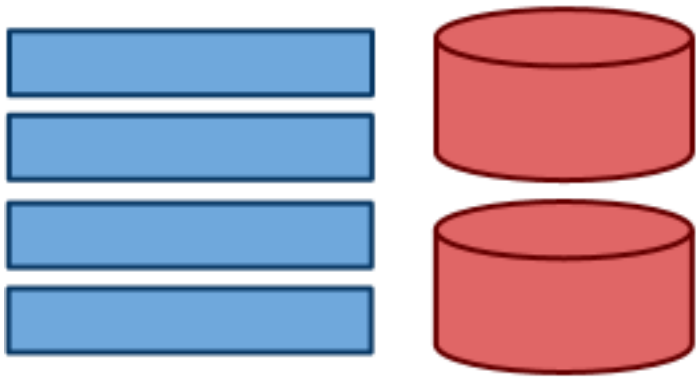
Work Queue + Python

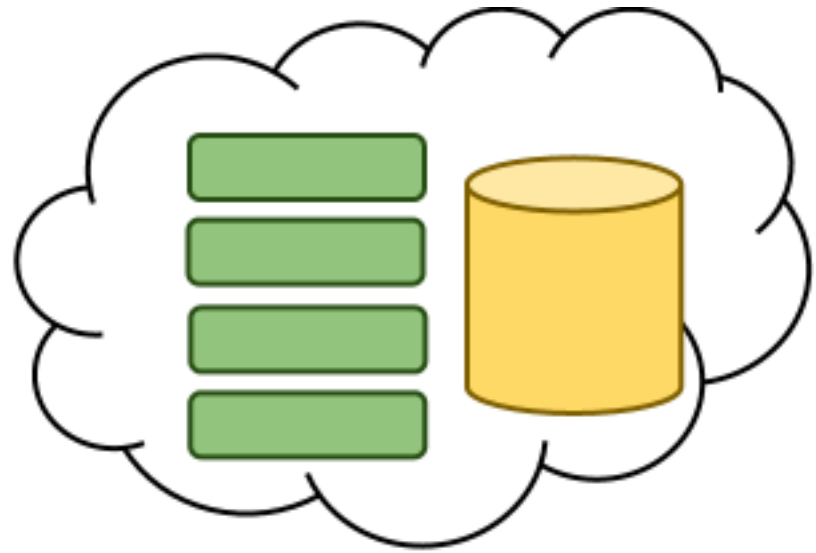
A Framework For Scalable Scientific Ensemble Applications

Peter Bui, Dinesh Rajan, Badi Abdul-Wahid,
Jesus Izaguirre, Douglas Thain
University of Notre Dame

