

slides at:
<https://ntrda.me/2GAiswY>

master-worker applications with makeflow and work queue

Tim Shaffer, Nate Kremer-Herman, Nick Hazekamp,
and Ben Tovar

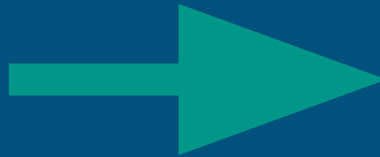


where we are

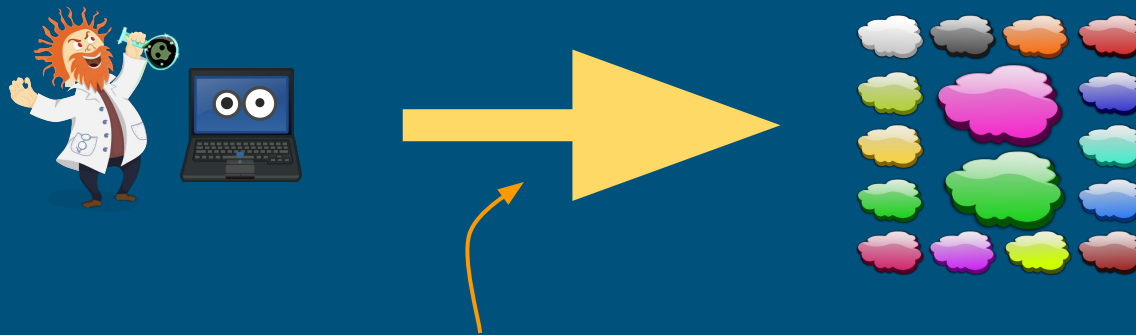


Scientist says:

"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



Douglas Thain
Director



Benjamin Tovar
**Research
Soft. Engineer**



Nicholas Hazekamp



Charles Zheng



Nate Kremer-Herman



Tim Shaffer



Andrew Litteken



Doug Smith

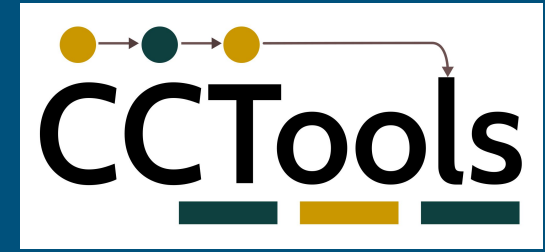


Herman Tong



Daniel Galvao Guerra

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

Setting-up CCTools
5min

Executing workflows with Makeflow (Nate)
20 min

Makeflow as a master-worker application (Nick)
25 min

Break

Writing master-worker applications with work queue (Ben)
with emphasis in resource management
50 min

setting up cctools

getting the examples (to try at home)

```
$ 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.

```
$ cd ~/cctools-tutorial
$ source etc/uofwm-env

# OR manually set the following:
cctools_home=/usr/local/cctools
PATH=${cctools_home}/bin:${PATH}
PYTHONPATH=${cctools_home}/lib/python2.7/site-packages:${PYTHONPATH}
TCP_LOW_PORT=10000
TCP_HIGH_PORT=10999
export PATH PYTHONPATH TCP_LOW_PORT TCP_HIGH_PORT

# See extra slides at end of presentation for manual
# install/setup in your personal machines
```

test your setup (to try at home)

