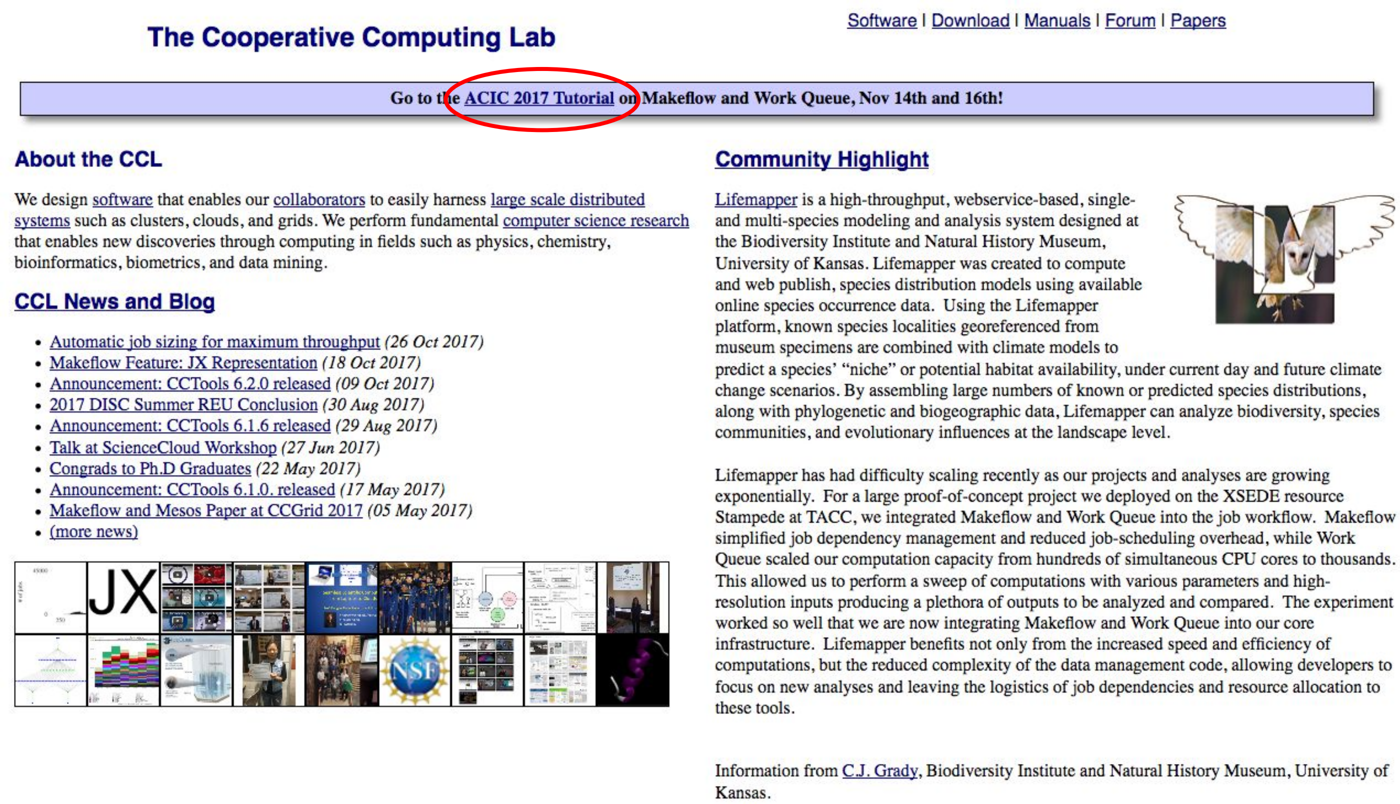
UNIVERSITY OF NOTRE DAME

Introduction to Makeflow and Work Queue

Nick Hazekamp
University of Notre Dame
nhazekam@nd.edu

CCTools

Go to http://ccl.cse.nd.edu and Click on ACIC Tutorial



Software | Download | Manuals | Forum | Papers

The Cooperative Computing Lab

Go to the ACIC 2017 Tutorial on Makeflow and Work Queue, Nov 14th and 16th!

About the CCL

We design software that enables our collaborators to easily harness large scale distributed systems such as clusters, clouds, and grids. We perform fundamental computer science research that enables new discoveries through computing in fields such as physics, chemistry, bioinformatics, biometrics, and data mining.

CCL News and Blog

- Automatic job sizing for maximum throughput *(26 Oct 2017)*
- Makeflow Feature: JX Representation *(18 Oct 2017)*
- Announcement: CCTools 6.2.0 released *(09 Oct 2017)*
- 2017 DISC Summer REU Conclusion *(30 Aug 2017)*
- Announcement: CCTools 6.1.6 released *(29 Aug 2017)*
- Talk at ScienceCloud Workshop *(27 Jun 2017)*
- Congrads to Ph.D Graduates *(22 May 2017)*
- Announcement: CCTools 6.1.0. released *(17 May 2017)*
- Makeflow and Mesos Paper at CCGrid 2017 *(05 May 2017)*
- (more news)

Community Highlight

Lifemapper is a high-throughput, webservice-based, single- and multi-species modeling and analysis system designed at the Biodiversity Institute and Natural History Museum, University of Kansas. Lifemapper was created to compute and web publish, species distribution models using available online species occurrence data. Using the Lifemapper platform, known species localities georeferenced from museum specimens are combined with climate models to predict a species' "niche" or potential habitat availability, under current day and future climate change scenarios. By assembling large numbers of known or predicted species distributions, along with phylogenetic and biogeographic data, Lifemapper can analyze biodiversity, species communities, and evolutionary influences at the landscape level.

Lifemapper has had difficulty scaling recently as our projects and analyses are growing exponentially. For a large proof-of-concept project we deployed on the XSEDE resource Stampede at TACC, we integrated Makeflow and Work Queue into the job workflow. Makeflow simplified job dependency management and reduced job-scheduling overhead, while Work Queue scaled our computation capacity from hundreds of simultaneous CPU cores to thousands. This allowed us to perform a sweep of computations with various parameters and high-resolution inputs producing a plethora of outputs to be analyzed and compared. The experiment worked so well that we are now integrating Makeflow and Work Queue into our core infrastructure. Lifemapper benefits not only from the increased speed and efficiency of computations, but the reduced complexity of the data management code, allowing developers to focus on new analyses and leaving the logistics of job dependencies and resource allocation to these tools.

Information from C.J. Grady, Biodiversity Institute and Natural History Museum, University of Kansas.

The Cooperative Computing Lab

- We **collaborate with people** who have large scale computing problems in science, engineering, and other fields.
- We **operate computer systems** on the O(10,000) cores: clusters, clouds, grids.
- We **conduct computer science research** in the context of real people and problems.
- We **develop open source software** for large scale distributed computing.

http://ccl.cse.nd.edu

Outline

Tuesday, Nov 14th

- Thinking Opportunistically
- Overview of the Cooperative Computing Tools
- Makeflow
- Makeflow + Work Queue
- Hands-On Tutorial

Thursday, Nov 16th

- Makeflow Features
  - Resource Management
  - Containers
- Work Queue API
- Hands-On Tutorial

Thinking Opportunistically

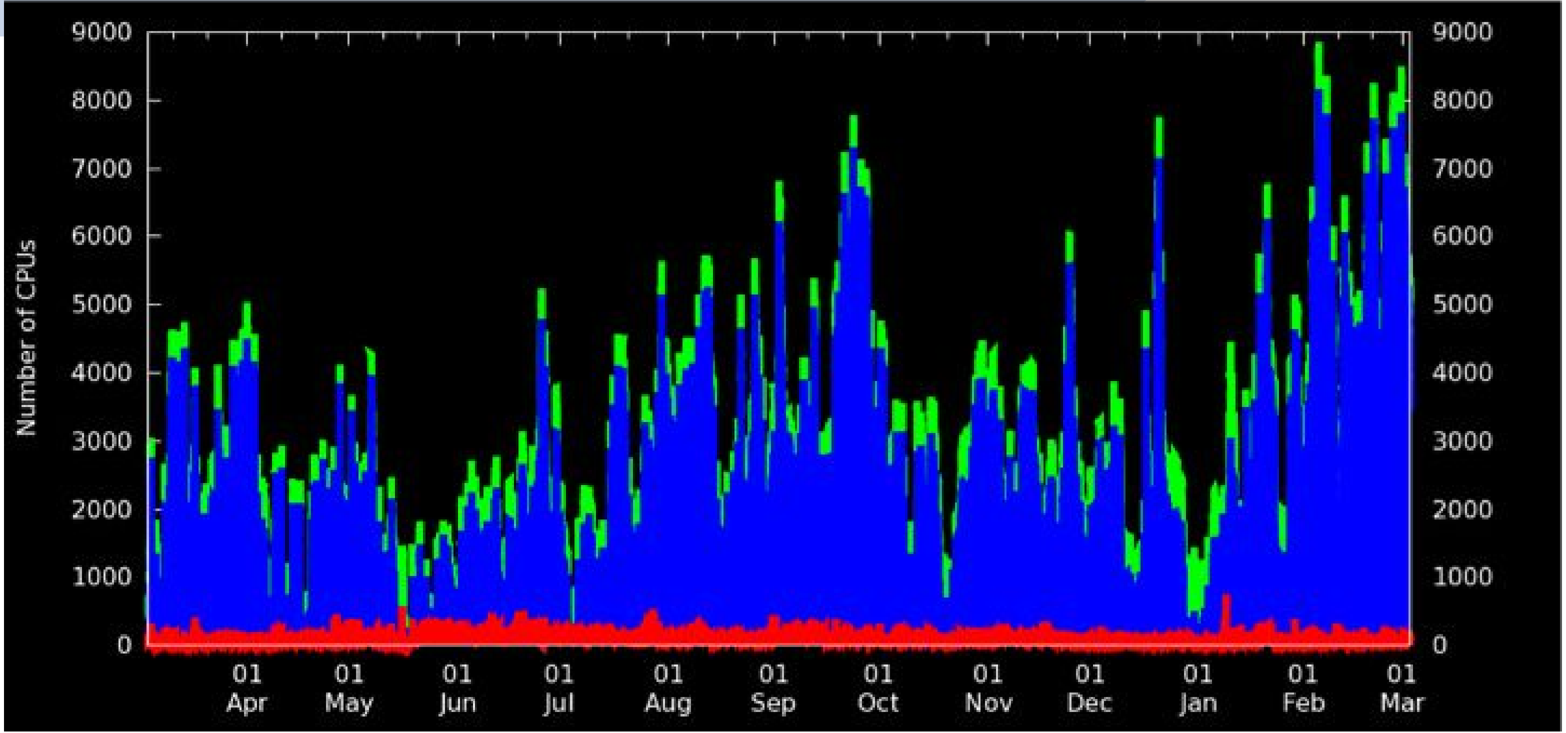
Opportunistic Computing

- Much of scientific computing is done in conventional computing centers with a fixed operating environment with professional sysadmins.
- But, there exists a large amount of computing power available to end users that is not prepared or tailored to your specific application:
  - National HPC facility
  - Campus-level cluster and batch system.
  - Volunteer computing systems: Condor, BOINC, etc.
  - Cloud services.
- Can we effectively use these systems for “long tail” scientific computing?