Distributed Computing



Campus Grid



Cloud Platform

Examples

- Condor cluster
- SGE grid
- Beowulf cluster

Examples

amazon web services™

Windows Azure

Programming Challenges

Resource Management

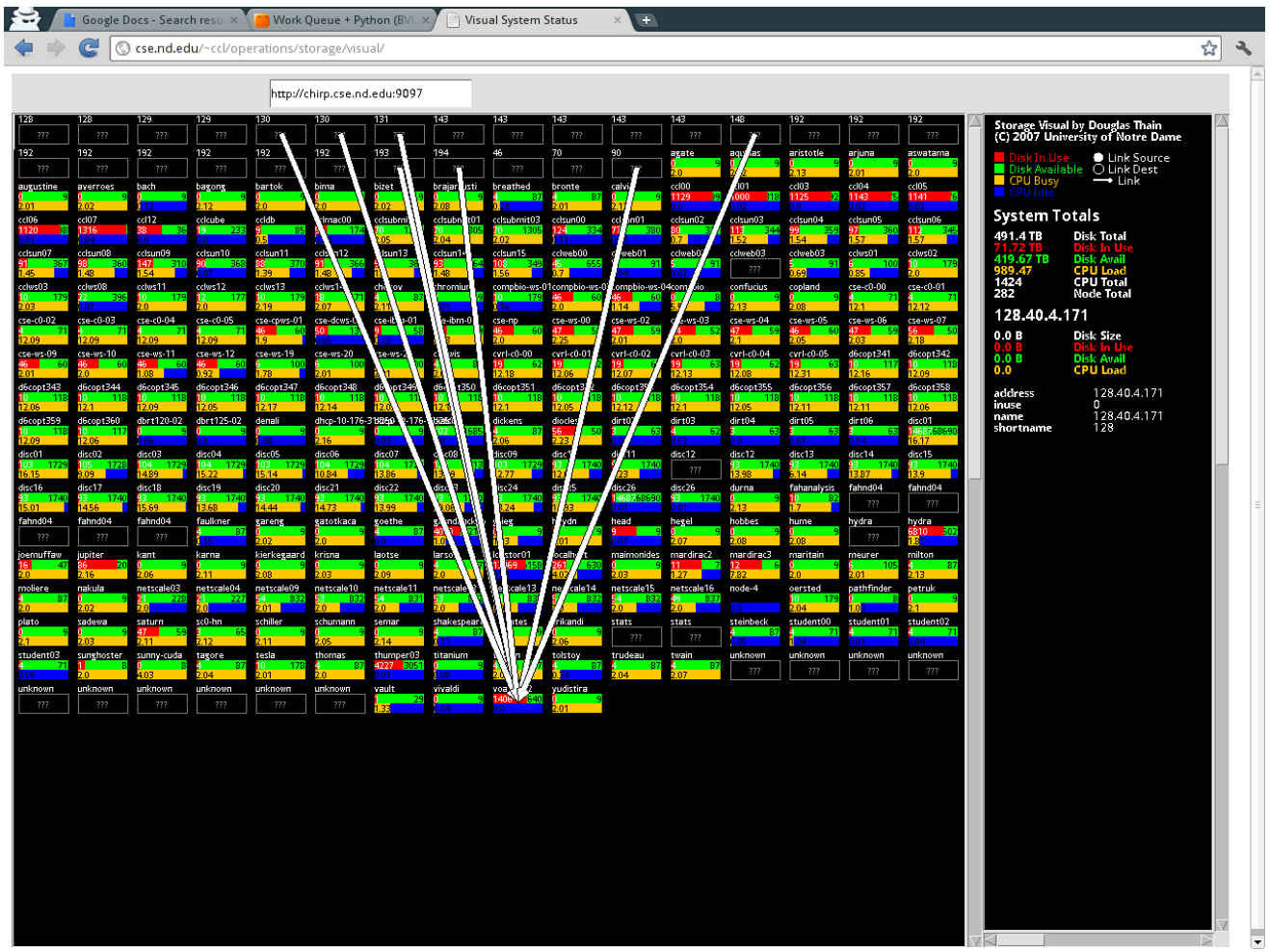
- Storage
- CPUs
- Network

Scheduling

- Packaging
- Deployment
- Task dispatch

Fault tolerance

- Nodes die
- Jobs fail
- Network problems



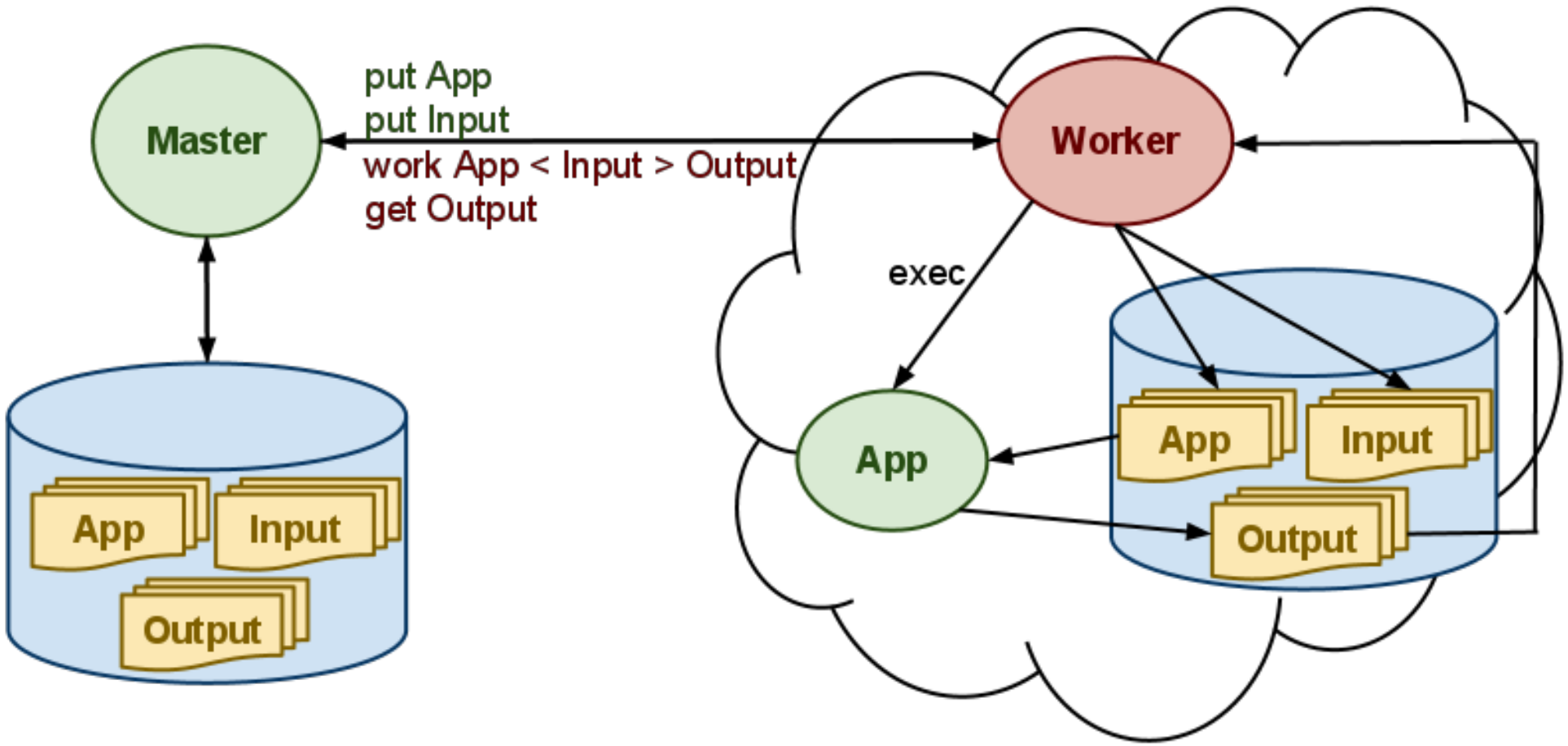
Work Queue

A flexible **master/worker** framework for building large scale scientific **ensemble** applications that span many machines including **clusters, grids, and clouds.**