```
# if the following command fails, did you set PATH?  
$ work_queue_worker --version  
work_queue_worker version 7.0.9 FINAL from source (released 2018-11-14 08:13:17 -0500)  
  Built by btovar@camd03.crc.nd.edu on 2018-11-14 08:13:17 -0500  
  System: Linux camd03.crc.nd.edu 3.10.0-862.el7.x86_64 #1 SMP Wed Mar 21 18:14:51 EDT 2018  
x86_64 x86_64 x8  
6_64 GNU/Linux  
  Configuration: --strict --build-label from source --build-date ...etc
```

makeflow



makeflow

A portable workflow manager.

makeflow

A portable workflow manager.

set of interdependent
computational tasks



makeflow

figure out which outputs are
inputs to some other task.
run tasks in order, as parallel as
possible.

A portable workflow manager.

set of interdependent
computational tasks



The diagram consists of two orange curved arrows. One arrow originates from the text 'set of interdependent computational tasks' and points towards the text 'A portable workflow manager.'. The other arrow originates from the text 'possible.' in the top right and points towards the text 'A portable workflow manager.'.

makeflow

runs tasks on:

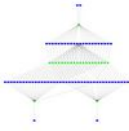
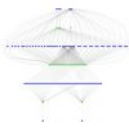
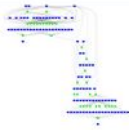

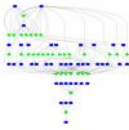

condor
slurm
sge
workqueue
ec2
mesos
...

A portable workflow manager.

set of interdependent
computational tasks

figure out which outputs are
inputs to some other task.
run tasks in order, as parallel as
possible.

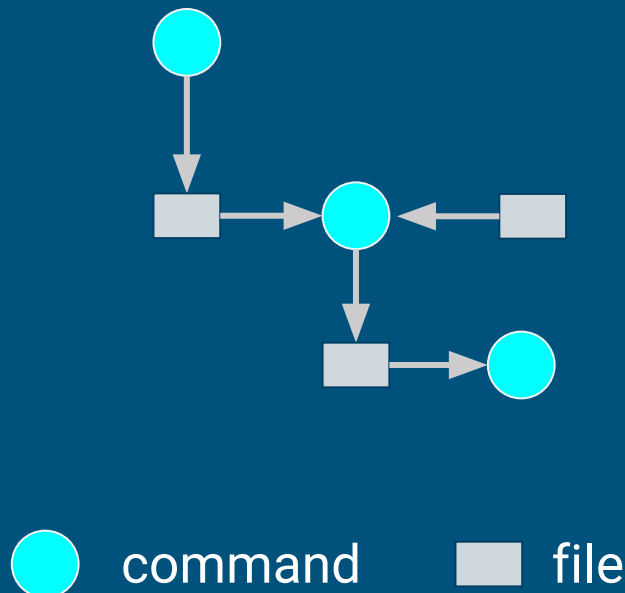
makeflow examples repository

	BLAST workflow adapted from the Biocompute web portal. (Shown at a scale of 10 splits.)
	SSAHA genomics analysis workflow, courtesy of Scott Emrich and Notre Dame Bioinformatics Laboratory. (Shown at scale of 25 splits.)
	BWA genomics analysis workflow, courtesy of Scott Emrich and Notre Dame Bioinformatics Laboratory. (Shown at scale of 20 splits.)
	SHRIMP genomics analysis workflow adapted from the Biocompute web portal. (Shown at a scale of 100 splits.)
	HECIL genomics analysis workflow, courtesy of Olivia Choudhury and Connor Howington.
	Lifemapper Species Distribution Modeling (SDM) workflow, courtesy of C.J. Grady. (Shown at scale of 10 species and 5 random trials.)
	SNPEXP Genomics analysis workflow courtesy of Scott Emrich and Notre Dame Bioinformatics Laboratory.
	BWA-GATK genomics workflow by Nick Hazekamp and Olivia Choudhury.

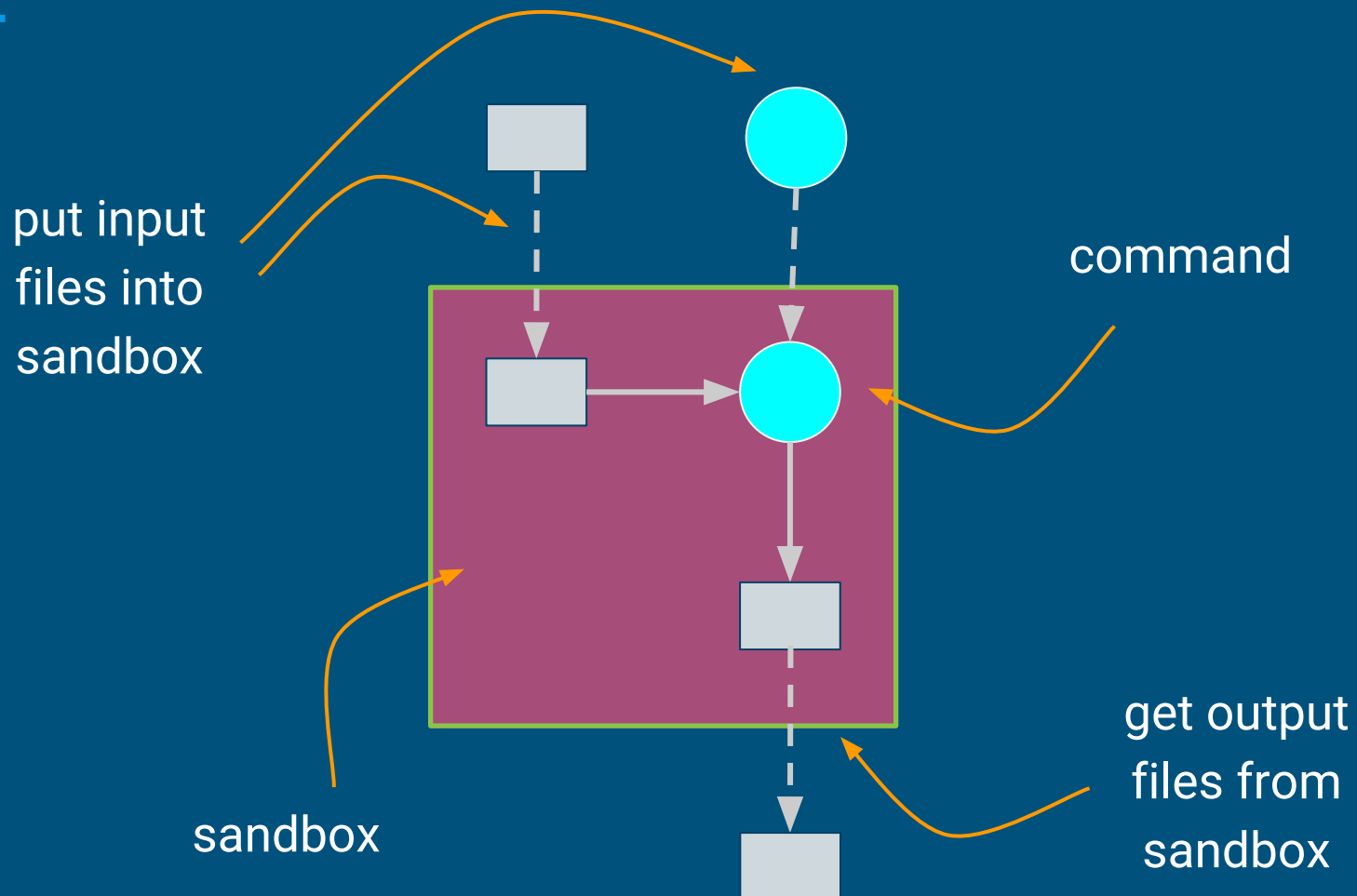
<https://github.com/cooperative-computing-lab/makeflow-examples>

dependencies in Makeflow

- Task X depends on Task Y if Task X produces a file Task Y needs.
- Directed acyclic graph (DAG) where nodes are tasks, and edges represent an input-output file dependency.

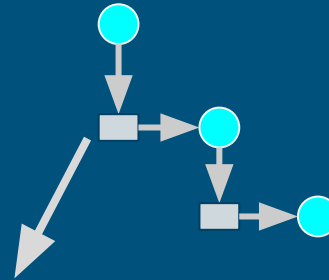


Task Execution Model

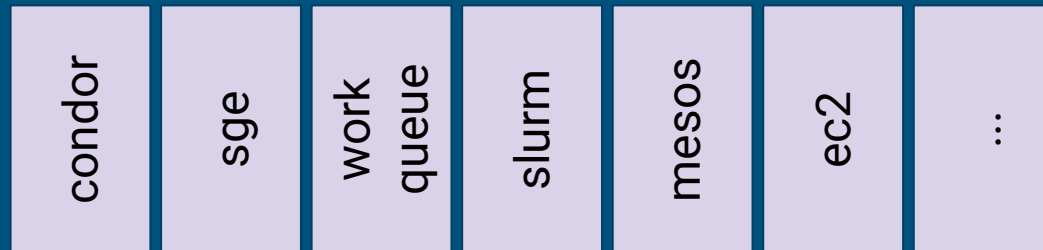
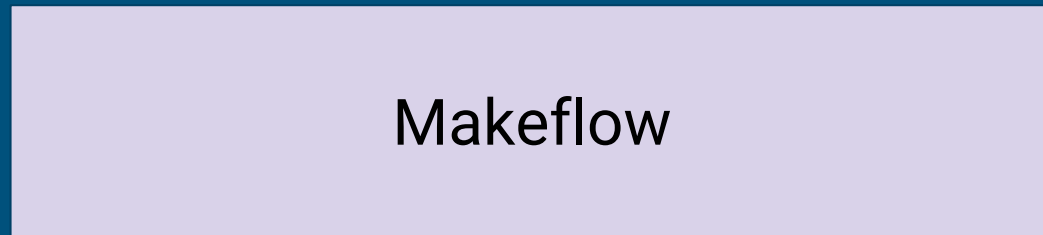


Makeflow Architecture

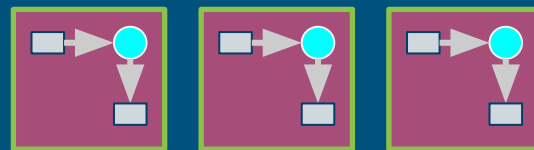
input is an abstract
directed acyclic graph



stats and
state kept by
transactions
logs



drivers



sandboxes

wrappers
that modify
how the
sandboxes
are created



describing a task in makeflow

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 some-input-file

$ ./sim.exe some-input-file some-output-file
$ ls
sim.exe some-input-file some-output-file
```

describing a task in makeflow

list of
outputs

colon :

list of
inputs

```
some-output-file: some-input-file sim.exe  
./sim.exe some-input-file some-output-file
```

tab

command from
the sandbox
perspective

makeflow example

```
# comments start with a '#'  
C: B sim.exe  
    ./sim.exe B C  
  
B: A sim.exe  
    ./sim.exe A B
```

what is this workflow doing?

are the rules in the wrong order?

makeflow mini example (try at home)

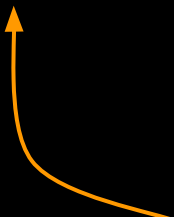
```
$ cd ~/cctools-tutorial/makeflow/example_01
$ ls
A  example_01.mf  sim.exe

$ makeflow example_01.mf
...

$ ls
A  B  C  example_01.mf  example_01.mf.makeflowlog  sim.exe
```

rerunning a workflow (try at home)

```
$ makeflow example_01
...
recovering from log file example_01.mf.makeflowlog...
starting workflow....
nothing left to do.
```

 transactions and
recovery log

```
$ makeflow -c example_01
$ ls
A example_01.mf sim.exe
```

declaring resources needed

my-simulations

Task 1:

4 cores
1024 MB of memory
1000 MB of disk

Task 2:

4 cores
1024 MB of memory
1000 MB of disk

my-postprocess

Task 3:

1 cores
2048 MB of memory
2000 MB of disk

declaring resources for tasks

Tasks are grouped into **categories**.

All tasks in a category have identical requirements for cores, memory and disk.

Unless specified otherwise, all tasks belong to the "**default**" category.

```
# memory and disk in MBs

.MAKEFLOW CATEGORY my-simulations
.MAKEFLOW CORES 4
.MAKEFLOW MEMORY 1024
.MAKEFLOW DISK 1000

.MAKEFLOW CATEGORY my-postprocess
.MAKEFLOW CORES 1
.MAKEFLOW MEMORY 512
.MAKEFLOW DISK 2000

.MAKEFLOW CATEGORY my-simulations
Y: X sim.exe
    ./sim.exe X Y
B: A sim.exe
    ./sim.exe A B

.MAKEFLOW CATEGORY my-postprocess
results: B Y postprocess
    ./postprocess B Y > results
```

running tasks on remote resources (try at home)

```
$ cd ~/cctools-tutorial/makeflow/example_02
```

```
# Run on condor with -Tcondor
```

```
# size of slots requested as appropriate
```

```
$ makeflow example_02.mf -Tcondor
```

```
...
```

```
# Confirm with the condor log the resource allocations:
```

```
$ grep -B1 -A2 Cpus example_02.mf.condorlog
```

```
...
```