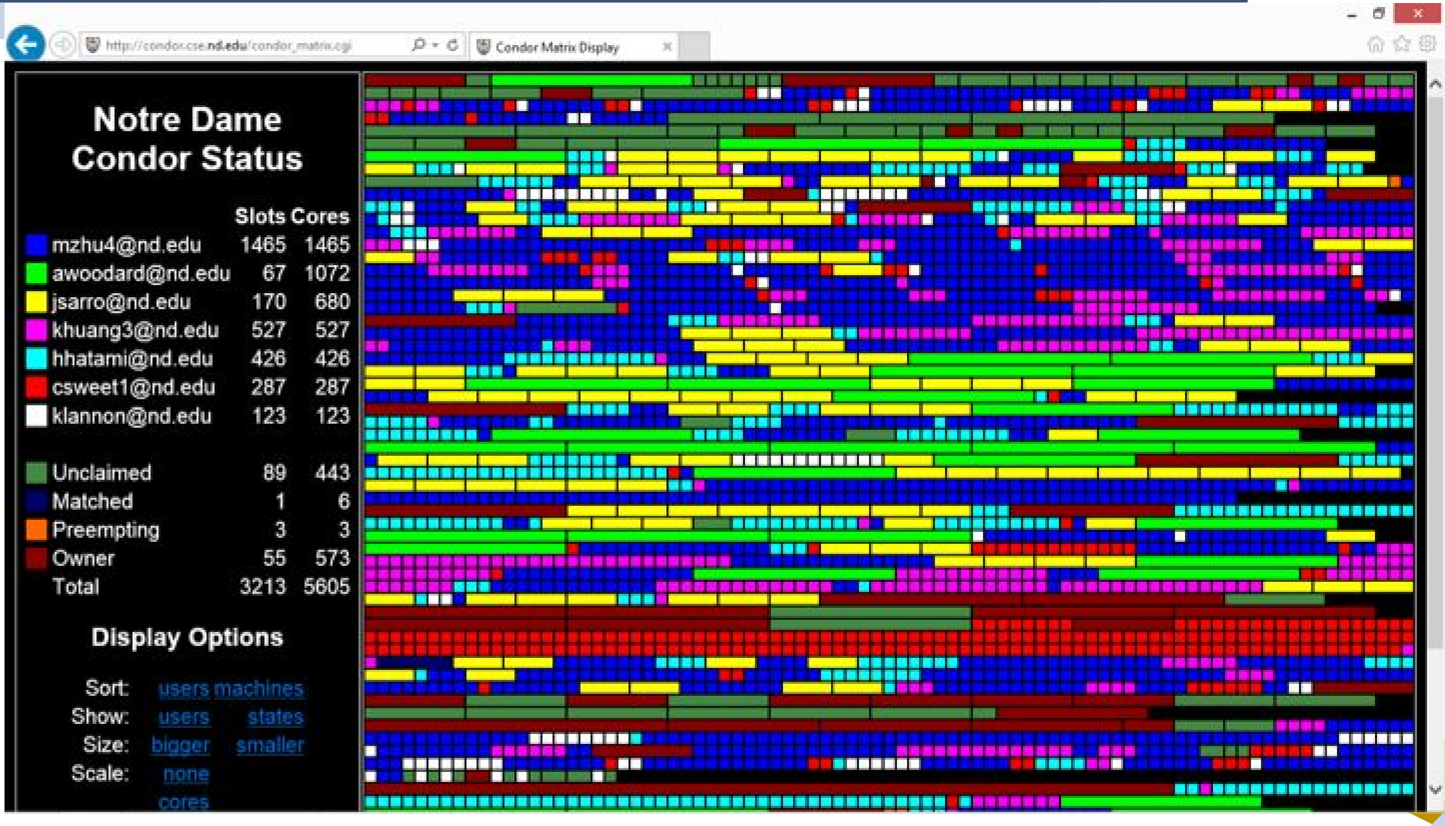
Opportunistic Challenges

- When borrowing someone else's machines, you cannot change the OS distribution, update RPMs, patch kernels, run as root…
- This often puts important technology just out of reach of the end user, e.g.:
  - FUSE might be installed, but without setuid binary.
  - Docker might be available, but you aren't a member of the required Unix group.
- The resource management policies of the hosting system may work against you:
  - Preemption due to submission by higher priority users.
  - Limitations on execution time and disk space.
  - Firewalls only allow certain kinds of network connections.

Backfilling HPC with Condor at Notre Dame



Users of Opportunistic Cycles



Notre Dame Condor Status

		Slots	Cores
🟦	mzhu4@nd.edu	1465	1465
🟩	awoodard@nd.edu	67	1072
🟨	jsarro@nd.edu	170	680
🟪	khuang3@nd.edu	527	527
🟦	hhatami@nd.edu	426	426
🟥	csweet1@nd.edu	287	287
⬜	klannon@nd.edu	123	123
🟩	Unclaimed	89	443
🟦	Matched	1	6
🟧	Preempting	3	3
🟫	Owner	55	573
	Total	3213	5605

Display Options

Sort: users machines
Show: users states
Size: bigger smaller
Scale: none
cores

I can get as many machines
on the cloud/grid as I want!

How do I organize my application
to run on those machines?

Cooperative Computing Tools

Our Philosophy

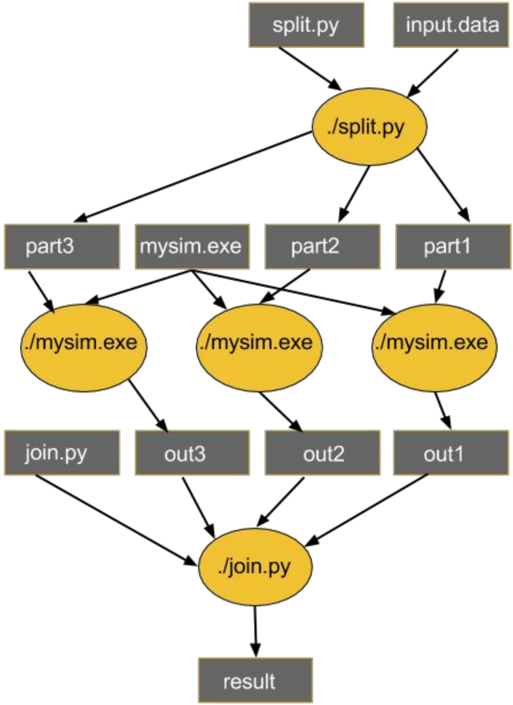
- Harness all available resources: desktops, clusters, clouds, and grids.
- Make it easy to scale up from one desktop to national scale infrastructure.
- Provide familiar interfaces that make it easy to connect existing apps together.
- Allow portability across operating systems, storage systems, middleware…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

A Quick Tour of the CCTools

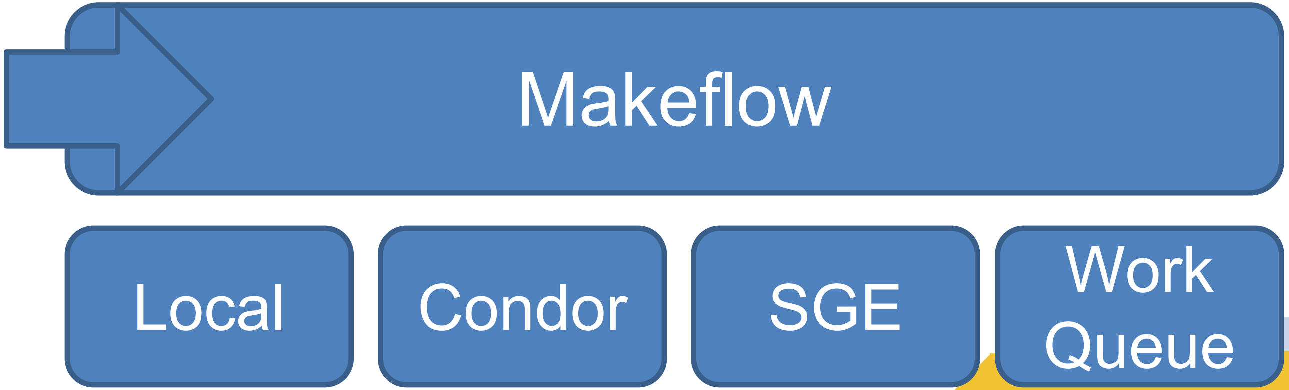
- Open source, GNU General Public License.
- Compiles in 1-2 minutes, installs in $HOME.
- Runs on Linux, Solaris, MacOS, FreeBSD, …
- Interoperates with many distributed computing systems.
  - Condor, SGE, Torque, Globus, iRODS, Hadoop…
- Components:
  - Makeflow – A portable workflow manager.
  - Work Queue – A lightweight distributed execution system.
  - Parrot – A personal user-level virtual file system.
  - Chirp – A user-level distributed filesystem.

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

MAKEFLOW (MAKE + WORKFLOW)



- Provides portability across batch systems.
- Enable parallelism (but not too much!)
- Fault tolerance at multiple scales.
- Data and resource management.



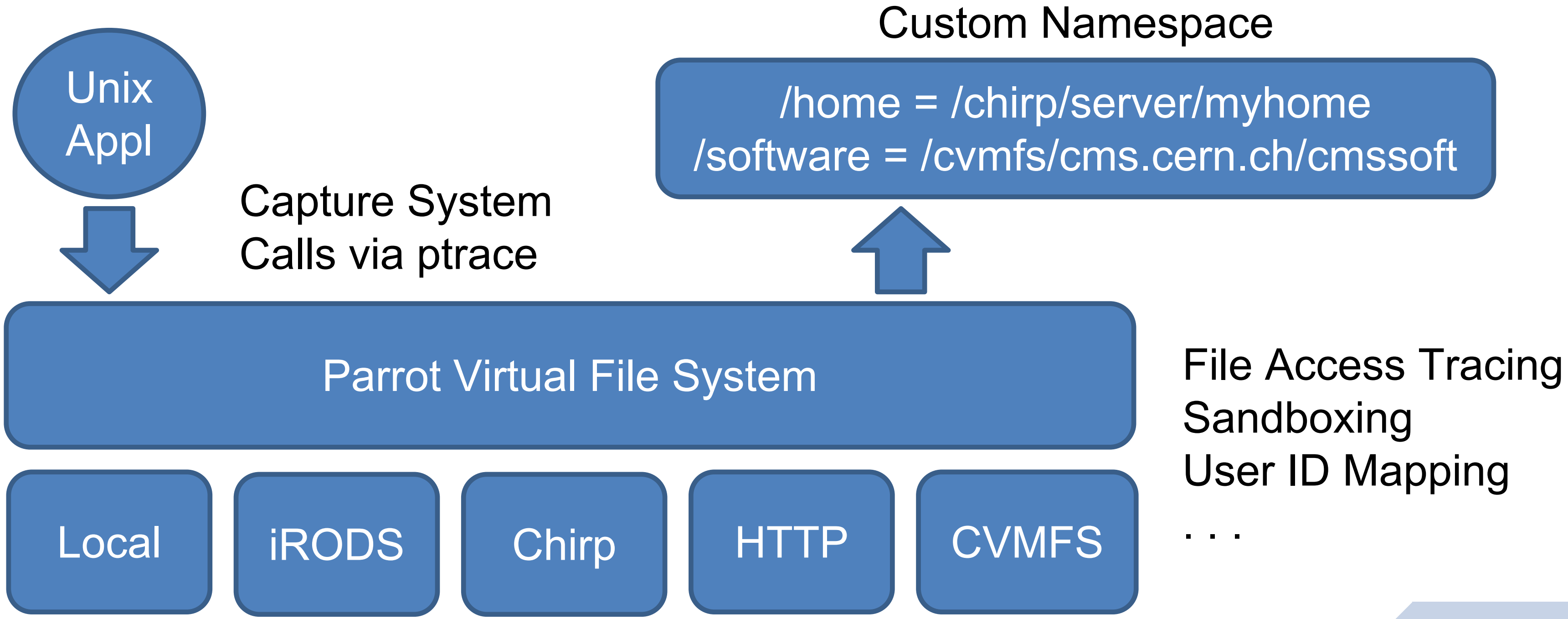
http://ccl.cse.nd.edu/software/makeflow

Work Queue API

```
#include  “work_queue.h”
while( not done ) {

        while (more work ready) {
        task = work_queue_task_create();
                // add some details to the task
                work_queue_submit(queue, task);
        }

        task = work_queue_wait(queue);
        // process the completed task
}
```