Features

- Data management
- Fault tolerance
- Scheduling
- Fast abort
- Flexible worker deployment
- Catalog discovery service

Master/Worker Model



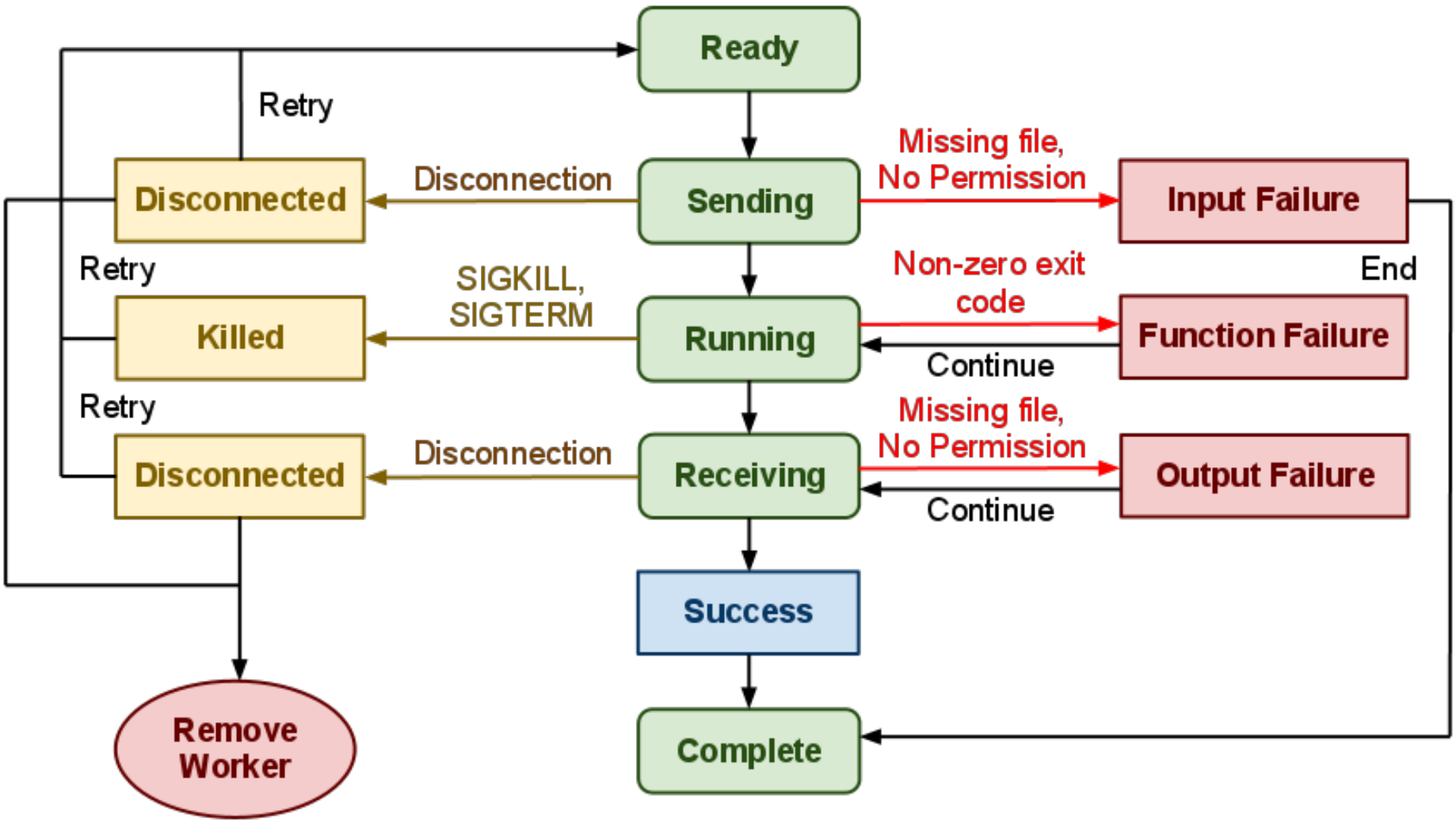
Central Master

- Divides work into tasks
- Sends tasks to Workers
- Gathers results

Pool of Workers

- Receive input and executable files
- Execute task command
- Return output files