Partitionable Resources :	Usage	Request	Allocated
Cpus :		1	1
Disk (KB) :	21	2048000	3530078
Memory (MB) :	0	512	512

```
...
```

when things go wrong (try at home)

```
$ cd ~/cctools-tutorial/makeflow/example_03
```

```
# broken_sim.exe does not work
```

```
$ ./broken_sim.exe A B
```

```
trying step 1 of ./broken_sim.exe A B
```

```
trying step 2 of ./broken_sim.exe A B
```

```
trying step 3 of ./broken_sim.exe A B
```

```
an error! oh no!
```

when things go wrong (try at home)

```
$ cat example_03.mf
B log-of-A-to-B: A broken_sim.exe
  ./broken_sim.exe A B > log-of-A-to-B

# -r      N to retry a failed rule N times (default 10)
# -dall   To print debug info to stdout
$ makeflow -r2 -dall example_03.mf
...

# partial outputs available per task
$ cat makeflow.failed.0/log-of-A-to-B
```

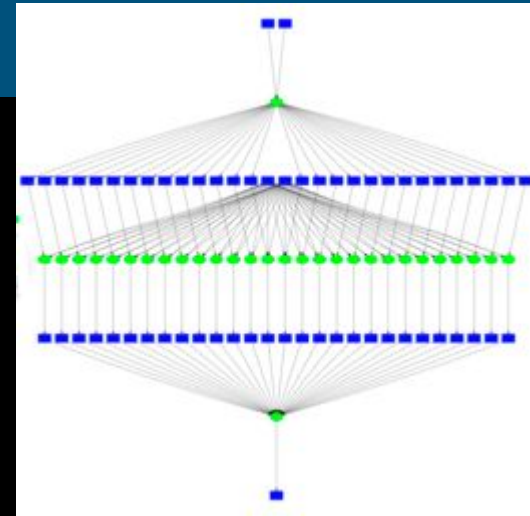

common pattern: execute large tasks locally

```
split1 split2 ... splitn: my-large-file  
LOCAL ./split-my-files my-large-file
```

```
output1: split1  
    ./process split1 > output1
```

...

```
my-large-output: output1 output2 ... outputn  
LOCAL ./merge output1 output2 ... > my-large-output
```



makeflow containers tutorial

<http://ccl.cse.nd.edu/software/tutorials/makeflow/container-tutorial.ph>

p

Makeflow Container Tutorial

For this tutorial, we will assume you have access to the XSEDE platform, specifically the Open Science Grid (OSG). If you do not, please speak with your campus champion or get in touch with someone at [XSEDE](#). This tutorial should be read only after completing the [Makeflow tutorial](#).

Downloading the Singularity Image

Login to the XSEDE single sign-on portal login.xsede.org using `ssh`, PuTTY, or a similar tool. Then, login to the Open Science Grid by running:

```
gsissh osg
```

Once you have a shell, we will enter the tutorial directory used in the [Makeflow tutorial](#):

```
cd $HOME  
cd tutorial
```

We will now pull the Singularity image we will use for this tutorial:

```
singularity pull docker://nekelluna/ccl_makeflow_examples  
WARNING: pull for Docker Hub is not guaranteed to produce the  
WARNING: same image on repeated pull. Use Singularity Registry  
WARNING: (shub://) to pull exactly equivalent images.  
Docker image path: index.docker.io/nekelluna/ccl_makeflow_examples:latest  
Cache folder set to XXX/.singularity/docker  
Importing: base Singularity environment  
Exploding layer: sha256:22dc81ace0ea2f45ad67b790cddad29a45e206d51db0af826dc9495ba21a0b06.tar.gz  
Exploding layer: sha256:1a8b3c87dba3ed16956c881a26eb0c40502c176a35767627d87557d16eale0df.tar.gz  
Exploding layer: sha256:91390a1c435a20661a9e9afdaeb818638299a20d6ee1cc06bbcab8ae4d51994f.tar.gz
```

JX - JSON-like alternative makeflow language

Workflows can also be described in pure JSON or in an extended language known as JX which can be evaluated to produce pure JSON.

JX is more flexible than the Unix Make syntax (E.g., it has functions, and operators.)

<http://ccl.cse.nd.edu/software/manuals/jx-tutorial.html>

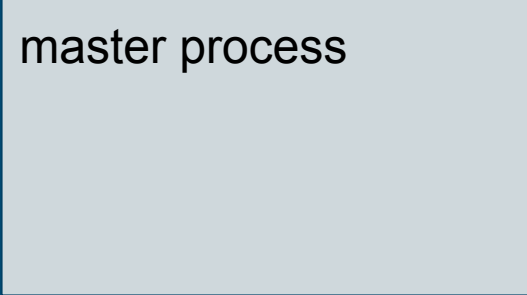
```
# Generate an array of 100 files named "output.1.txt", etc...
[ "output."+x+".txt" for x in range(1,100) ]

# Generate one string containing those 100 file names.
join( [ "output."+x+".txt" for x in range(1,100) ], " ")

# Generate five jobs that produce output files alpha.txt, beta.txt, ...
{
  "command" : "simulate.py > "name+".txt",
  "outputs" : [ name+".txt" ],
  "inputs" : "simulate.py",
} for name in [ "alpha", "beta", "gamma", "delta", "epsilon" ]
```

makeflow as a master-worker application

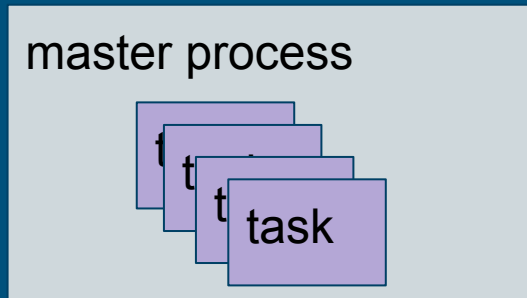
master-worker application



master process

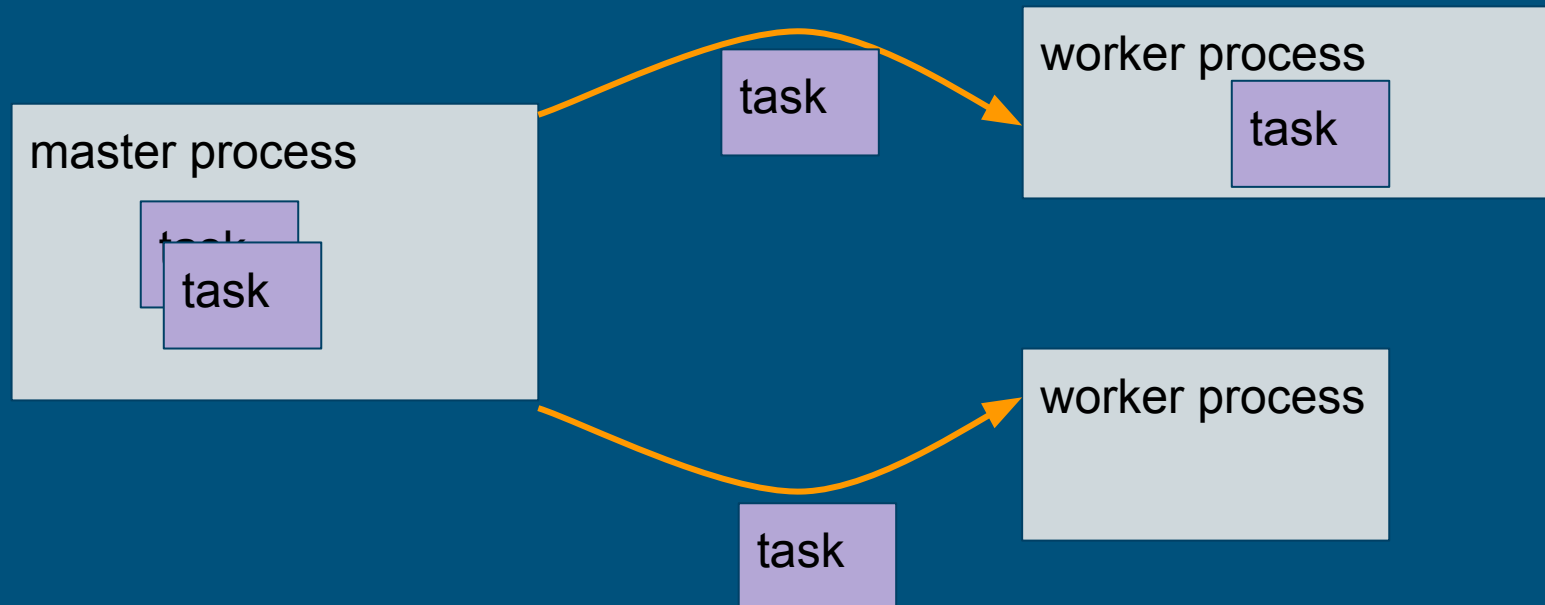
In a master worker application...