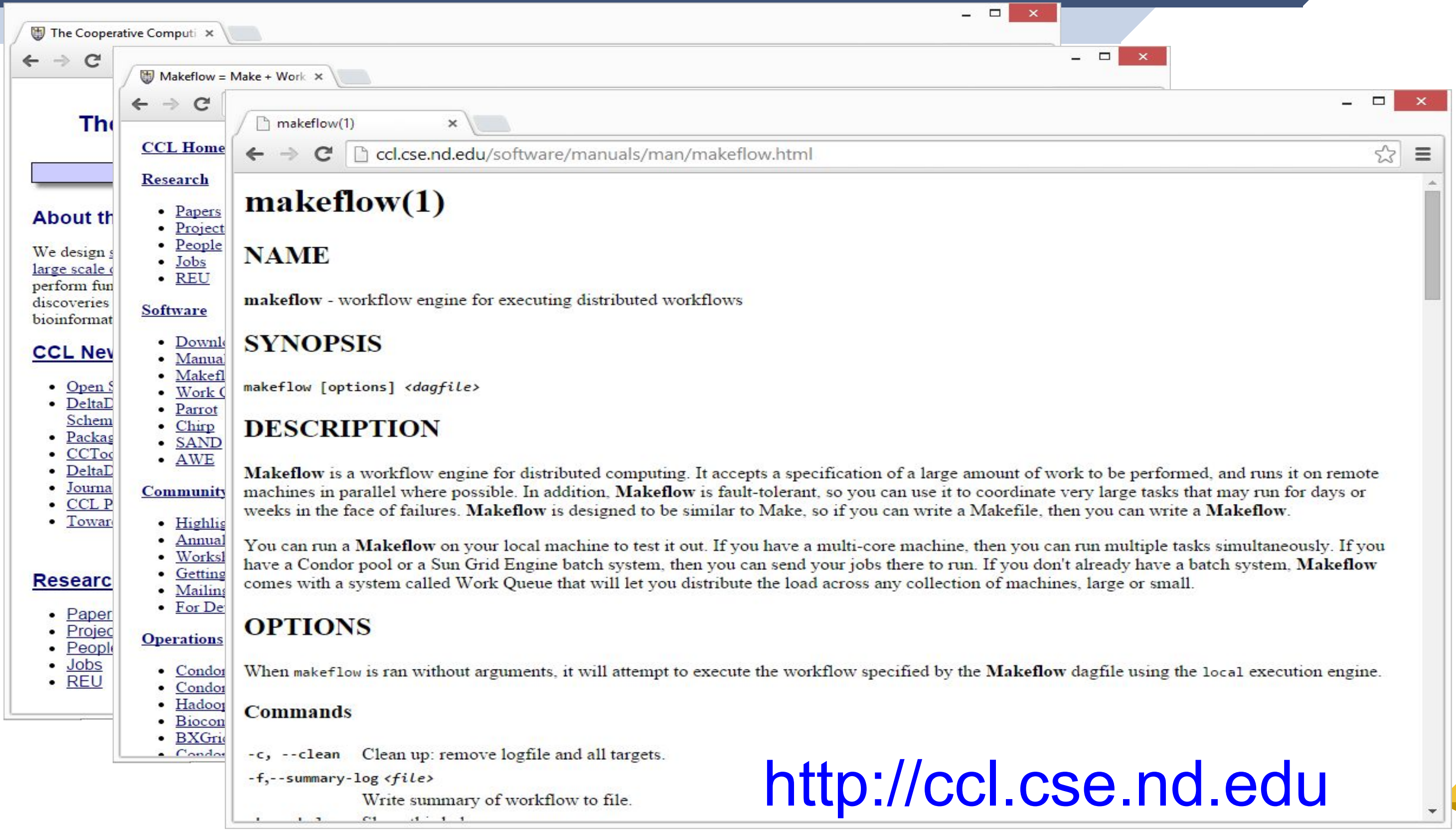
http://ccl.cse.nd.edu/software/workqueue

Parrot Virtual File System


Custom Namespace
/home = /chirp/server/myhome
/software = /cvmfs/cms.cern.ch/cmssoft
Unix Appl
Capture System Calls via ptrace
Parrot Virtual File System
File Access Tracing
Sandboxing
User ID Mapping
. . .
Local
iRODS
Chirp
HTTP
CVMFS

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

Lots of Documentation



makeflow(1)

NAME

makeflow - workflow engine for executing distributed workflows

SYNOPSIS

`makeflow [options] <dagfile>`

DESCRIPTION

Makeflow is a workflow engine for distributed computing. It accepts a specification of a large amount of work to be performed, and runs it on remote machines in parallel where possible. In addition, **Makeflow** is fault-tolerant, so you can use it to coordinate very large tasks that may run for days or weeks in the face of failures. **Makeflow** is designed to be similar to Make, so if you can write a Makefile, then you can write a **Makeflow**.

You can run a **Makeflow** on your local machine to test it out. If you have a multi-core machine, then you can run multiple tasks simultaneously. If you have a Condor pool or a Sun Grid Engine batch system, then you can send your jobs there to run. If you don't already have a batch system, **Makeflow** comes with a system called Work Queue that will let you distribute the load across any collection of machines, large or small.

OPTIONS

When `makeflow` is ran without arguments, it will attempt to execute the workflow specified by the **Makeflow** dagfile using the `local` execution engine.

Commands

`-c, --clean` Clean up: remove logfile and all targets.

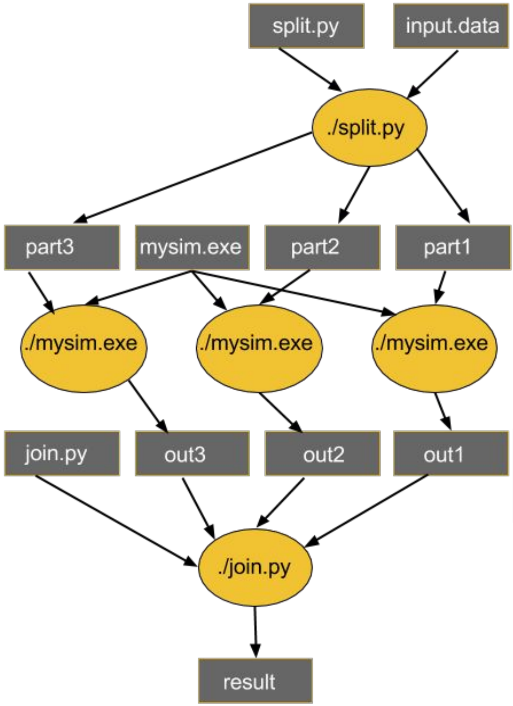
`-f,--summary-log <file>`
Write summary of workflow to file.

http://ccl.cse.nd.edu

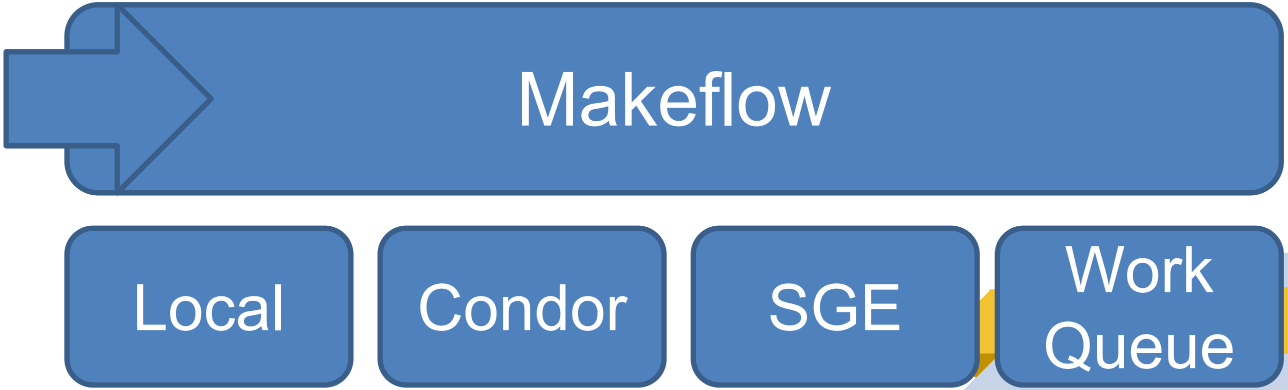
Makeflow

A Portable Workflow System

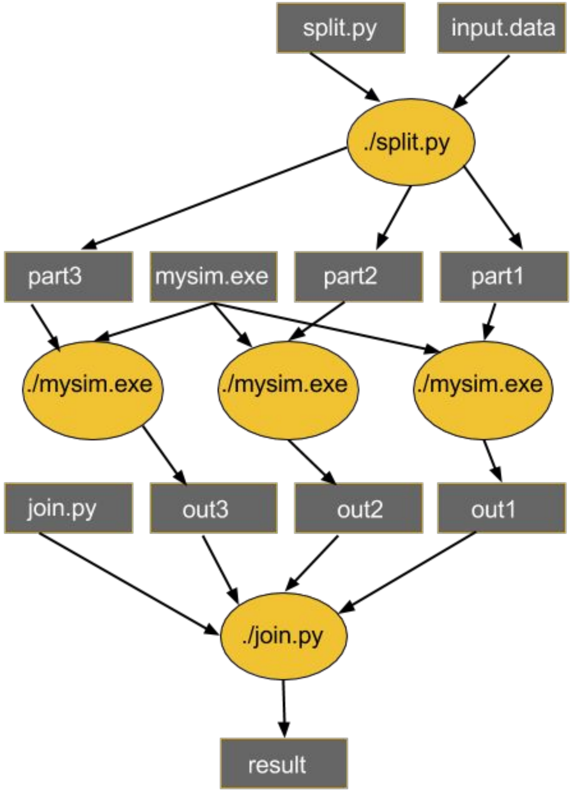
MAKEFLOW (MAKE + WORKFLOW)



- Provides portability across batch systems.
- Enable parallelism (but not too much!)
- Trickle out work to batch system
- Fault tolerance at multiple scales.
- Data and resource management.



MAKEFLOW (MAKE + WORKFLOW)
BASED OFF AN OLD IDEA: MAKEFILES

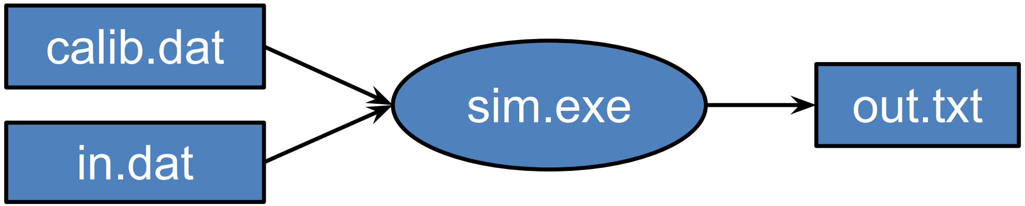


```
part1 part2 part3: input.data split.py
    ./split.py input.data

out1: part1 mysim.exe
    ./mysim.exe part1 >out1

out2: part2 mysim.exe
    ./mysim.exe part2 >out2

out3: part3 mysim.exe
    ./mysim.exe part3 >out3

result: out1 out2 out3 join.py
    ./join.py out1 out2 out3 > result
```

Makeflow Syntax

```
[output files] : [input files]
        [command to run]
```

One Rule



sim.exe in.dat –p 50 > out.txt

```
out.txt : in.dat calib.dat sim.exe
    sim.exe in.data –p 50 > out.txt
```

Makeflow Syntax : sims.mf

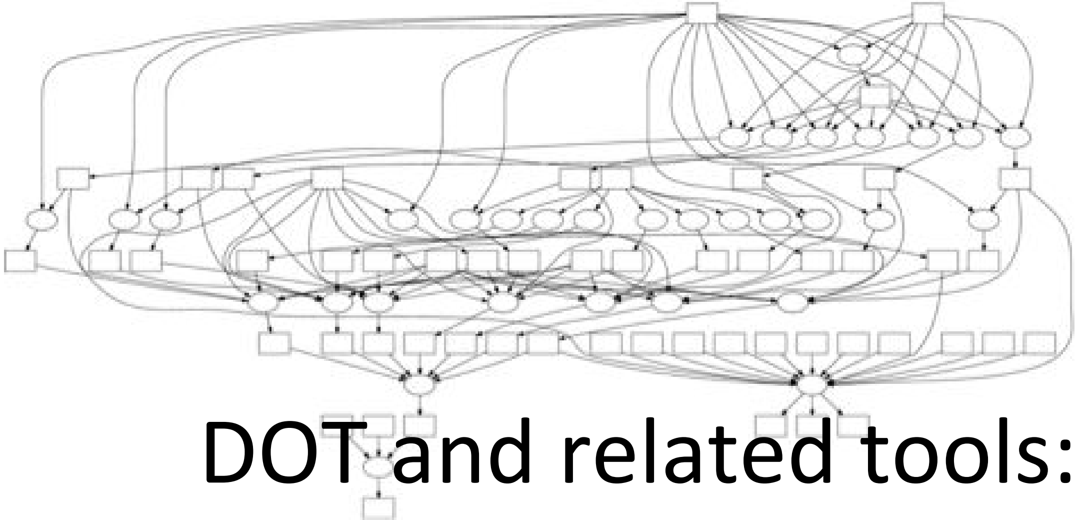
```
out.10 : in.dat calib.dat sim.exe
    sim.exe –p 10 in.data > out.10

out.20 : in.dat calib.dat sim.exe
    sim.exe –p 20 in.data > out.20

out.30 : in.dat calib.dat sim.exe
    sim.exe –p 30 in.data > out.30
```