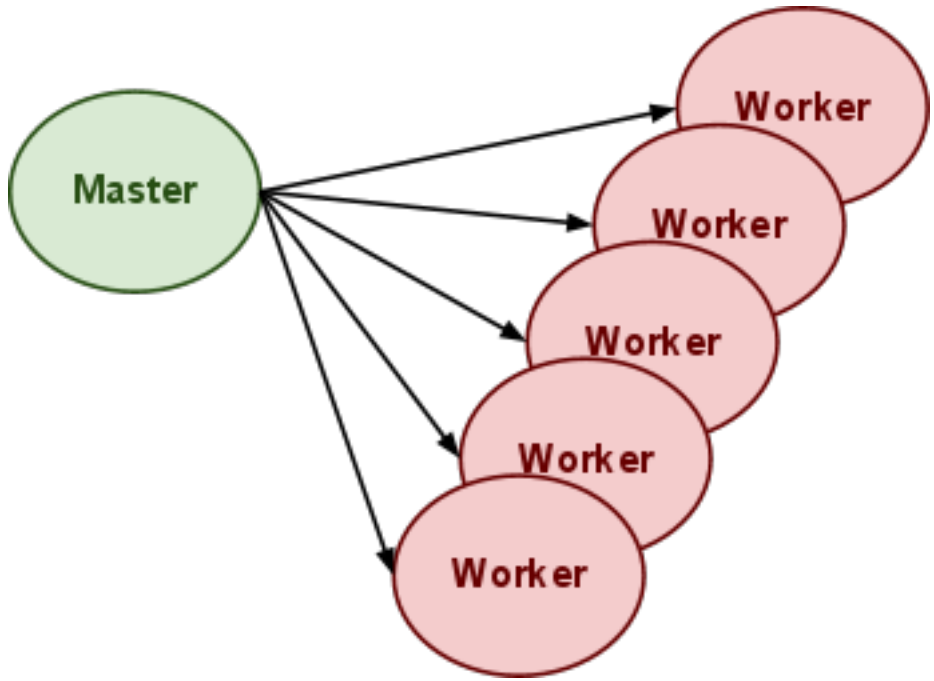
Work Queue: Data Management, Fault Tolerance



Work Queue: Scheduling, Fast Abort

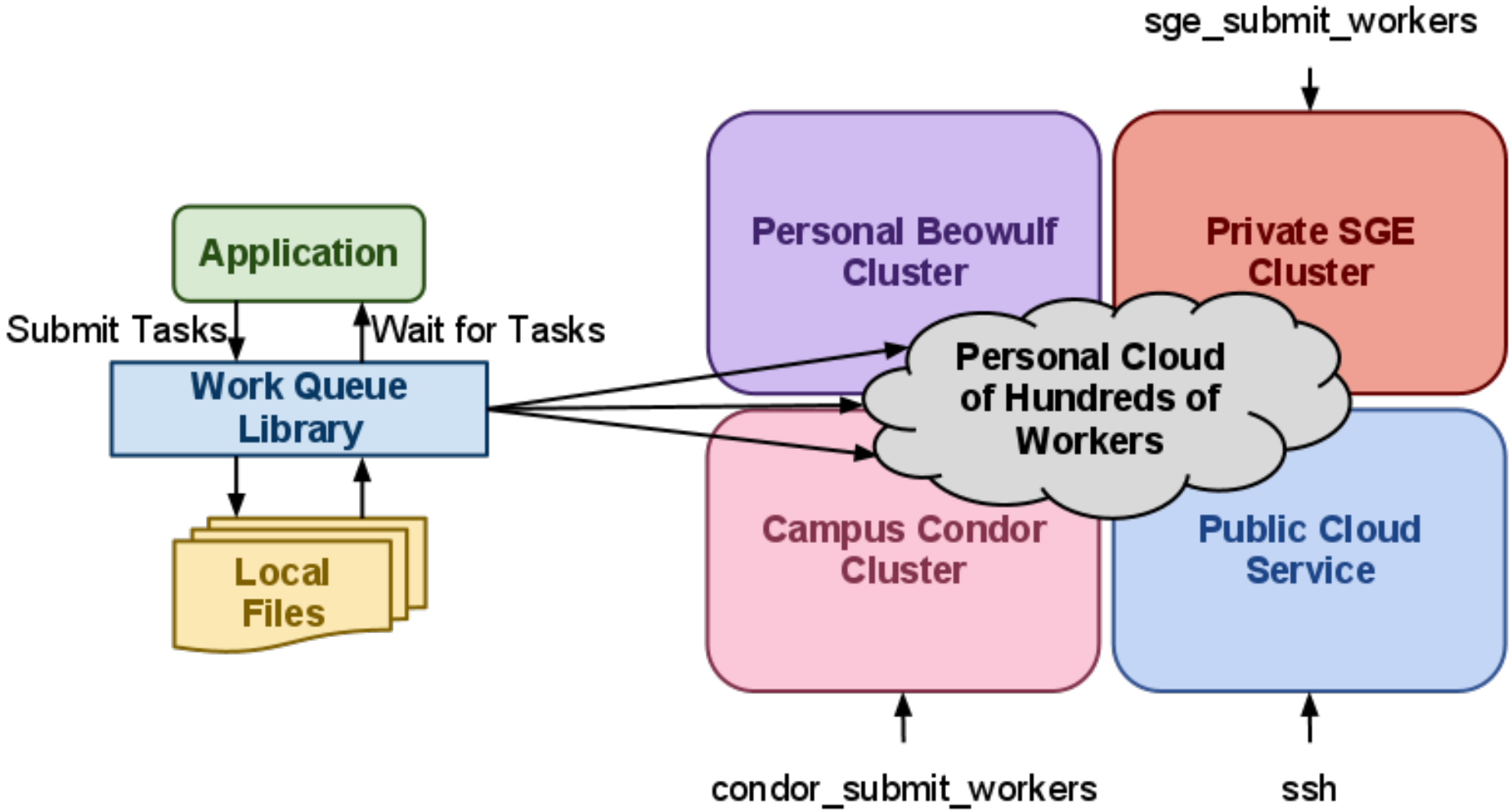
Provides multiple algorithms for assigning tasks to workers:

1. **First Come First Serve**
2. **Cached Files**
3. **Fastest Time**
4. **Preferred Hosts**
5. **Random**



To prevent stragglers, collect statistics, and perform ***fast abort*** on slow workers.