master-worker application



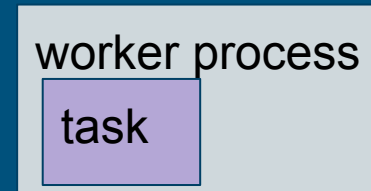
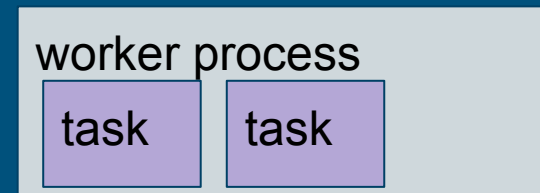
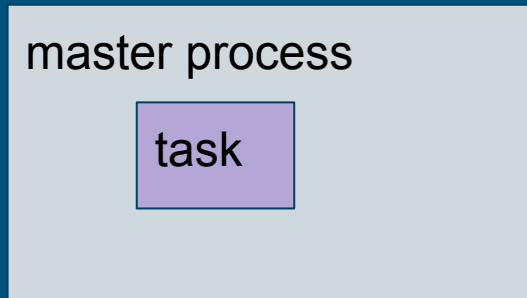
the master process generates tasks...

master-worker application



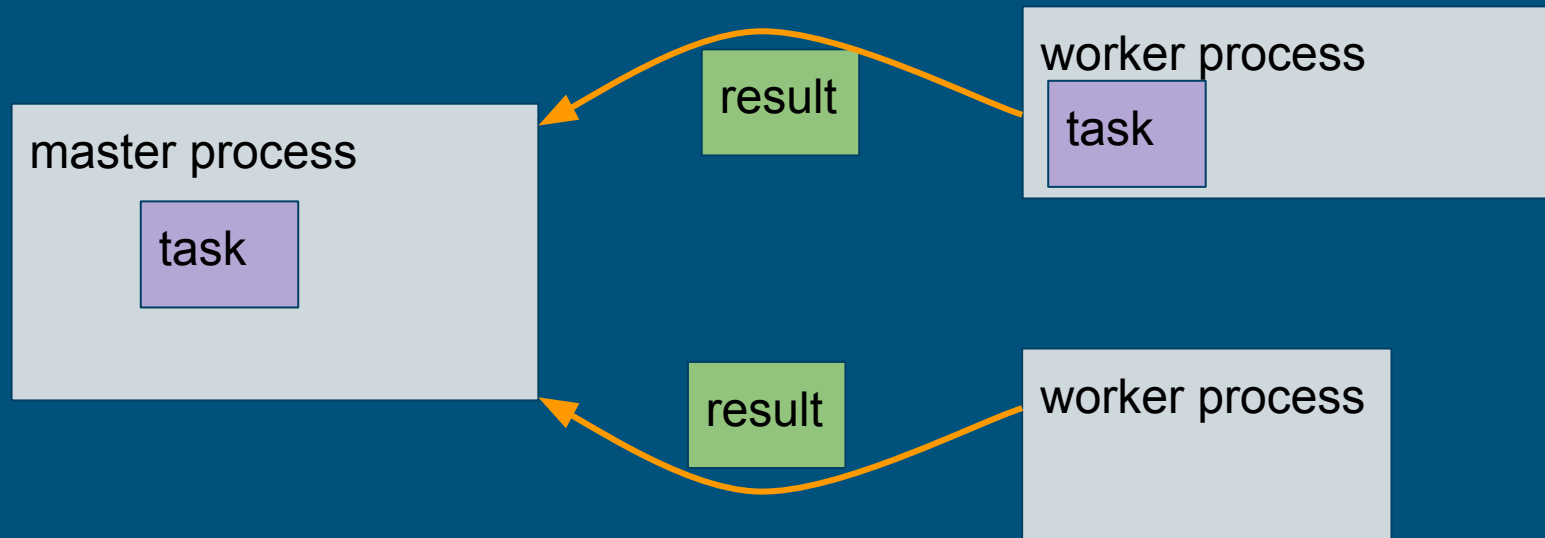
... delivers them to worker processes to execute...

master-worker application



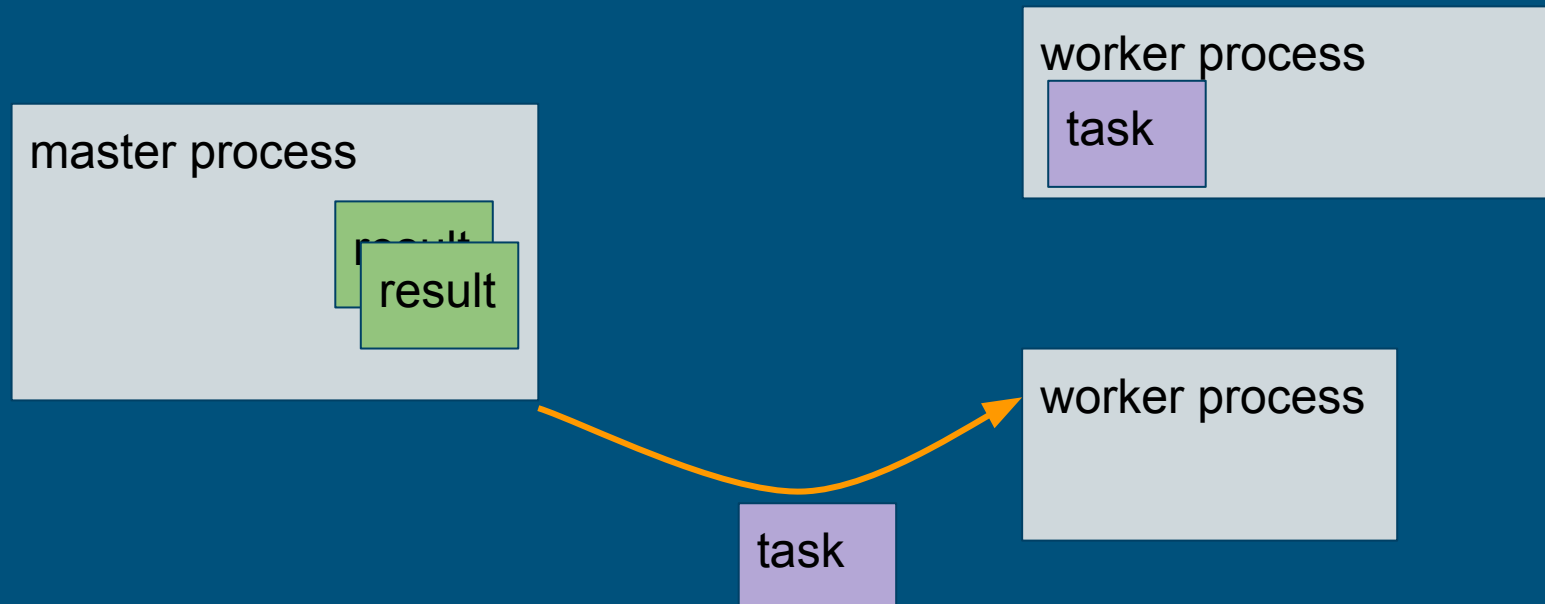
... waits for workers to execute tasks ...

master-worker application



and gathers the results on completion.

master-worker application



and on and on until no more tasks are generated.