How to run a Makeflow

- Run a workflow locally (multicore?)
  - makeflow -T local sims.mf
- Clean up the workflow outputs:
  - makeflow –c sims.mf
- Run the workflow on Torque:
  - makeflow –T torque sims.mf
- Run the workflow on Condor:
  - makeflow –T condor sims.mf

Visualization with DOT

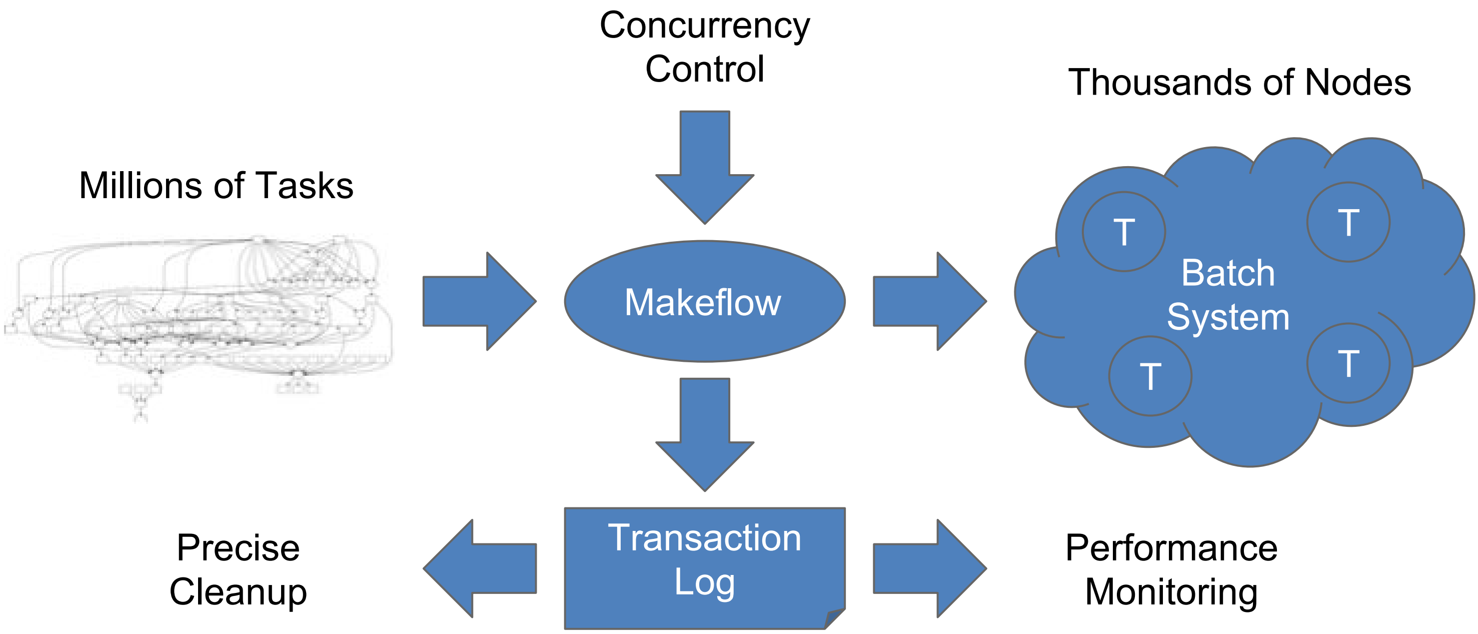
- makeflow_viz –D example.mf > example.dot
- dot –T gif < example.dot > example.gif



DOT and related tools:

http://www.graphviz.org

Makeflow Shapes a Workflow



Concurrency Control

Millions of Tasks

Makeflow

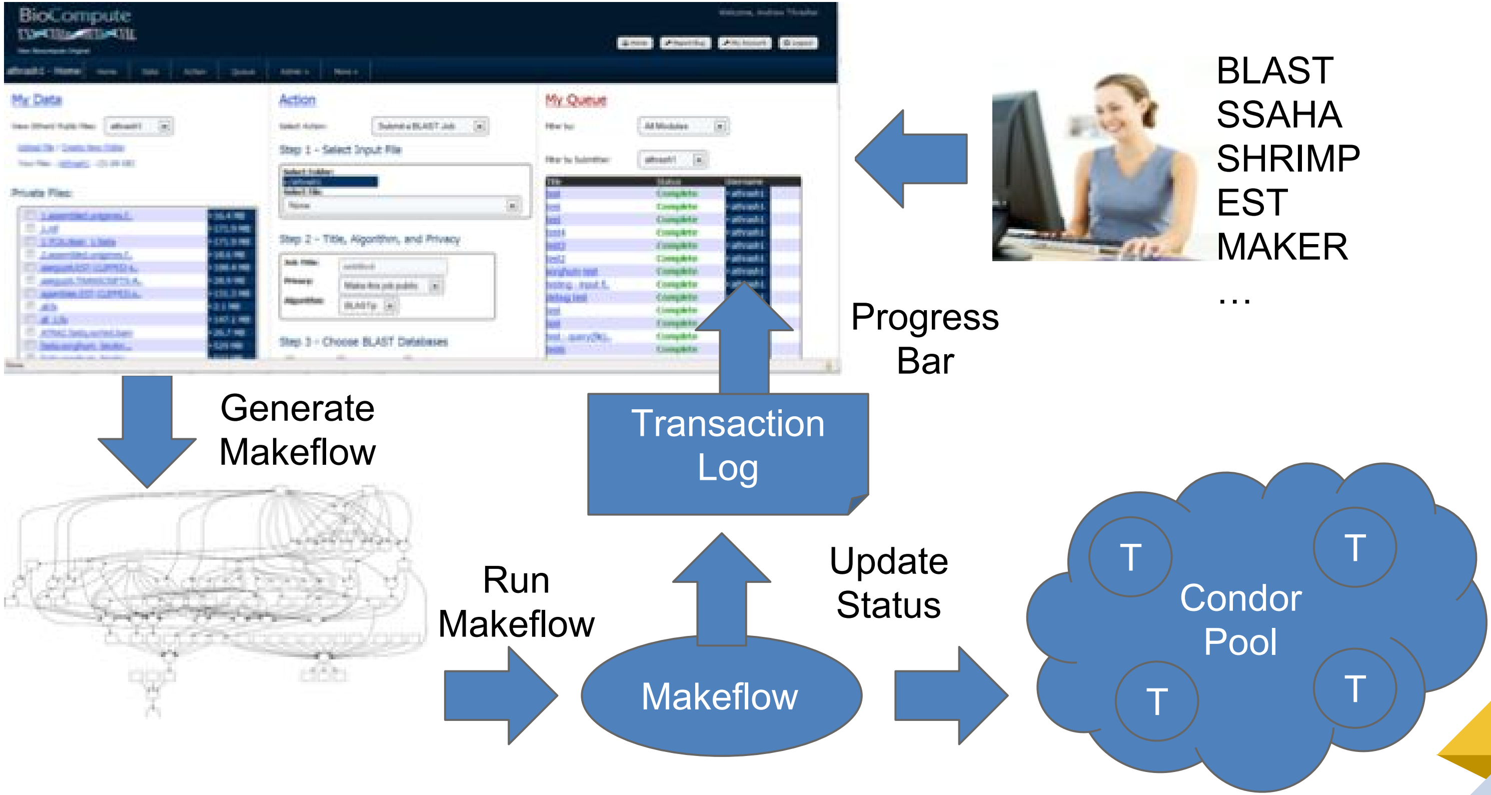
Thousands of Nodes

Transaction Log

Precise Cleanup

Performance Monitoring

Example: Biocompute Portal



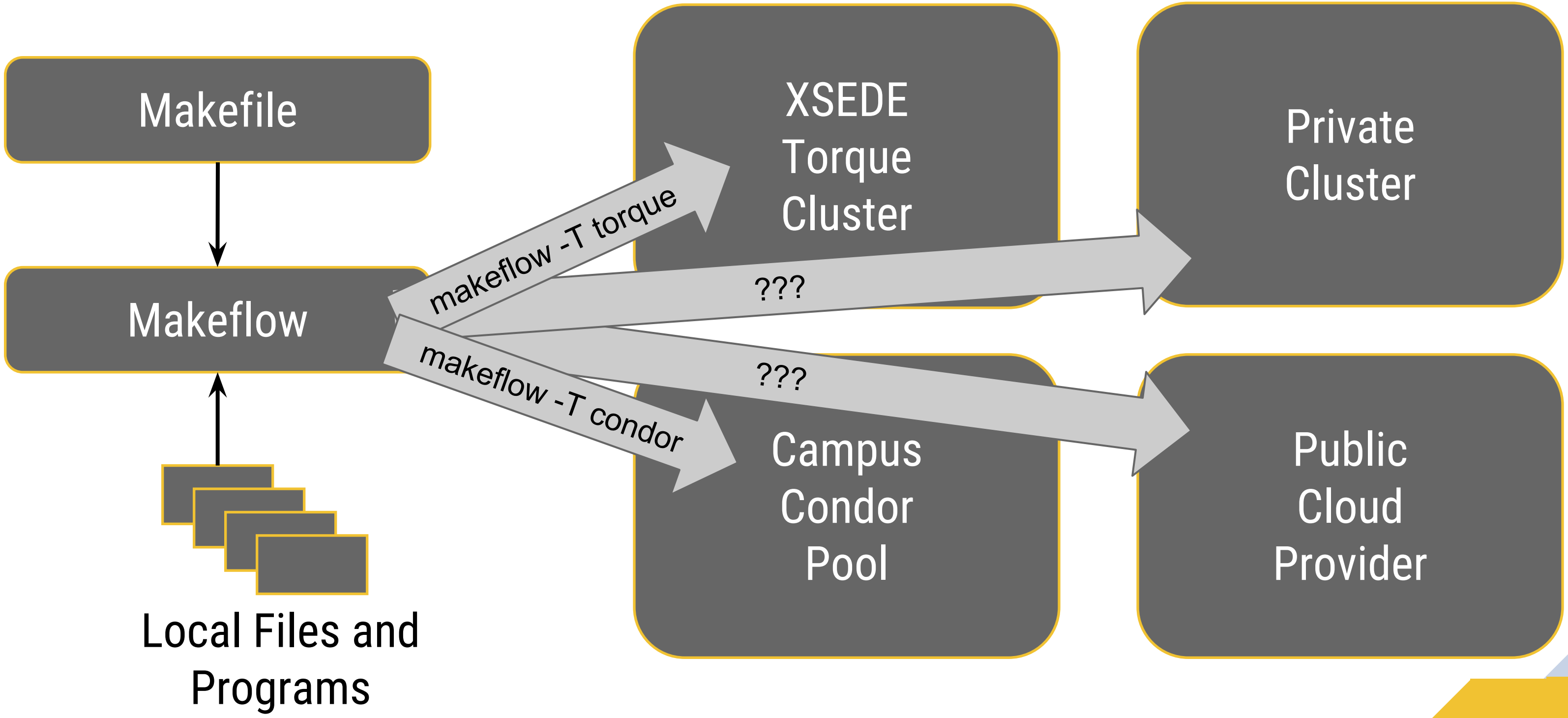
Makeflow Applications

- Bioinformatics
- Biometrics
- High Energy Physics

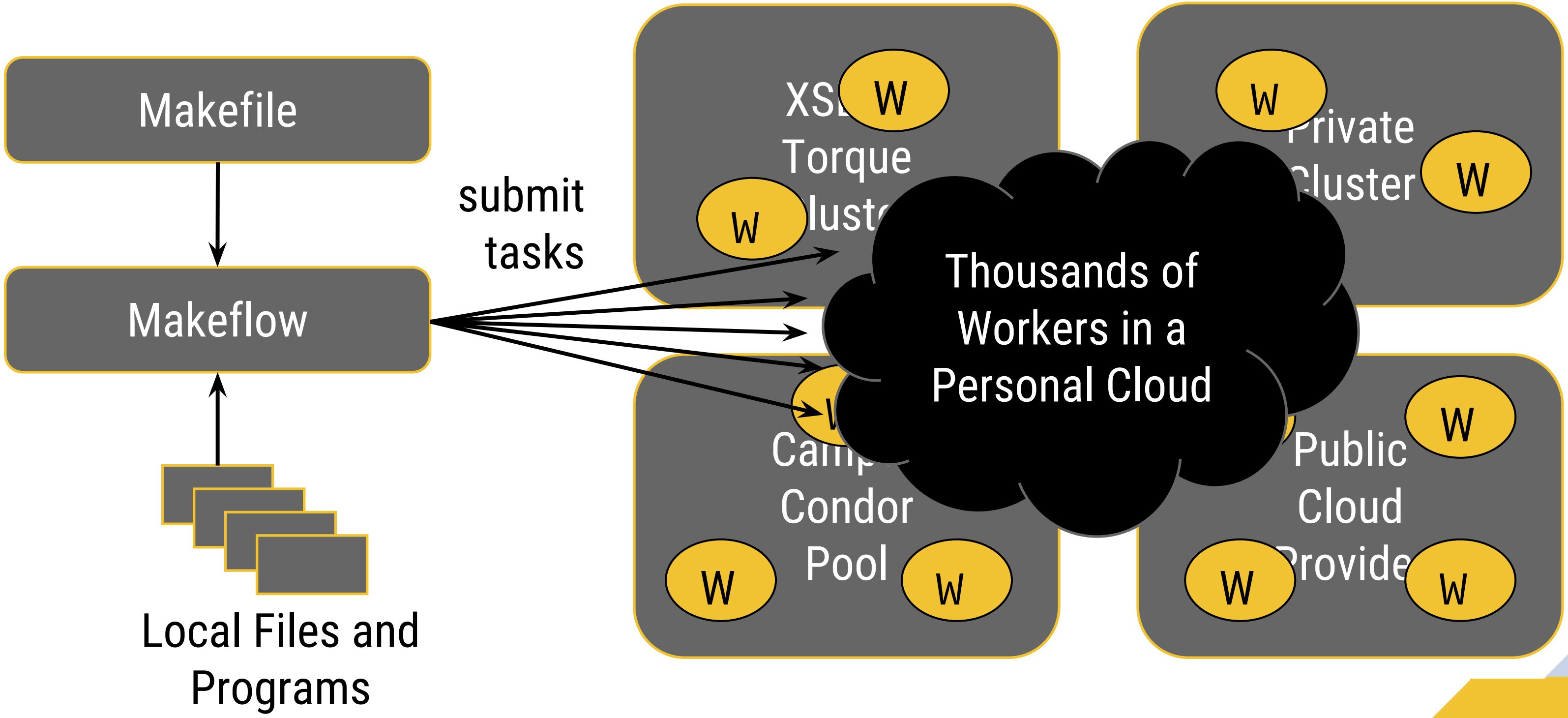
Makeflow + Work Queue

A Portable Workflow System

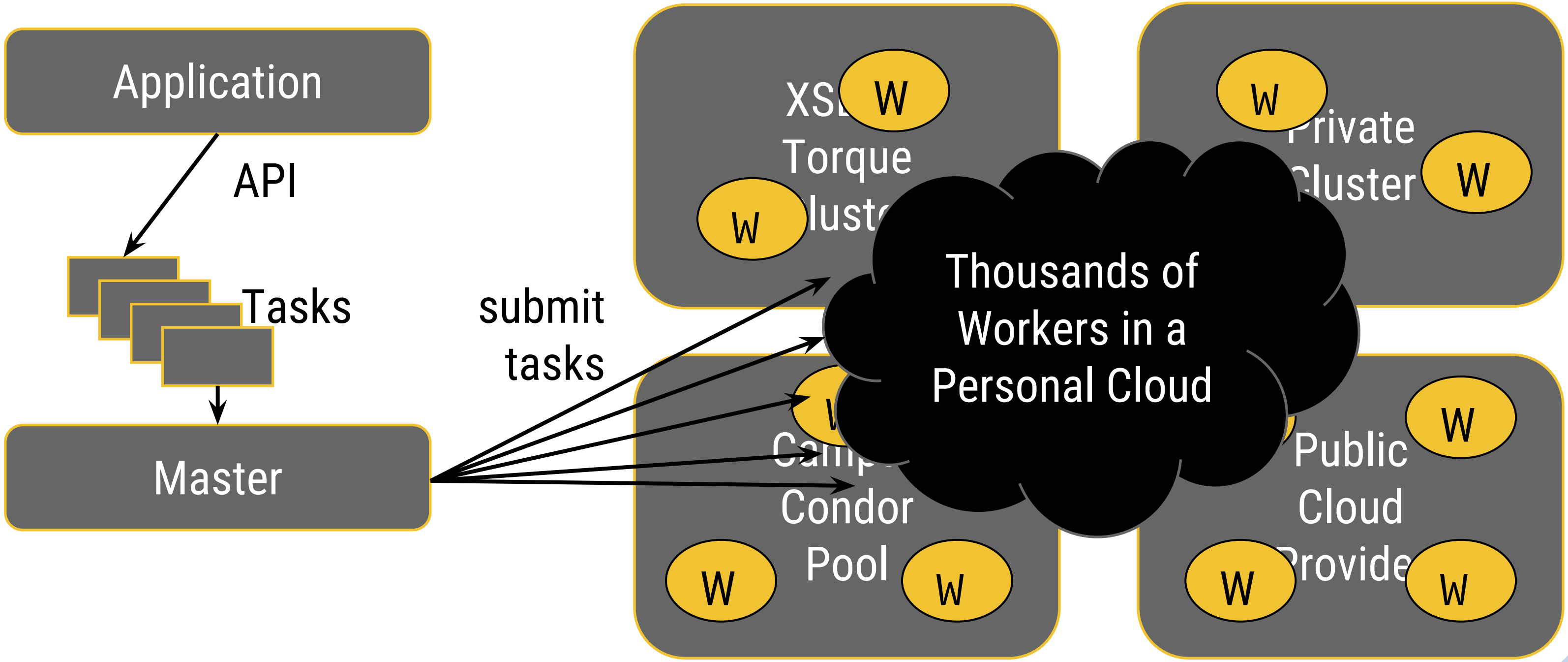
MAKEFLOW


Makefile
Makeflow
Local Files and Programs
makeflow -T torque
???
makeflow -T condor
???
XSEDE Torque Cluster
Private Cluster
Campus Condor Pool
Public Cloud Provider

MAKEFLOW + WORK QUEUE



WORK QUEUE


Application
API
Tasks
Master
submit
tasks
XSEDE
Torque
Cluster
Campus
Condor
Pool
Private
Cluster
Public
Cloud
Provider
W
Thousands of
Workers in a
Personal Cloud

Advantages of Work Queue

- Harness multiple resources simultaneously.
- Hold on to cluster nodes to execute multiple tasks rapidly.
  - (ms/task instead of min/task)
- Scale resources up and down as needed.
- Better management of data, with local caching for data intensive tasks.
- Matching of tasks to nodes with data.

Makeflow and Work Queue

To start the Makeflow

```
% makeflow –T wq  sims.mf
Could not create work queue on port 9123.

% makeflow –T wq –p 0 sims.mf
Listening for workers on port 8374…
```

To start one worker:

```
% work_queue_worker  master.hostname.org 8374
```

Start 25 Workers in Batch System

Submit workers to Condor:

```
condor_submit_workers master.hostname.org 8374 25
```

Submit workers to SGE:

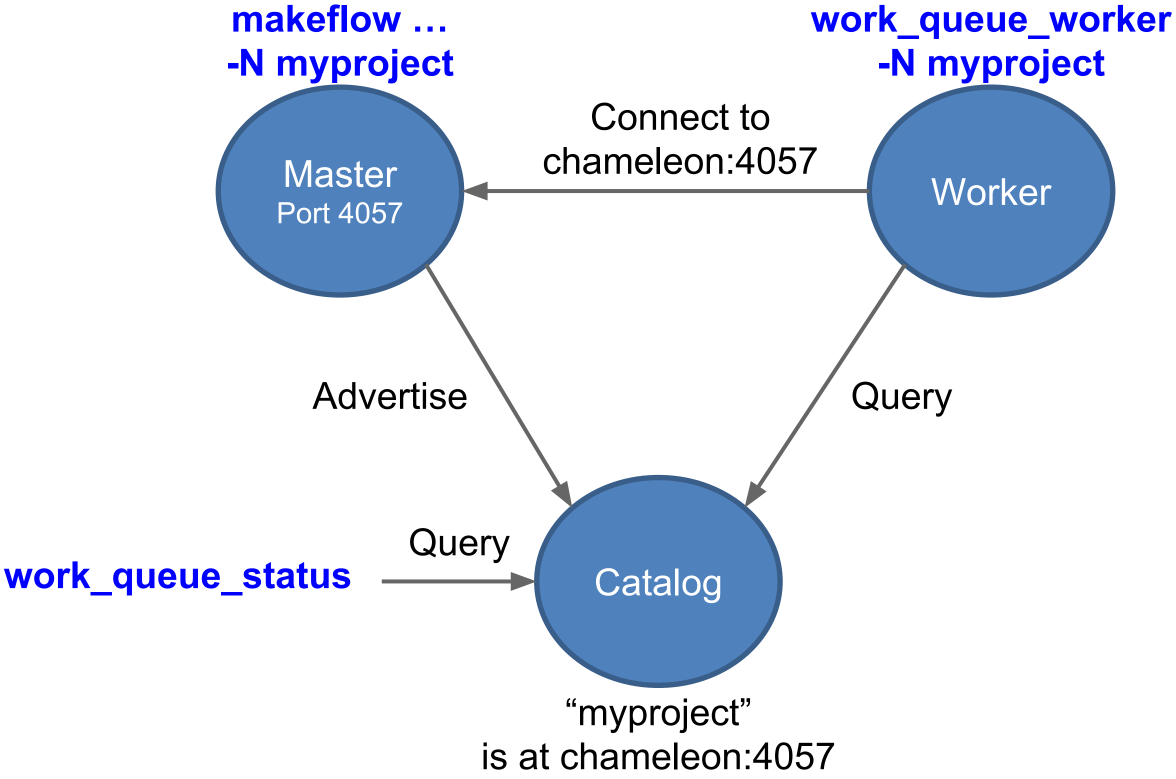
```
sge_submit_workers master.hostname.org 8374 25
```

Submit workers to Torque:

```
torque_submit_workers master.hostname.org 8374 25
```

Keeping track of port numbers gets old fast…

Project Names



Project Names

Start Makeflow with a project name:

```
% makeflow –T wq –N myproject  sims.mf
Listening for workers on port XYZ…
```