Work Queue: Worker Deployment, Architecture



Work Queue + Python

- Library is written in **C** and provides a straightforward API
  - **C** is a low-level language
  - Domain scientists familiar with scripting languages
- Provide **Python** bindings to library
  - Initially hand-written, but switched to SWIG
  - Allow scientists to high-level language
  - Access to large community and ecosystem of third-party software

Python-WorkQueue Module

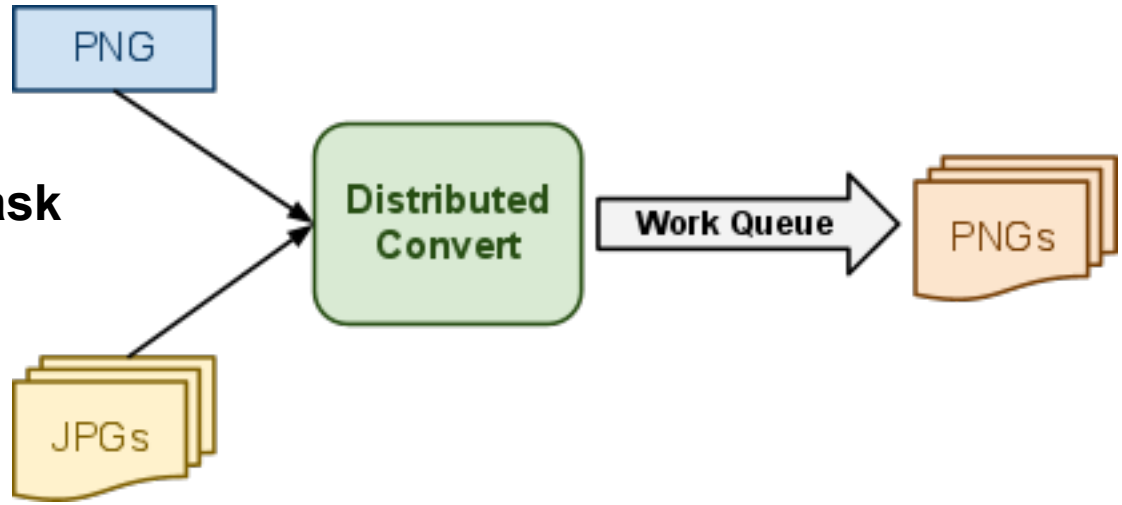
WorkQueue

```
# Import work_queue module
from work_queue import WorkQueue

# Create master work queue
wq = WorkQueue()

# Set catalog project name
wq.specify_name('project.name')

# Set selection algorithm
wq.specify_algorithm(
   WORK_QUEUE_SCHEDULE_FILES)

# Set fast abort factor
wq.activate_fast_abort(1.5)
```

Task

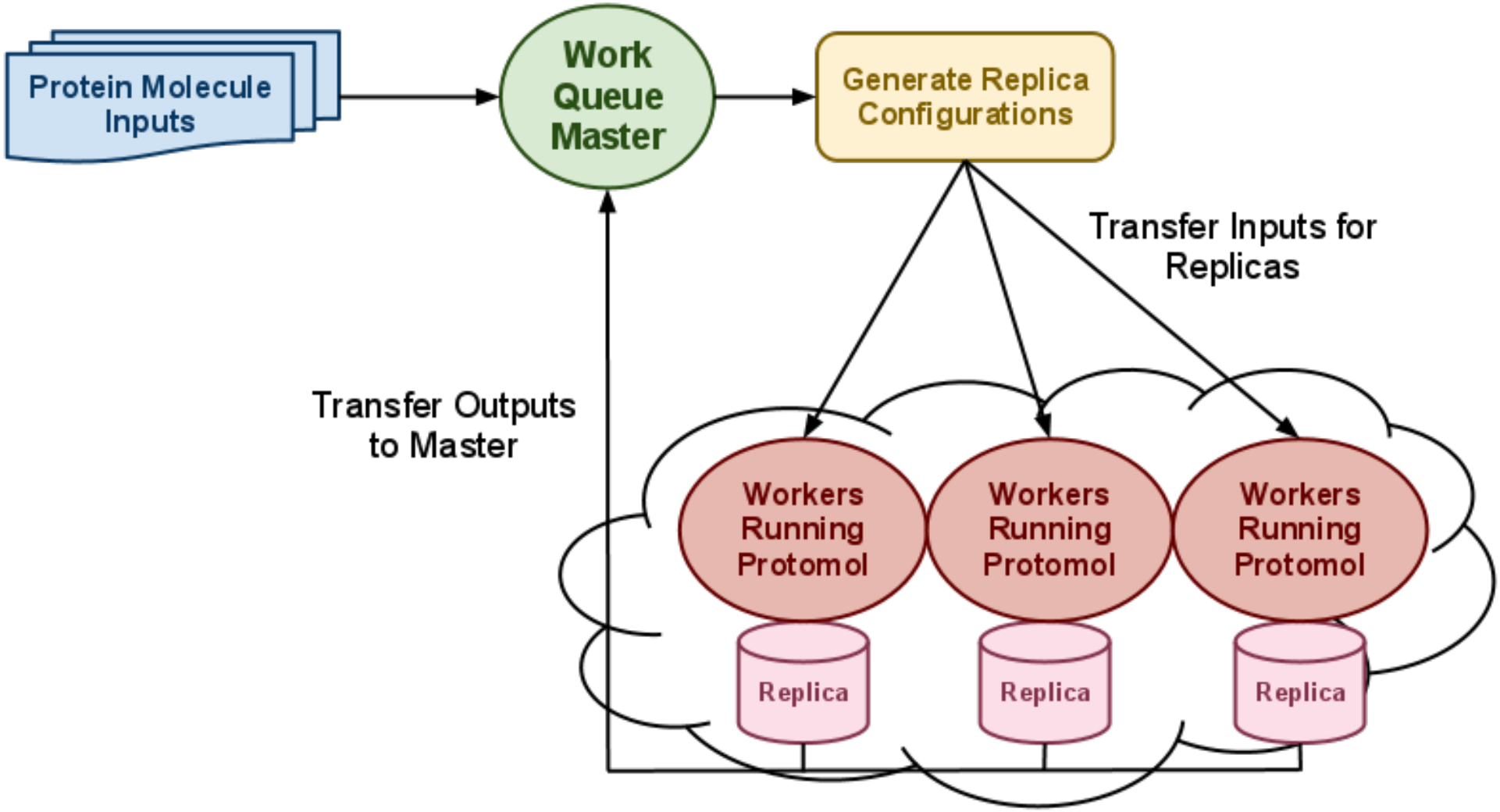
```
# Import work_queue module
from work_queue import Task

# Create task
task = Task('date > output.txt')

# Specify output file
task.specify_output_file('output.txt')

# Submit task to master
wq.submit(task)
```