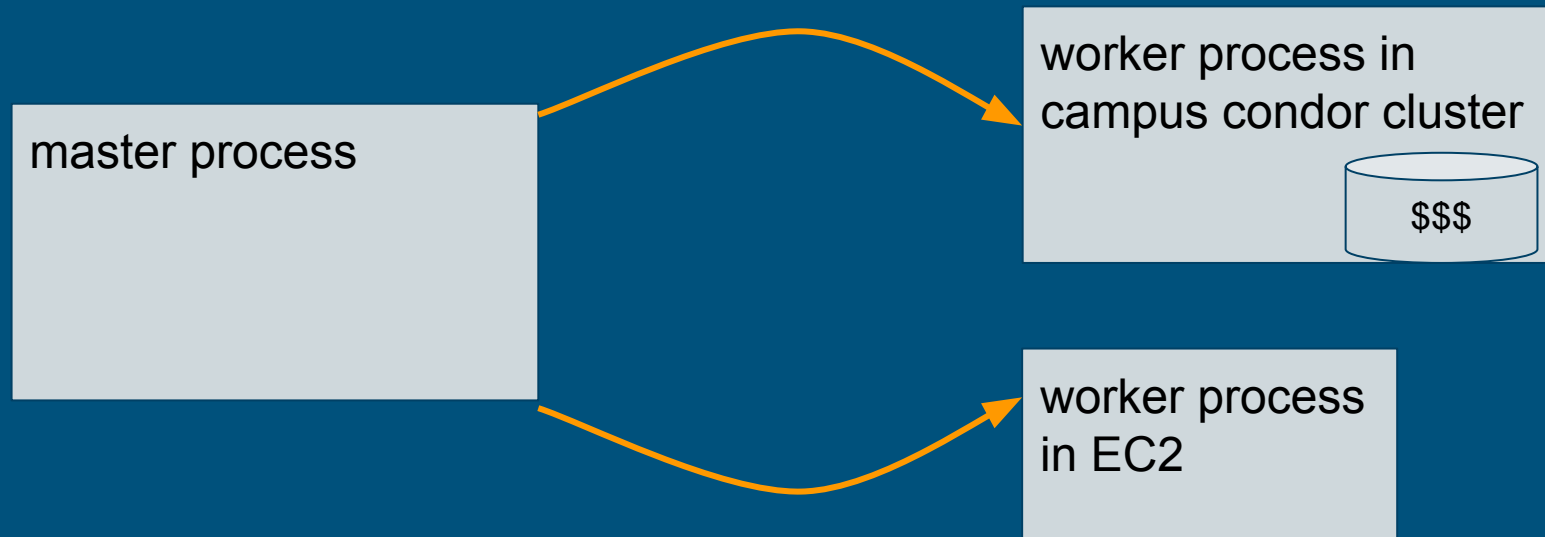
pure condor vs wq master-worker

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.)

makeflow as a work queue master (try at home)

```
$ cd ~/cctools-tutorial/makeflow/example_04

# -Twq to use work queue
# -M to give a name to our master. Workers will use this
# name to find the master

$ makeflow -Twq -M ${USER}-my-makeflow example_04.mf

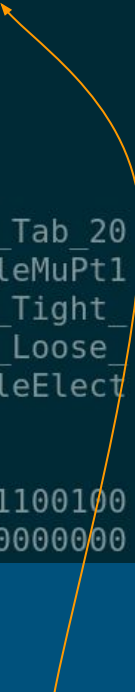
# see tasks waitings, workers connected, etc...
$ work_queue_status

...
```

work_queue_status

```
cclws16 master example_03 > work_queue_status
```

PROJECT	HOST	PORT	WAITING	RUNNING	COMPLETE	WORKERS
shadho-thermalize	45.55.219.221	9123	400	0	60438	0
btovar-my-makeflow	cclws16.cse.nd.edu	9000	2	0	0	0
vortex_network59	condorfe.crc.nd.edu	37763	0	3	997	3
vortex_network58	condorfe.crc.nd.edu	37762	0	2	998	2
vortex_network57	condorfe.crc.nd.edu	37761	0	37	2197	680
vortex_network56	condorfe.crc.nd.edu	37760	0	2	998	2
lbwa_step_check	disc24.crc.nd.edu	9050	1	0	0	0
lobster_rbucci_Extr_Tab_20	earth.crc.nd.edu	9000	6	36	4867	15
lobster_rbucci_SingleMuPt1	earth.crc.nd.edu	9004	30	20	1647	7
lobster_rbucci_Extr_Tight	earth.crc.nd.edu	9001	17	42	4686	14
lobster_rbucci_Extr_Loose	earth.crc.nd.edu	9002	10	36	4505	12
lobster_rbucci_SingleElect	earth.crc.nd.edu	9003	32	22	207	8
forcebalance	entropy.ucsd.edu	1800	0	1	1612	3
forcebalance	nighthawk.ucsd.edu	3397	15	0	0	0
smallpyr_2c_local_01100100	submit-1.chtc.wisc.edu	10017	1	0	0	0
smallpyr_2c_local_00000000	submit-1.chtc.wisc.edu	10013	0	1	72	54

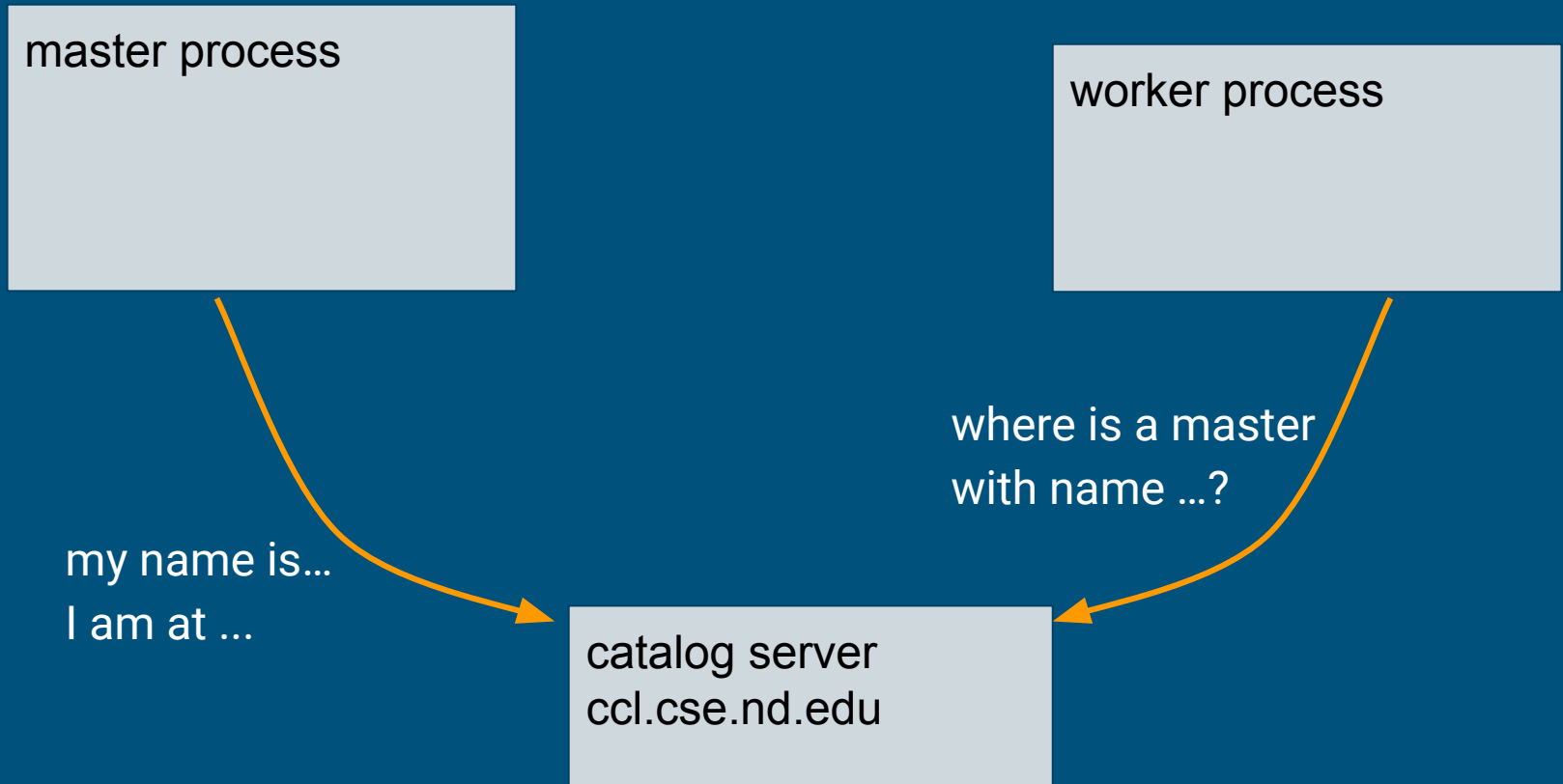


`$ {USER}-my-makeflow`

feed a worker to the master (try at home)

```
# in another terminal...  
  
# -M ${USER}-my-makeflow 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 --single-shot -M ${USER}-my-makeflow
```


how did the worker find the master?



if the defaults don't work for you

Before launching the makeflow, specify the range of ports available

default range is 9000-9999

source ../etc/uofwm-env sets the following range:

```
export TCP_LOW_PORT=10000
export TCP_HIGH_PORT=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 ${USER}-my-makeflow \
                        --timeout 60        \
                        3
```

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.)

