queue.WORK_QUEUE_OUTPUT)

# produce monitoring snapshots at certain events
(e.g., a regexp in a log appears, or a file is created/deleted)
t.specify_snapshot_file('snapshot-spec.json')

# resources per task
t.specify_cores(n)
t.specify_memory(n)
t.specify_disk(n)
```

Work Queue API

http://ccl.cse.nd.edu/software/manuals/api/python

http://ccl.cse.nd.edu/software/manuals/api/perl

http://ccl.cse.nd.edu/software/manuals/api/C

setting up cctools

getting the examples

```
$ ssh submit-1.chtc.wisc.edu
$ cd ~
$ git clone \
https://github.com/cooperative-computing-lab/cctools-tutorial
```

setting up cctools at U of Wisconsin M.

```
# in your ~/.bashrc file:
cctools_home=/usr/local/cctools
PATH=${cctools_home}/bin:${PATH}
PYTHONPATH=${cctools_home}/lib/python2.7/site-packages:${PYTHONPATH}
PERL5LIB=${cctools_home}/lib/perl5/site_perl/5.16.3:${PERL5LIB}

TCP_LOW_PORT=10000
TCP_HIGHT_PORT=10999
export PATH PYTHONPATH PERL5LIB TCP_LOW_PORT TCP_HIGH_PORT
```

installing up cctools anywhere else

```
$ wget
http://ccl.cse.nd.edu/software/files/cctools-7.0.13-x86_64
-centos7.tar.gz

# decompress cctools
$ tar -xf cctools-*-redhat7.tar.gz

# move to canonical destination
$ mv cctools-*-redhat7 cctools

# setup environment (you may want to add these
# lines to the end of .bashrc)
$ export PATH=:~/cctools/bin:${PATH}
# … etc… for PYTHONPATH and others
```

from source

```
$ $ wget
http://ccl.cse.nd.edu/software/files/cctools-7.0.13-source
.tar.gz

# decompress cctools
$ tar -xf cctools-*-src.tar.gz

# configure and install (swig dependency)
$ cd cctools-*-src
$ ./configure --with-swig-path=/path/to/swig
$ make
$ make install
```

test your setup

```
# if the following command fails, did you set PATH?
$ work_queue_worker --version
work_queue_worker version  7.0.13 FINAL from source (released 2019-05-14 09:42:11 -0400)
     Built by btovar@camd04.crc.nd.edu on 2019-05-14 09:42:11 -0400
     System: Linux camd04.crc.nd.edu 3.10.0-957.el7.x86_64 #1 SMP Thu Oct 4 20:48:51 UTC 2018
x86_64 x86_64 x86_64 GNU/Linux
     Configuration: --strict --build-label from source --build-date  --tcp-low-port 9000
--sge-parameter -pe smp $cores --strict --with-cvmfs-path /opt/libcvmfs --with-uuid-path /opt/uuid
--prefix /var/condor/execute/dir_2578/cctools-fb72a868-x86_64-centos7
```

thanks!

questions:
btovar@nd.edu

forum:
https://ccl.cse.nd.edu/community/forum

manuals:
http://ccl.cse.nd.edu/software

repositories:
https://github.com/cooperative-computing-lab/cctools
https://github.com/cooperative-computing-lab/makeflow-examples

This work was supported by:

NSF grant ACI 1642609
"SI2-SSE: Scaling up Science on Cyberinfrastructure with the Cooperative Computing Tools"

DOE grant ER26110
"dV/dt - Accelerating the Rate of Progress Towards Extreme Scale Collaborative Science"

UNIVERSITY OF NOTRE DAME

CCTools

extra slides

Stand-alone resource monitoring

```
resource_monitor -L"cores: 4" -L"memory: 4096" -- cmd
```

```
cclws16  ~ > resource_monitor -i1 -Omon --no-pprint -- /bin/date
Thu May 12 20:27:21 EDT 2016
cclws16  ~ > cat mon.summary
```


```
{"executable_type":"dynamic","monitor_version":"6.0.0.9edd8e96","host":"cclws16.cse.nd.edu
","command":"/bin/date","exit_status":0,"exit_type":"normal","start":[1463099241605723,"us
"],"end":[1463099243000239,"us"],"wall_time":[1.39452,"s"],"cpu_time":[0.002999,"s"],"core
s":[1,"cores"],"max_concurrent_processes":[1,"procs"],"total_processes":[1,"procs"],"memor
y":[1,"MB"],"virtual_memory":[107,"MB"],"swap_memory":[0,"MB"],"bytes_read":[0.0105429,"MB
"],"bytes_written":[0,"MB"],"bytes_received":[0,"MB"],"bytes_sent":[0,"MB"],"bandwidth":[0
,"Mbps"],"total_files":[90546,"files"],"disk":[11659,"MB"],"peak_times":{"units":"s","cpu_
time":1.39452,"cores":0.394445,"max_concurrent_processes":0.394445,"memory":0.394445,"virt
ual_memory":1.39428,"bytes_read":1.39428,"total_files":1.39428,"disk":1.39428}}%
```


http://ccl.cse.nd.edu/software/manuals/resource_monitor.html

(does not work as well on static executables that fork)

configuring tasks

```
from work_queue import Task

t = Task('shell command to be executed')

t.specify_input_file('path/to/some/file')

# files can be cached at workers
t.specify_input_file('path/to/other/file', cache=True)

# same for output files
t.specify_output_file('path/to/output/file')
t.specify_output_file('path/to/other/output', cache=True)

# if directory name, send/receive recursively
t.specify_directory('some/dir',
                    recursive=True,
                    type=work_queue.WORK_QUEUE_INPUT)
                    # or type=work_queue.WORK_QUEUE_OUTPUT)
```