Example: Distributed Convert

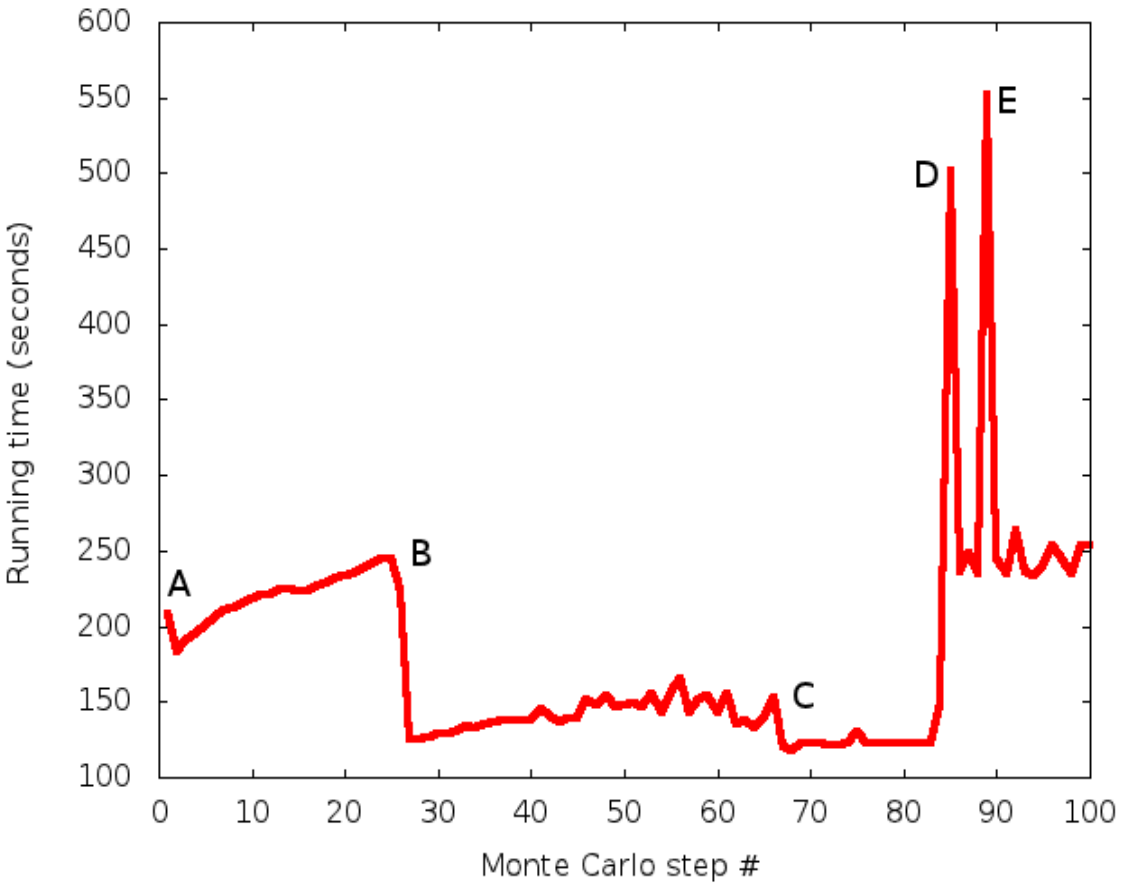



```
from workqueue import WorkQueue, Task
import os, sys

wq = WorkQueue()
output_ext = sys.argv[1]

# For each file, construct & submit a transcoding task
for input_file in sys.argv[2:]:
    output_file  = os.path.splitext(input_file)[0] + '.' + output_ext

    task = Task('convert %s %s' % (input_file, output_file))
    task.specify_input_file(input_file)
    task.specify_output_file(output_file)
    wq.submit(task)

# While workqueue is not empty, poll for task and then print command and result
while not wq.empty():
    task = wq.wait(1)
    if task:
        print task.command, task.result
```