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
}
```


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:
```

```
$ makeflow ... --password my.password.txt
```

```
# tell workers to use the password:
```

```
$ work_queue_workers ... --password my.password.txt
```

```
# NOTE THAT THE PASSWORD IS SIMPLY TO VERIFY A HANDSHAKE
```

```
# IT MAY NOT PROTECT AGAINST MALICIOUS ATTACKS
```

break



writing master-worker applications with work queue

makeflow vs. work queue when describing workflows

makeflow:

- directed-acyclic graph dynamic model
- workflow structure is fixed and static
- computes dependencies between tasks
- classic unix make language or JSON

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

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 B')

# ...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('B')

# 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 (try at home)

```
# tell python where to find work queue
$ source ~/cctools-tutorial/etc/uofwm-env

$ cd ~/cctools-tutorial/work_queue/example_01

# modify wq_mini.py with a master name you like,
# (currently set to ${USER}-my-first-master), then
$ python example_01.py

# in some other terminal, launch a worker for that master
# workers don't need PYTHONPATH set.
$ work_queue_worker -M your-master-name --single-shot
```

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('./cmd -output out.{1} -parameter {1}'.format(i))
    t.specify_input_file('cmd', cache=True)
    t.specify_output_file('out.{1}'.format(i))
    # 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:
        ...
```


tasks to the races -- duplicating tasks

```
# 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
```

specifying tasks resources

```
q.specify_category_max_resources('my_category',  
{  
    'cores' : 1,  
    'memory' : 1024,  
    'disk' : 1014  
})
```