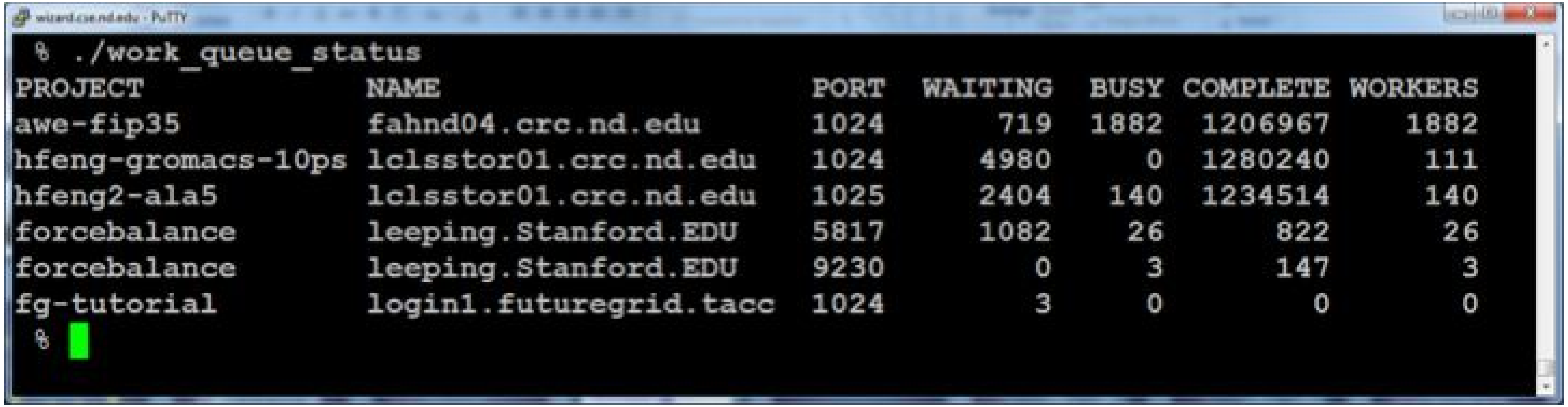
Start one worker:

```
% work_queue_worker -N myproject
```

Start many workers:

```
% torque_submit_workers –N myproject  5
```

work_queue_status



```
 % ./work_queue_status
PROJECT            NAME                       PORT  WAITING  BUSY COMPLETE WORKERS
awe-fip35          fahnd04.crc.nd.edu         1024      719  1882  1206967    1882
hfeng-gromacs-10ps lclsstor01.crc.nd.edu      1024     4980     0  1280240     111
hfeng2-ala5        lclsstor01.crc.nd.edu      1025     2404   140  1234514     140
forcebalance       leeping.Stanford.EDU       5817     1082    26      822      26
forcebalance       leeping.Stanford.EDU       9230        0     3      147       3
fg-tutorial        login1.futuregrid.tacc     1024        3     0        0       0
 %
```

Advantages of Work Queue

- MF +WQ is fault tolerant in many different ways:
  - If Makeflow crashes (or is killed) at any point, it will recover by reading the transaction log and continue where it left off.
  - Makeflow keeps statistics on both network and task performance, so that excessively bad workers are avoided.
  - If a worker crashes, the master detects failure and restarts the task elsewhere.
  - Workers can be added and removed at any time during workflow execution.
  - Multiple masters with the same project name can be added and removed while the workers remain.
  - If the worker sits idle for too long (default 15m) it will exit, so as not to hold resources idle.

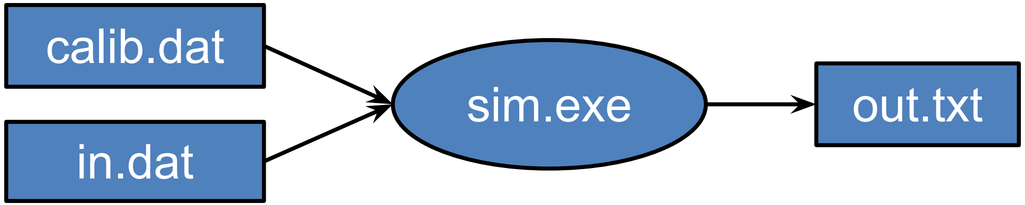
Alternative Makeflow Formats

Utilizing JSON/JX for easier scripting

Makeflow JSON Syntax

- Verbose flexible structure
- Familiar structure
- Consists of four items:
  - "categories": Object<Category>
  - "default_category": String
  - "environment": Object<String>
  - "rules": Array<Rule>

Makeflow JSON Syntax



```
sim.exe in.dat –p 50 > out.txt
```

```
{
        "outputs": [{"path": "out.txt"}],
        "inputs": [ {"path": "in.dat"}, {"path": "calib.dat"}, {"path": "sim.exe"}]
        "command": "sim.exe –p 50 in.data > out.txt",
}
```

Makeflow JSON Syntax

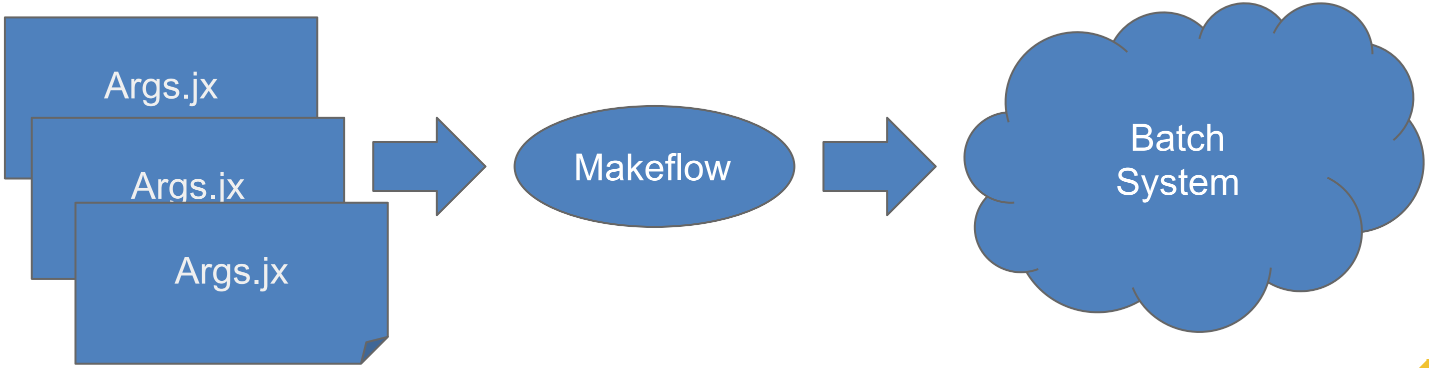
```
{
        "outputs": [{"path": "out_10.txt"}],
        "inputs": [ {"path": "in.dat"},  {"path": "calib.dat"},  {"path": "sim.exe"}]
        "command": "sim.exe –p 10 in.data > out_10.txt",
},
{
        "outputs": [{"path": "out_20.txt"}],
        "inputs": [ {"path": "in.dat"},  {"path": "calib.dat"},  {"path": "sim.exe"}]
        "command": "sim.exe –p 20 in.data > out_20.txt",
},...
```

Makeflow JSON Rule

- "inputs": Array<File>
- "outputs": Array<File>
- "command": String
- "local_job": Boolean
- "category": String
- "resources": Resources
- "allocation": String
- "environment": Object<String>

Makeflow JX Syntax

- Allows for more compact makeflows.
  - Provides functions for expanding tasks: range, variables, etc...
- Can be used as templates in conjunction with an arguments file.
- Useful for consistently structure data and different data.



Makeflow JX Syntax

```
{
        "outputs": [{"path": format("out_%d.txt", i)}],
        "inputs": [ {"path": "in.dat"},  {"path": "calib.dat"},  {"path": "sim.exe"}]
        "command": format("sim.exe –p %d in.data > out_%d.txt", i),
} for i in range(10, 30, 10),
```

How to run a Makeflow

- Run a workflow from json
  - makeflow --json sims.json
- Clean up the workflow outputs:
  - makeflow –c --json sims.json
- Run the workflow from jx:
  - makeflow --jx sims.jx
- Run the workflow with jx and args:
  - makeflow --jx sims.jx --jx-args args.jx

Resource Management

Allowing tasks to share resources

Why Manage Resource?

- More accurate accounting and provisioning.
- Allows for multi-tenant situations.
- Provides consistent resources to tasks.
  - Prevents slower execution.
  - Mitigate failures from under provisioning.

How can this happen?

Makeflow Resource Specification

- Category
  - Cores
  - Memory
  - Disk

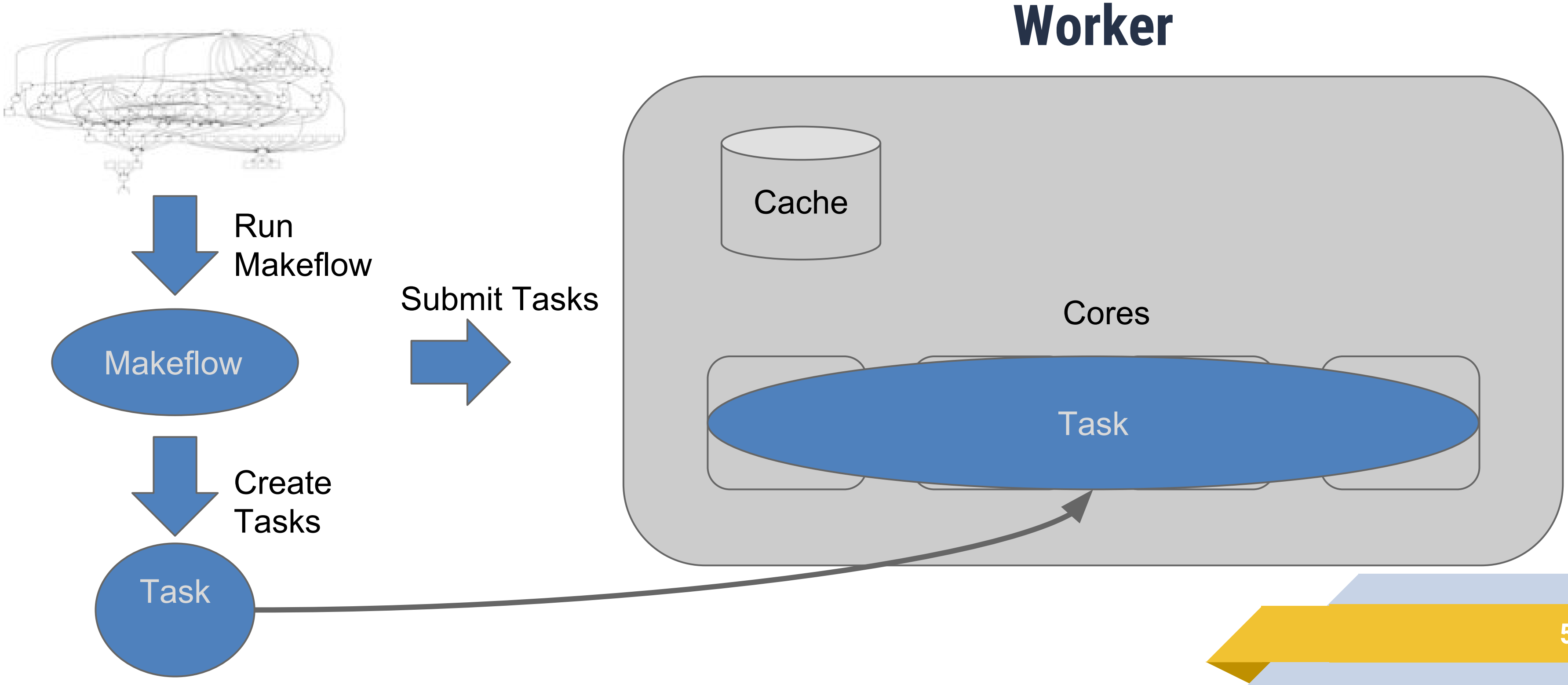
```
…
CATEGORY=analysis
DISK=1024
MEMORY=1024
CORES=1

out1: part1 mysim.exe
    ./mysim.exe part1 >out1

out2: part2 mysim.exe
    ./mysim.exe part2 >out2

...
```