Application: Replica Exchange



Evaluation: Replica Exchange



Events

A: Start 100 SGE workers

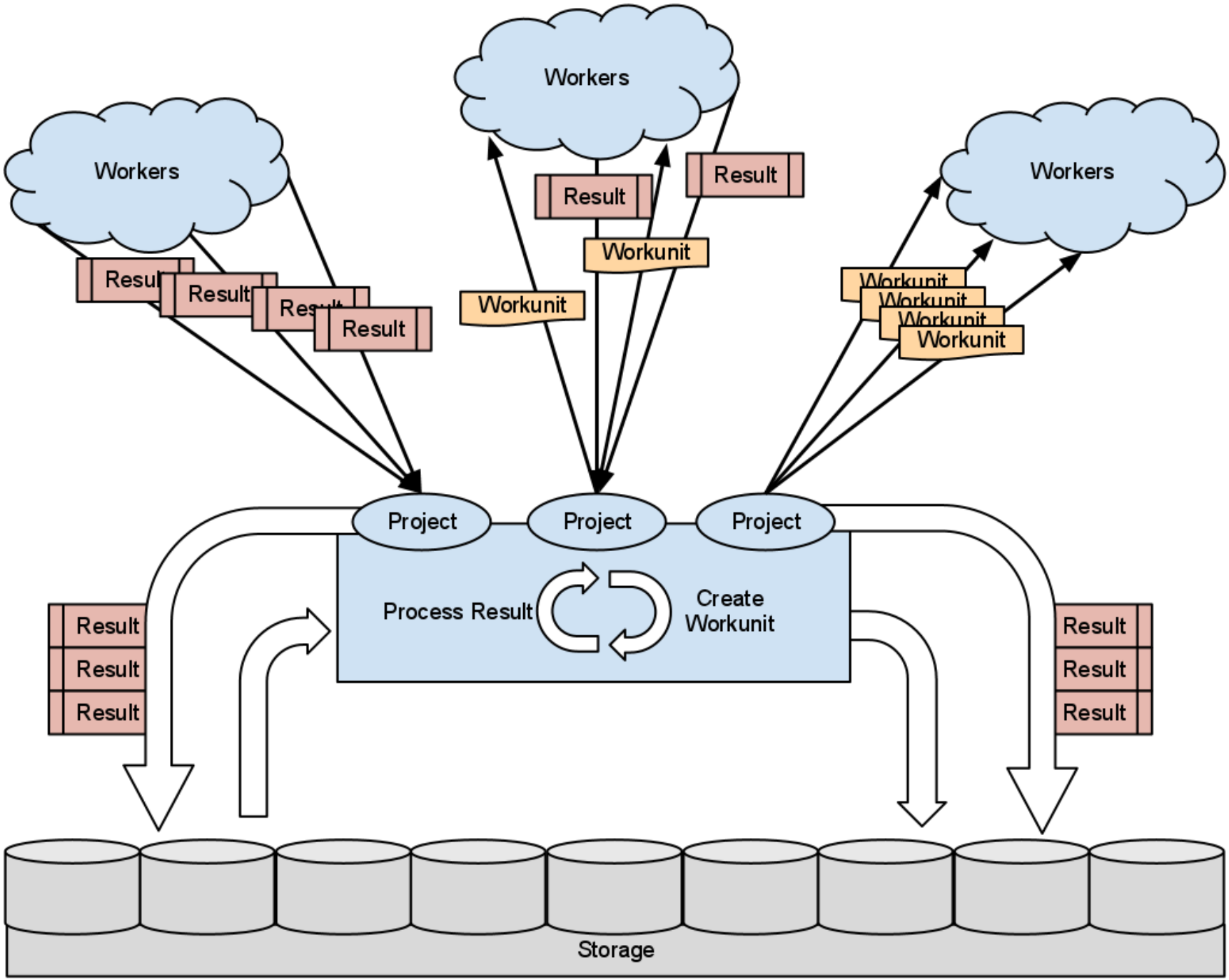
B: Add 150 Condor workers

C: Add 110 Condor and **40 Amazon EC2** workers

D: Remove 100 SGE workers

E: Remove 125 Condor and **25 Amazon EC2** workers

Application: Folding@Work



Evaluation: Folding@Work

Results after One Month

Tasks Assigned	283830
Results received	122141
Simulation time gathered	305 us
Execution time average (min)	125
Execution time std. dev (min)	87
Number of workers	5000
Number of unique machines	370

Represents about 3,000 CPU days of work.