```
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:
        ...
```

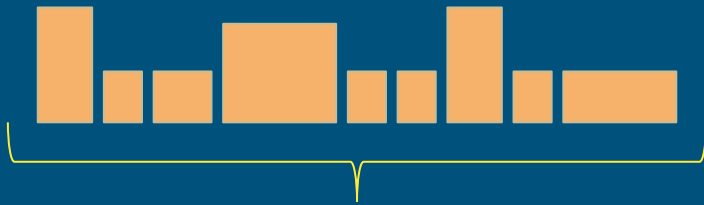
Monitoring and enforcement

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:

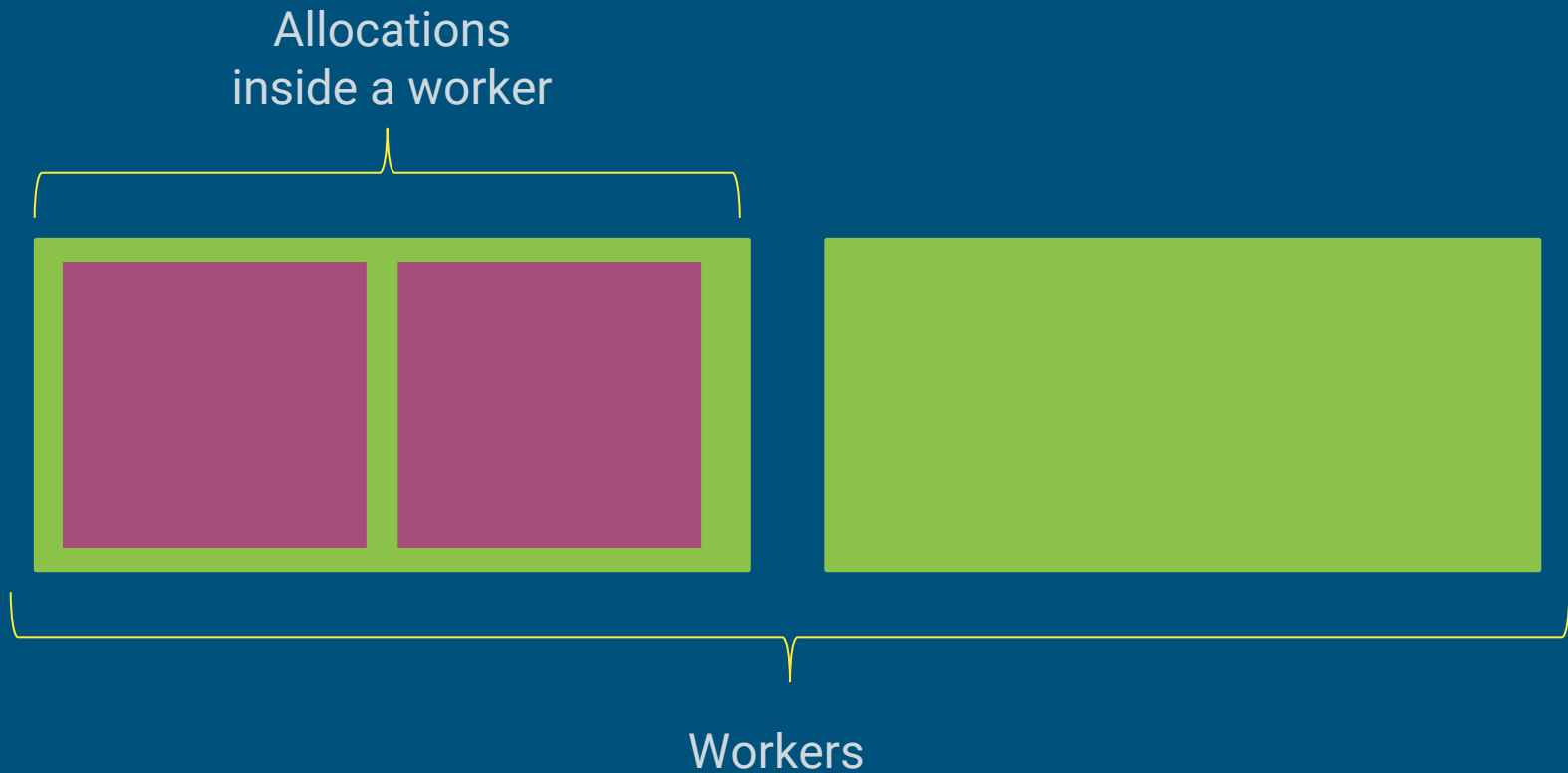
Resource contention/exhaustion, reduce throughput



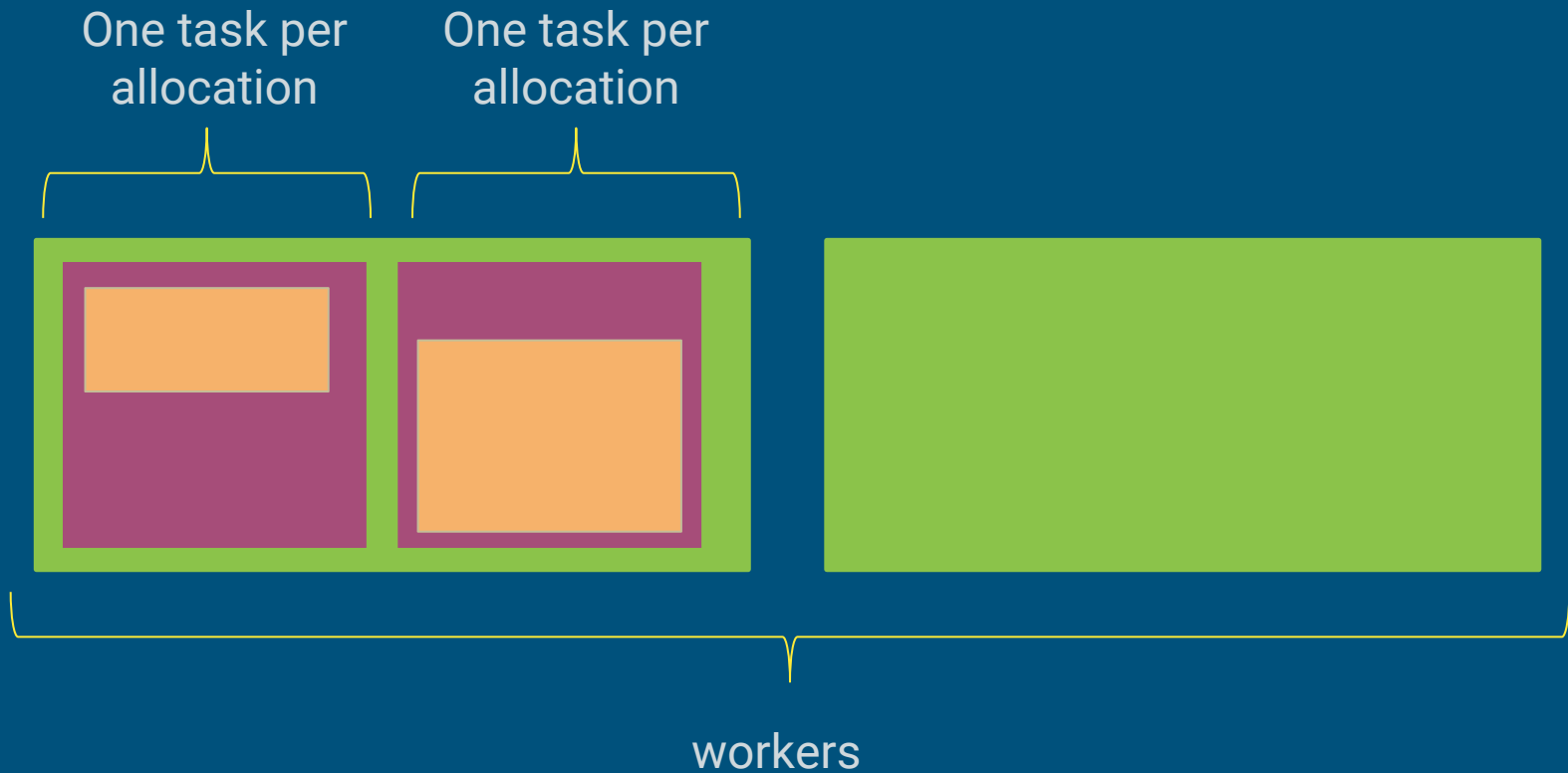
Task-in-the-Box



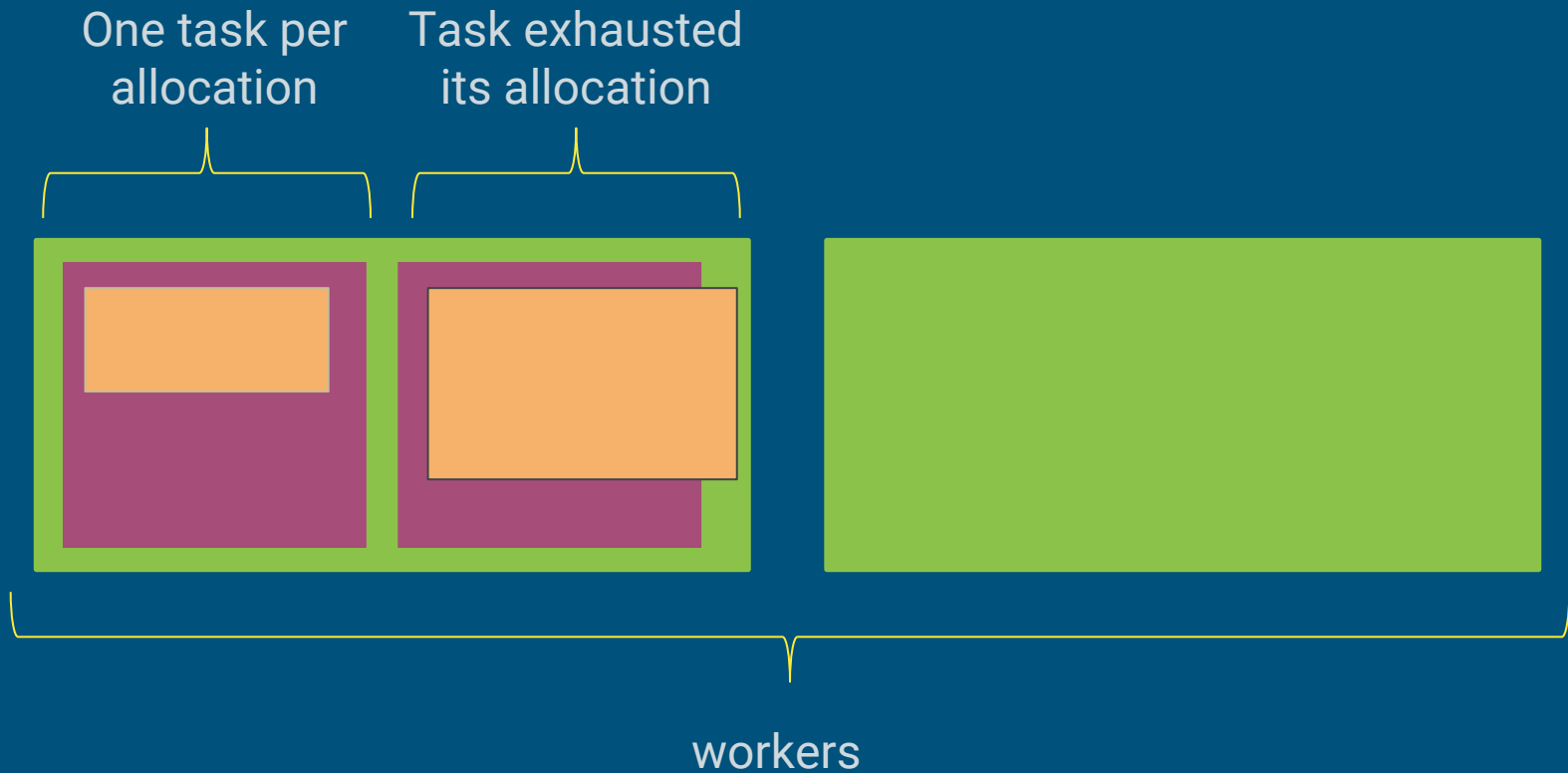
Task-in-the-Box



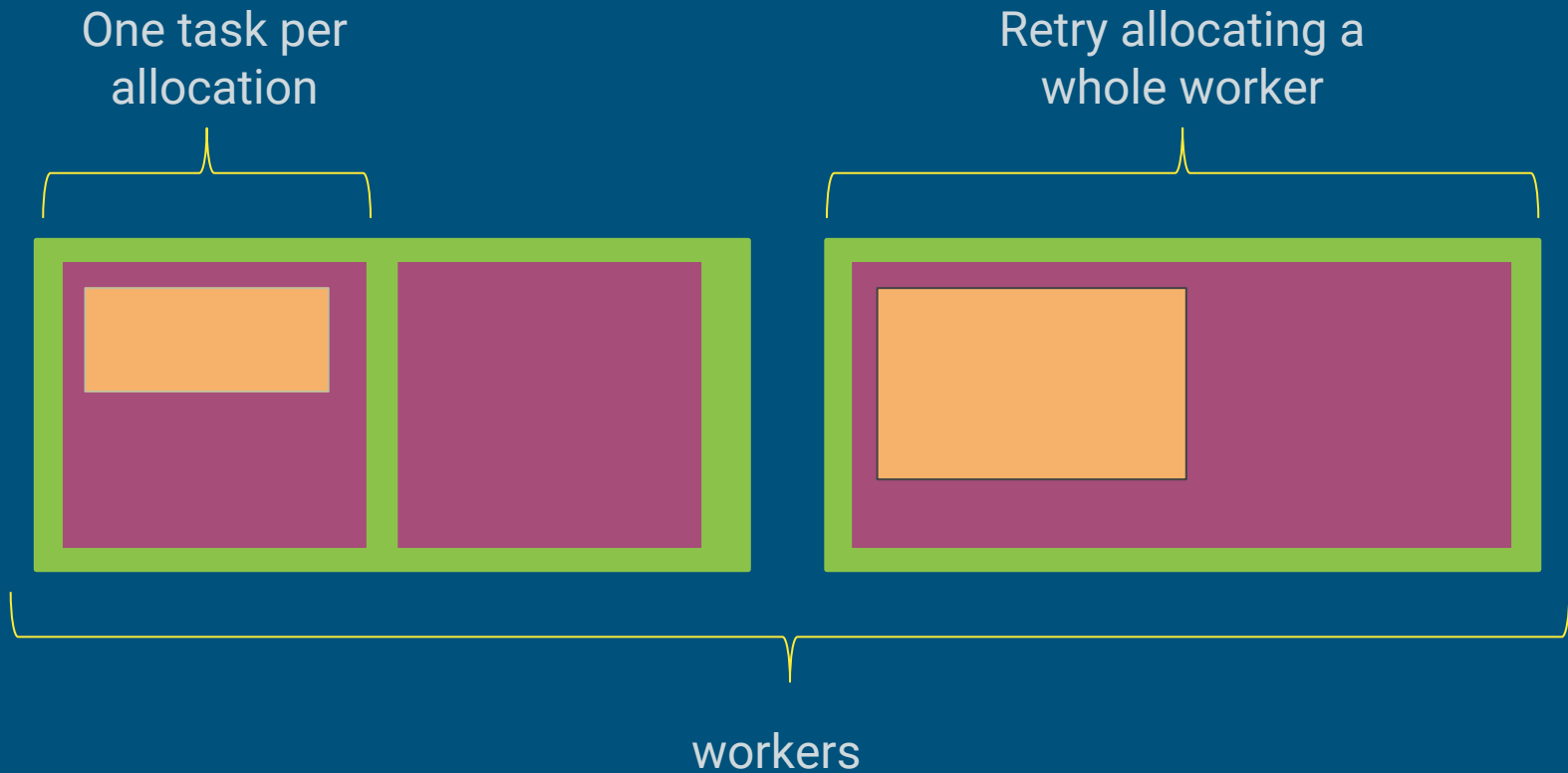
Task-in-the-Box



Task-in-the-Box



Task-in-the-Box

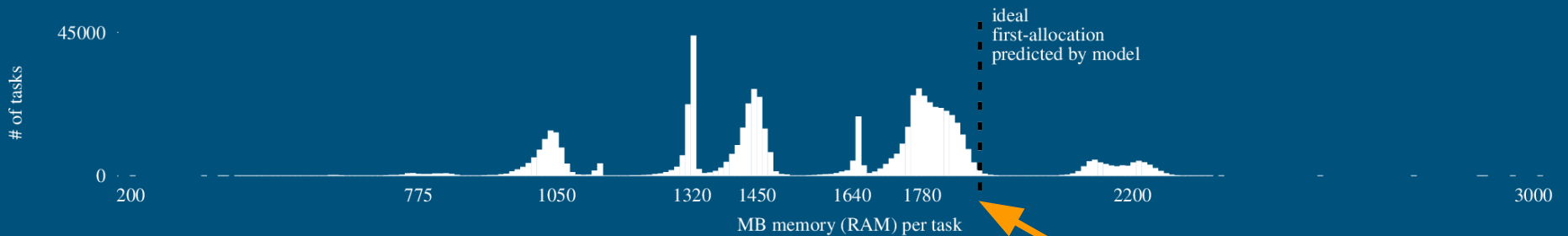


ND CMS example

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](https://doi.org/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 tried 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 200MB file, using 10MB of memory buffer.
for i in range(0,30):
    t = WQ.Task('python task.py 200')
    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 (try at home)