Makeflow Resource Specification

- Category
  - Cores
  - Memory
  - Disk

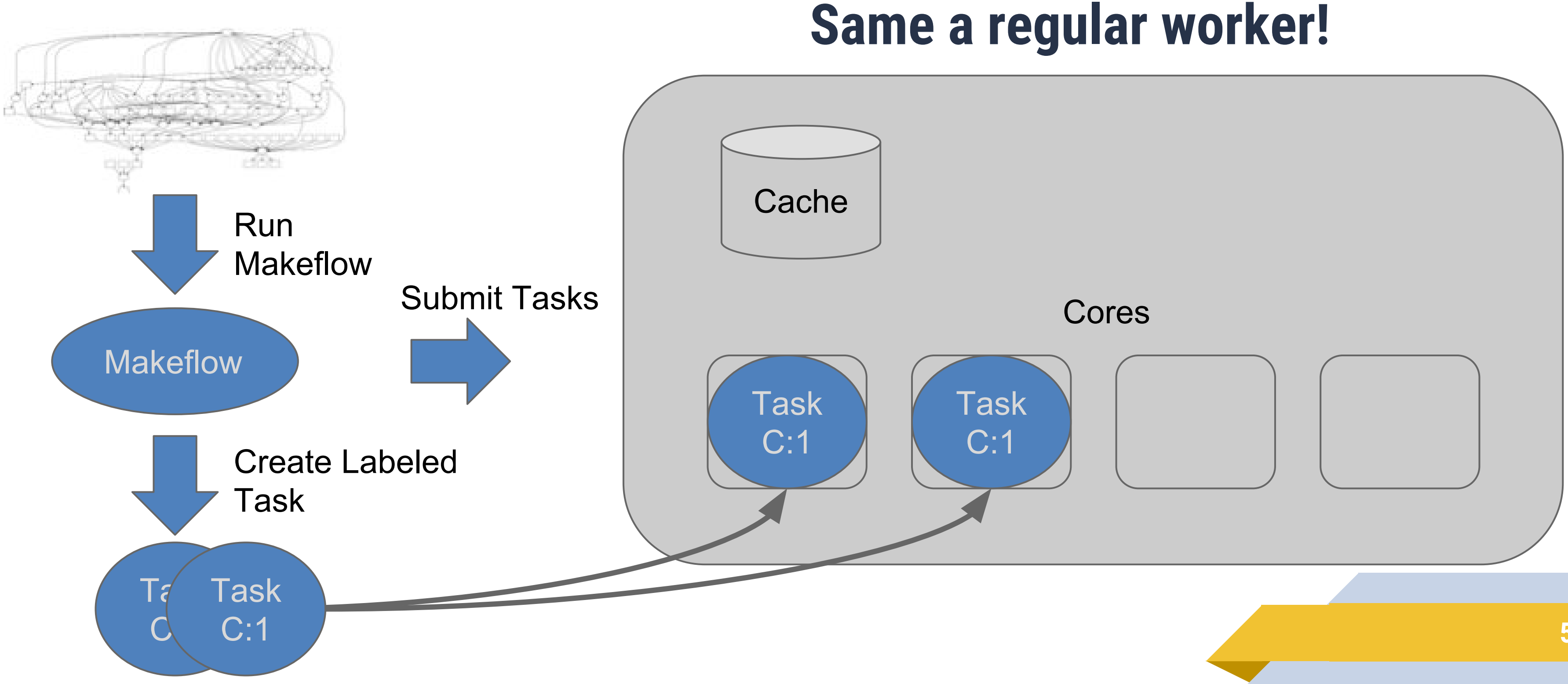
```
…
CATEGORY=analysis
DISK=1024
MEMORY=1024
CORES=1

out1: part1 mysim.exe
    ./mysim.exe part1 >out1
…

CATEGORY=join
DISK=2048
MEMORY=2048
CORES=2

result: out1 out2 out3 join.py
    ./join.py out1 out2 out3 > result
```

Work Queue Workers



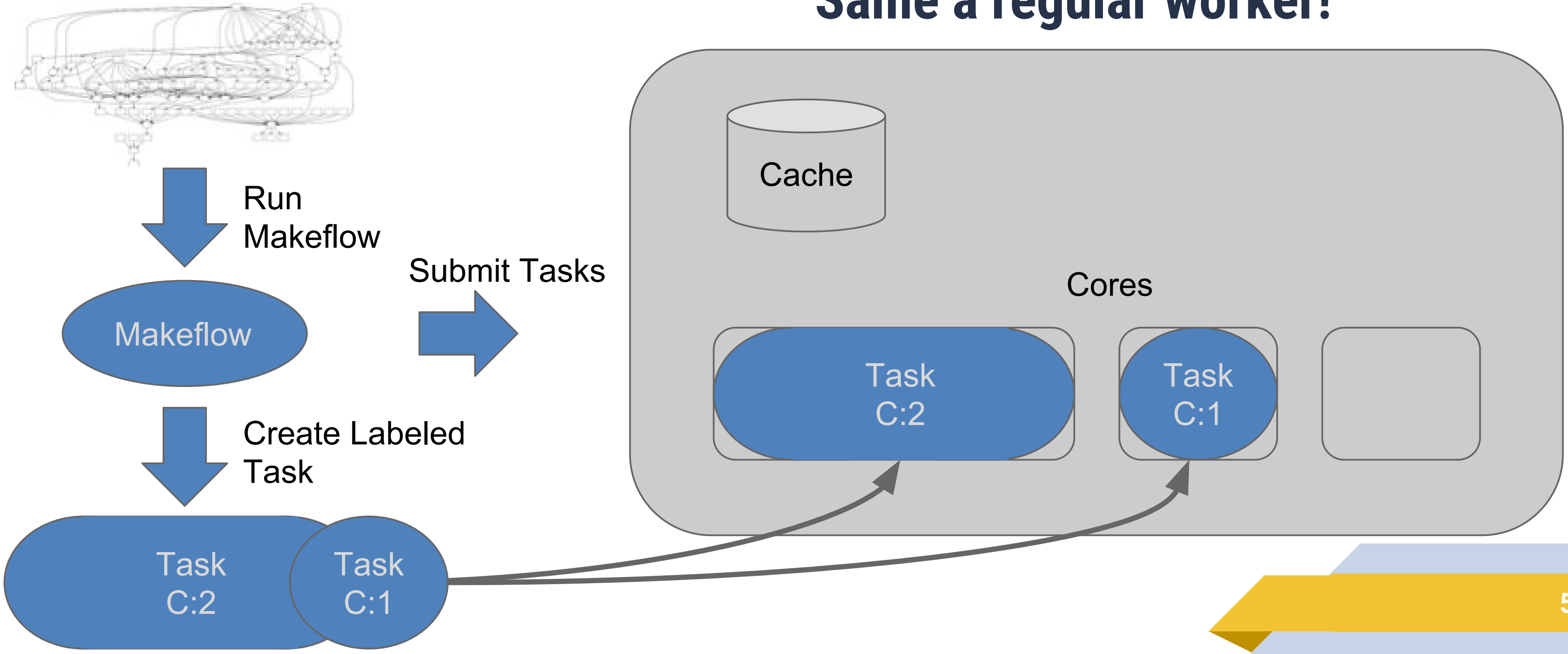
Work Queue Multi-tenant Workers

Same a regular worker!



Work Queue Multi-tenant Workers

Same a regular worker!



Resource Monitor

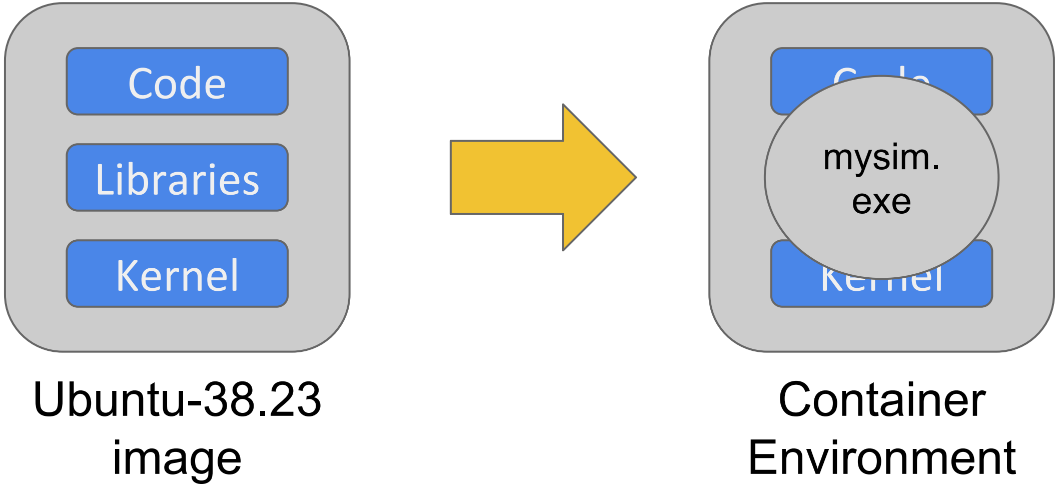
- Watches process to ensure correct resource usage
- Evict jobs that act outside of resource allocation
- Report actual usage for future calibration
- Can be used in conjunction with Makeflow to automate an accurate image size.

Container Integration

Providing consistent environments

Containers Create Precise Execution Environments

```
docker run ubuntu-38.23 mysim.exe
```

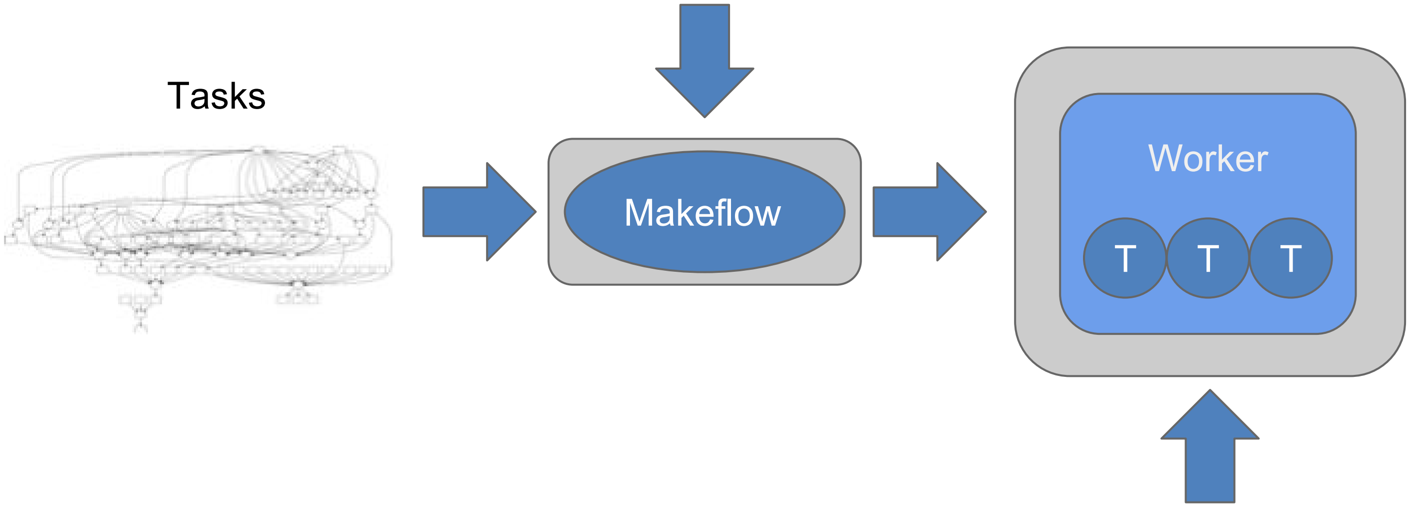


Approaches to Containers with Makeflow

- Approach 1:
  - Create containers for starting MF and WQ, then let them run as normal.
  - You are responsible for moving container images responsibly.
- Approach 2:
  - Let MF create containers as needed for each task.
  - Provides more control over moving container images.
  - Sending and staring up containers for each task.