Future Work

- Use SWIG to generate bindings for additional languages (PERL, Lua, etc.)
- Monitoring and visualization software
- Extend Work Queue to better support hierarchical workflows
  - Multiple masters
  - Resource manage and allocation
- Integrate into Programming Paradigms course

Summary

Work Queue is a flexible and powerful framework for constructing scalable scientific ensemble applications.

- Provides data management, fault-tolerance, multiple scheduling algorithms, fast abort, and support for multiple distributed systems.
- With **Python-WorkQueue** module it is now available in a user-friendly language.

Questions?

CCTools Software Download

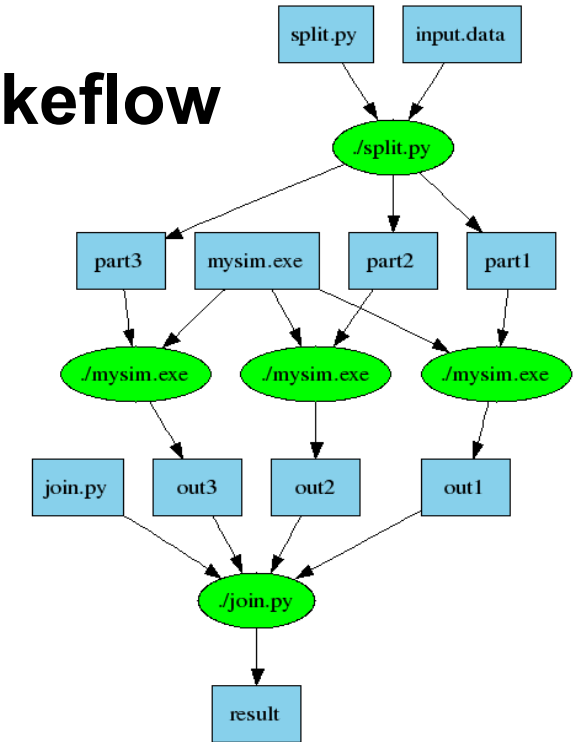
http://cse.nd.edu/~ccl/software

Analysis

- **Work Queue** transparently handles worker additions and failures
- **Work Queue** harnesses resources from multiple distributed systems
- **Work Queue** scales to hundreds to thousands of workers

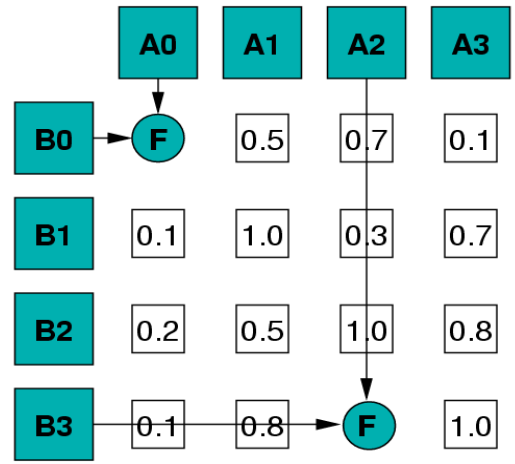
Work Queue: Success Stories

Makeflow



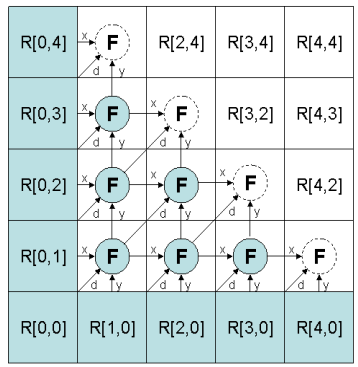
AllPairs(A[i], B[j], F(a,b))

AllPairs

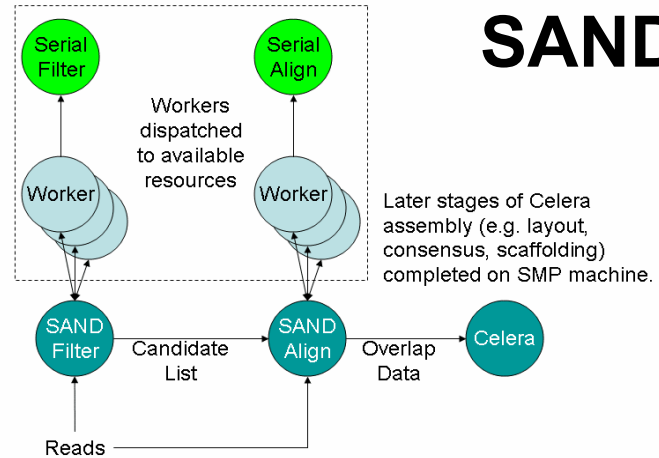


Wavefront

Wavefront (R[x,0], R[0,y], F(x,y,d))



SAND



Work Queue versus MPI

Work Queue

- Orchestrates ***ensemble*** of multiple external executables
- Number of workers ***dynamic***
- Scale up to ***large*** number of workers (10s, 100s, 1000s)
- Reliable and ***fault tolerant*** at the task level
- Allows for ***heterogeneous*** deployment environments
- Workers communicate only with ***Master***

MPI

- Coordinates multiple ***instances*** of single executable
- Number of workers ***static***
- Difficult to scale up to ***limited*** number of workers (16, 32, 64)
- Reliable at application level but ***no fault tolerance***
- Requires ***homogeneous*** deployment environment
- Workers can communicate with ***anyone***