```
$ 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.

resources in Makeflow without WQ

```
$ makeflow -Tcondor --monitor=my_dir Makeflow
```

```
# one resource summary per rule:
```

```
$ cat mydir/resource-rule-2.summary
```

```
# Makeflow file
```

```
.MAKEFLOW CATEGORY MY_FIRST_CATEGORY  
.MAKEFLOW MODE MAX_THROUGHPUT  
.MAKEFLOW CATEGORY MY_SECOND_CATEGORY  
.MAKEFLOW MODE MIN_WASTE  
.MAKEFLOW CATEGORY MY_OTHER_CATEGORY  
.MAKEFLOW MODE FIXED
```

```
.MAKEFLOW CATEGORY MY_FIRST_CATEGORY  
output_a: input_a  
    cmd < input_a > output_a
```

```
.MAKEFLOW CATEGORY MY_SECOND_CATEGORY  
output_b: input_b  
    cmd < input_b > output_b
```

```
.MAKEFLOW CATEGORY MY_OTHER_CATEGORY  
output_c: input_c  
    cmd < input_c > output_c
```

```
% makeflow --monitor=my_dir --retry-count=5
```

configuring runtime logs

We recommend to always to enable all 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

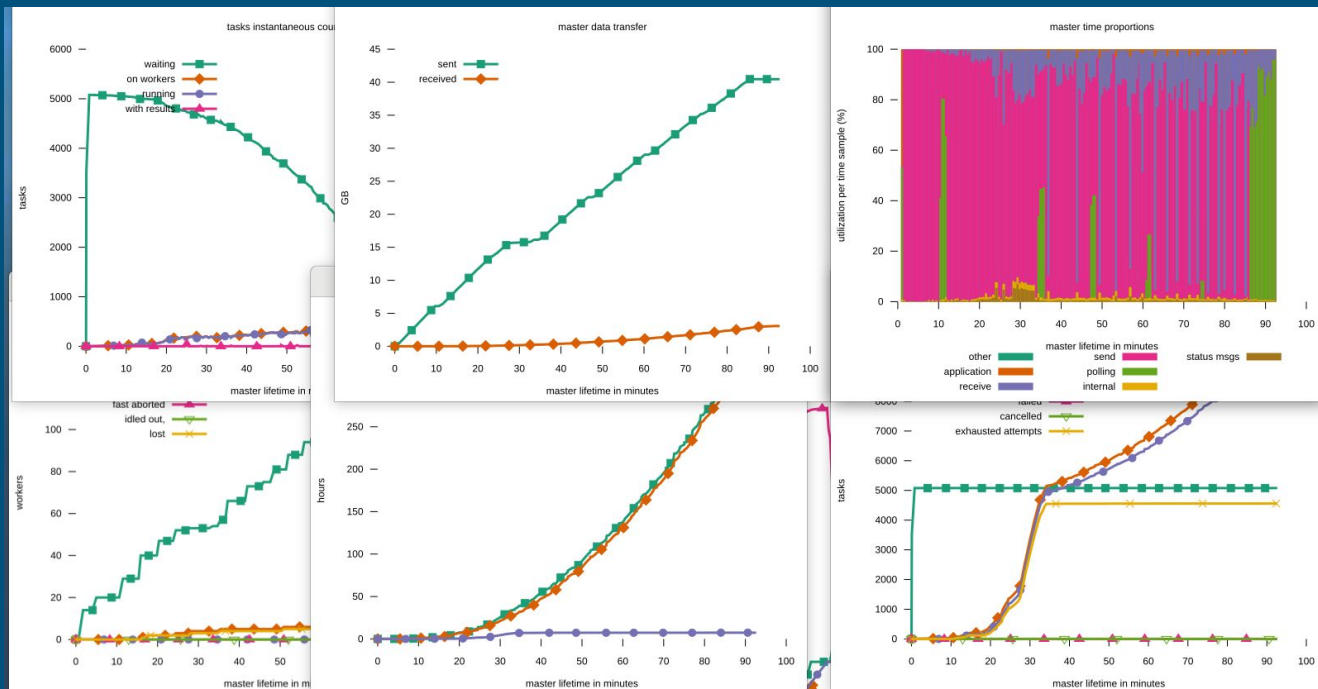
```
$ grep '\<TASK 1\>' example_02.tr
```

```
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":[1550698004
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_memor
y":[0,"MB"],"bytes_read":[0.00138569,"MB"],"bytes_written":[0,"MB"],"bytes_received":[0,"MB"],"byte
dth":[0,"Mbps"],"total_files":[7,"files"],"disk":[201,"MB"],"machine_cpus":[8,"cores"],"machine_lo
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":[15506980460
0.989,"cores"],"cores":[1,"cores"],"wall_time":[0.146097,"s"],"cpu_time":[0.144457,"s"],"max_conc
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":[1550698046226078
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
```


miscellaneous work queue calls

```
# kill workers slower than alpha times the average  
q.activate_fast_abort(alpha)
```

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

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

```
# specify password file  
q.specify_password_file(filename)
```

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

```
# 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/html/namespaces.html>

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"],"cores":[1,"cores"],
"max_concurrent_processes":[1,"procs"],"total_processes":[1,"procs"],"memory":[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)

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>



extra slides

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)
```

download binary package (try at home)

For most of you at University of Wisconsin-Madison:

<http://ccl.cse.nd.edu/software/download>

and download the most recent stable version for redhat 7 into ~/cctools-tutorial

or for today, shortcut in a terminal:

```
$ cd ~/cctools-tutorial
$ bin/download-cctools
```

install cctools (try at home)

```
$ cd ~/cctools-tutorial

# 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)
$ PATH=~:/cctools-tutorial/cctools/bin:${PATH}
$ export PATH

# (or for today, set your environment with:)
$ source ~/cctools-tutorial/etc/cctools-env
```


from source (maybe try at home)

```
$ cd ~/cctools-tutorial  
  
# decompress cctools  
$ tar -xf cctools-*-src.tar.gz  
  
# configure and install  
$ cd cctools-*-src  
$ ./configure --prefix ~/cctools-tutorial/cctools  
$ make  
$ make install
```