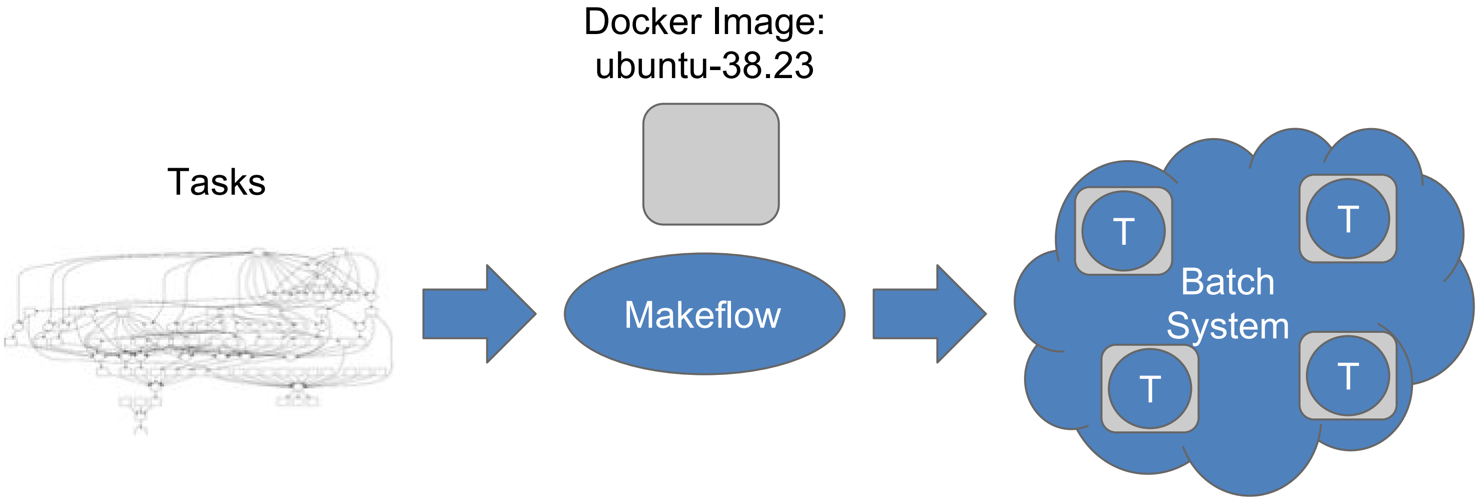
Approach 1: Container for MF/WQ

```
docker run ubuntu makeflow
```



```
docker run ubuntu work_queue_worker
```

Approach 2: Container for Each Task



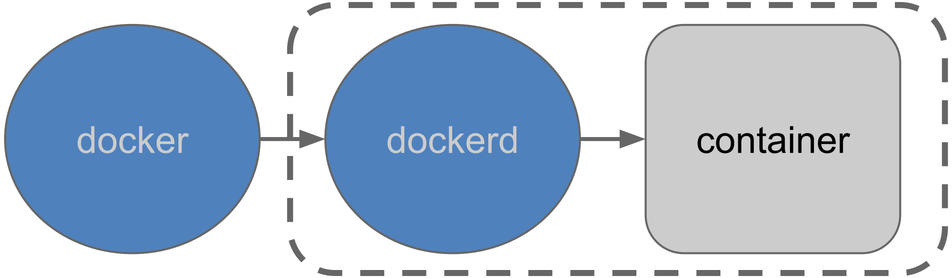
```
makeflow --docker ubuntu-38.23 –T sge . . .
```

Container Technology is Evolving

docker.io

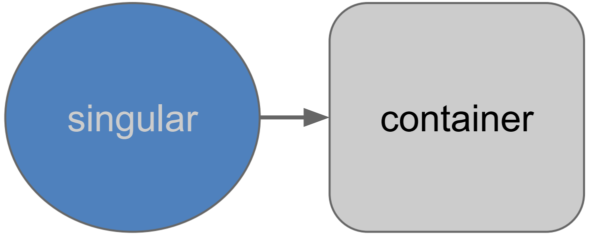
singularity.lbl.gov

docker run ubuntu command



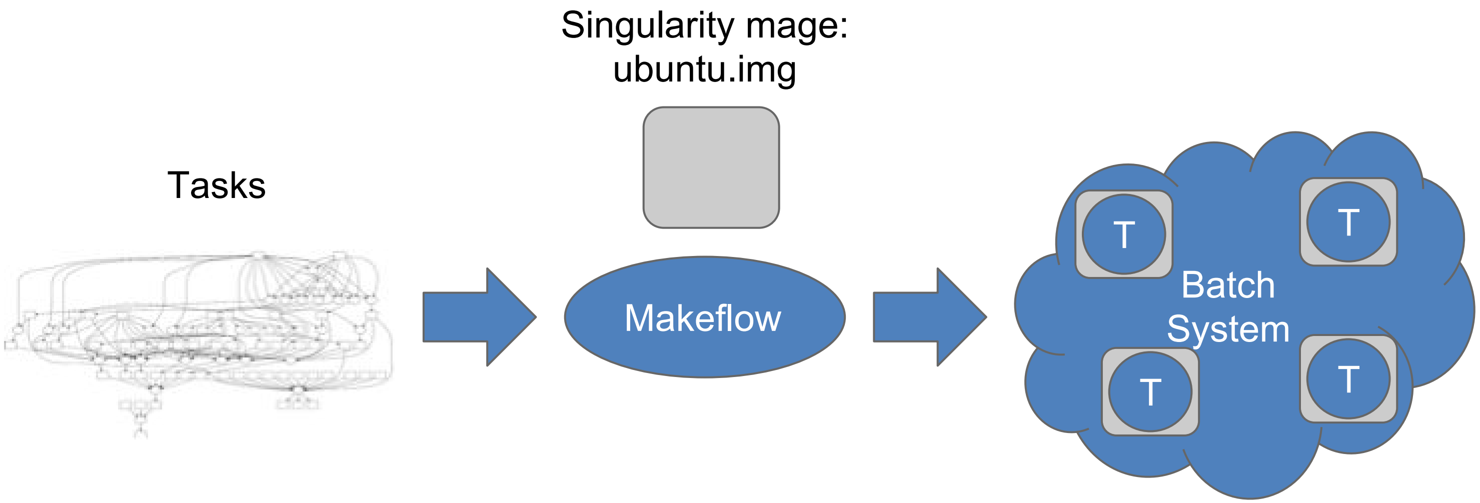
Installed service running as root

singularity exec ubuntu command



Container runs directly as a child process
(still needs setuid tool, though)

Approach 2 using Singularity



makeflow --singularity ubuntu.img –T sge . . .

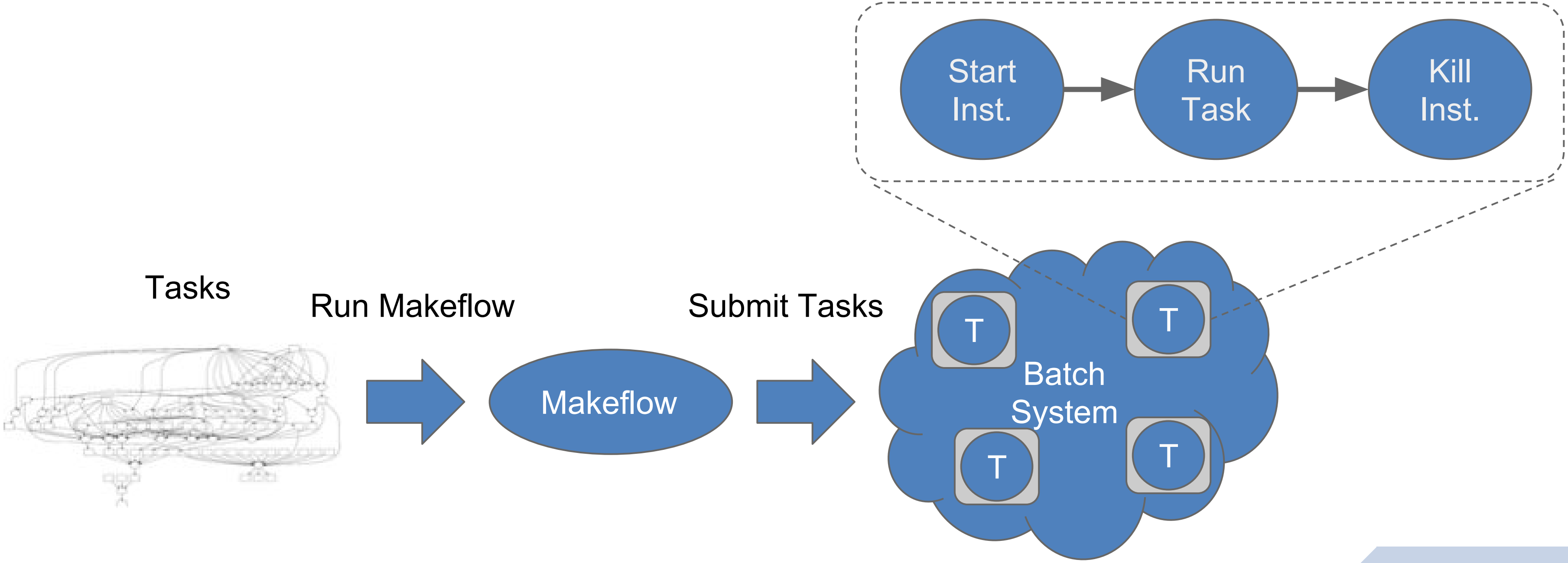
Cloud Operation

Methods to Deploying

Approaches to Cloud Provisioning with Makeflow

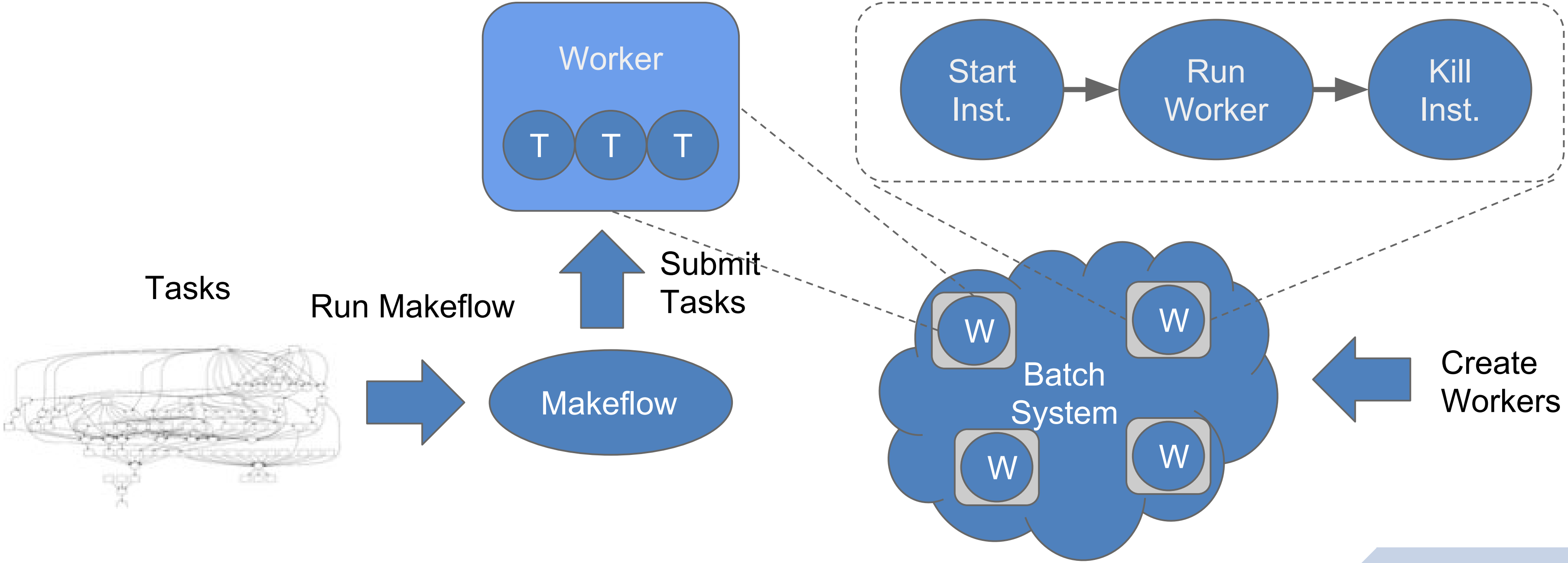
- Approach 1:
  - MF creates unique instance for each task.
  - Provides complete isolation between tasks.
  - Requires startup and tear-down time of instances.
- Approach 2:
  - Create instances and run WQ Workers on them, submitting to WQ from MF.
  - Relies on WQ for task isolation, but caches shared files.
  - Instance management relies on the user.

Approach 1: Individual instances per task



```
makeflow -T amazon --amazon-config my.config ...
```

Approach 2: Individual instances per worker



```
work_queue_factory -T amazon --amazon-config my.config
...
```

Questions?

Nick Hazekamp
Email : nhazekam@nd.edu

CCL Home : http://ccl.cse.nd.edu
Tutorial Link : http://ccl.cse.nd.edu/software/tutorials/acic17