How Many Eggs Can You Fit in One Nest?

Dynamically Shaping High Throughput Workflows

Douglas Thain, Ben Tovar, and Thanh Son Phung
University of Notre Dame
HTCondor Week, May 23 2022
HTCondor at Notre Dame

Dynamic Workflow Application

https://condor.cse.nd.edu
import work_queue as wq

queue = wq.WorkQueue(9123)

for p in range(1,100):
    task = wq.Task("./mysim -p {}").format(p)
    task.specify_memory(1024)
    queue.submit(task)

while not queue.empty():
    task = queue.wait(5)
    if task:
        printf("task {} completed".format(task.id))
Dynamic Workflows with Work Queue

Work Queue is a framework for building large distributed applications that span thousands of machines drawn from clusters, clouds, and grids. Work Queue applications are written in Python, Perl, or C using a simple API that allows users to define tasks, submit them to the queue, and wait for completion. Tasks are executed by a general worker process that can run on any available machine. Each worker calls home to the manager process, arranges for data transfer, and executes the tasks. A wide variety of scheduling and resource management features are provided to enable the efficient use of large fleets of multicore servers. The system handles a wide variety of failures, allowing for dynamically scalable and robust applications.

Who Uses Work Queue?

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

Learn About Work Queue

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

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

```python
import work_queue as wq

queue = wq.WorkQueue(9123)
for p in range(1,100):
    task = wq.PythonTask(myfunc,p)
    task.specify_memory(1024)

    queue.submit(task)

while not queue.empty():
    task = queue.wait(5)
    if task:
        printf("task {} completed".format(task.id))
```
Work Queue Architecture

- Python Appl
- Work Queue API
  - Local Files and Programs
  - Tasks
  - Results

Thousands of Workers in a Personal Cloud
- DOE
- SLURM
- Cluster
- Campus
- HTCondor
- Pool
- Public Cloud Provider
- Private Cluster

- Work Queue API
- Python Appl
- Tasks
- Results

- Local Files and Programs

- Thousands of Workers in a Personal Cloud

- DOE
- SLURM
- Cluster
- Campus
- HTCondor
- Pool
- Public Cloud Provider
- Private Cluster

- Work Queue API
- Python Appl
- Tasks
- Results

- Local Files and Programs
Some Work Queue Applications

Nanoreactors
ab-initio Chemistry

ForceBalance
FF Optimization

Lobster
CMS Data Analysis

Adaptive Weighted Ensemble
Molecular Dynamics

Low-Level API:
```
task = create(details);
submit( task );
task = wait( timeout );
```

Peter Bui, Dinesh Rajan, Badi Abdul-Wahid, Jesus Izaguirre, Douglas Thain,
Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications,
Workshop on Python for High Performance and Scientific Computing (PyHPC) at Supercomputing 2011.
A simple question:

What resources does each task need? (memory, cores, gpu, . . . )
Packing Tasks Into Manycore Nodes

Dynamic Workflow Application

Work Queue Manager

Allocate 2GB per Task A?

Work Queue Worker

12 cores and 12 GB RAM
Packing Tasks Into Manycore Nodes

Dynamic Workflow Application

Work Queue Manager

Allocate 4GB per Task A?

Work Queue Worker

\( T_A \quad T_A \quad T_A \)

12 cores and 12 GB RAM
Packing Tasks Into Manycore Nodes

Dynamic Workflow Application

Work Queue Manager

Work Queue Worker

Mix Task A and Task B?

T_A

T_B

T_A

T_B

T_B

T_B

12 cores and 12 GB RAM
Packing Tasks Into Manycore Nodes

Dynamic Workflow Application

Work Queue Manager

What if Function A Varies?

Work Queue Worker

12 cores and 12 GB RAM
We would like the user to tell us:
● How much memory does each task need?
● How many cores does each task need?
● Does this application use any GPUs?

But you might as well ask the user:
● How many roads must a man walk down?
Example: Memory Usage in Colmena-XTB

Dynamic K-means Bucketing
Example: Memory Consumption in TopEFT
Synthetic Distributions

- Exponential
- Normal
- Uniform
- Bimodal
- Trimodal
## Results

Metric - Average Task Efficiency (ATE): \( ATE(S) = \frac{1}{n} \sum_{i=1}^{n} \frac{c_i}{a_i} \) →tracks the ratio of resources a task uses to resources a task is given, on average.

<table>
<thead>
<tr>
<th></th>
<th>Colmena</th>
<th>TopEFT</th>
<th>Exponential</th>
<th>Uniform</th>
<th>Normal</th>
<th>Bimodal</th>
<th>Trimodal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Whole Machine</strong></td>
<td>15.8</td>
<td>0.6</td>
<td>12.4</td>
<td>39.1</td>
<td>15.7</td>
<td>31.3</td>
<td>30.7</td>
</tr>
<tr>
<td><strong>5% above Max</strong></td>
<td>33.2</td>
<td>69.1</td>
<td>48.4</td>
<td>59.6</td>
<td>15.7</td>
<td>56.7</td>
<td>43.7</td>
</tr>
<tr>
<td><strong>K-means Bucketing</strong></td>
<td><strong>43.9</strong></td>
<td><strong>91.0</strong></td>
<td><strong>56.3</strong></td>
<td><strong>62.4</strong></td>
<td><strong>16.1</strong></td>
<td><strong>93.9</strong></td>
<td><strong>71.3</strong></td>
</tr>
</tbody>
</table>
TopEFT CMS Data Analysis Application

Workflow's Shape

input files

one event

one chunk

processing function

processing function

processing function

processing function

accumulating function

accumulating function

final histogram
Dynamically update the size of chunks

- For a given target resource allocation, modify the size of future chunks according to historical data (e.g., mem vs #events).
- Tasks that exhaust resources are split in two or more.

If tasks **too small** for given allocation, new tasks are created with a **larger chunksize**.

If tasks **too big** for given allocation, split, new tasks are created with a **smaller chunksize**.
Dynamically Updating Chunksize to Memory Targets

- **Target memory:** 2GB
  - **Initial chunksize:** 1K
  - **User parameters:** set A

- **Target memory:** 1GB
  - **Initial chunksize:** 256K
  - **User parameters:** set A
Applications can have surprisingly complex resource distributions that are unknown to end users.

Let the workflow layer aggregate observations and then predict appropriate resource allocations.

Fit the nest to the eggs:
- Dynamically change task allocations to fit workers.

Fit the eggs to the next:
- Dynamically change task definitions to resource targets.

Continuing challenge: presenting end users with sufficient information to be constructive, but not overwhelming.
How much info to present?
User Interface to Resources?

Host: earth.crc.nd.edu:9123

Project: kmohrmann-workqueue-coffeaa

Owner: kmohrmann

Version: 7.4.5 FINAL

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

Quick Start:
conda install -c conda-forge ndcctools

dthain@nd.edu  btovar@nd.edu  tphung@nd.edu

This work was supported by NSF Award OAC-1931348
Run Time Dependency Management

Manager Environment

Python App  \(F_A\)  \(F_B\)

Parsl DFK

Work Queue Manager

Conda Pkgs  Pip Pkgs  Local Src

work_queue_worker

\(F_A\)  \(F_B\)

**How do we ensure that all the tasks get a consistent, minimal environment matching the manager?**
How to measure a single function call?

LFM - Lightweight Function Monitor

Activate LFMs with an import and the `@monitored` keyword

```python
from resource_monitor import monitored
from time import sleep

# declare a function to be monitored with the `@monitored()` decorator

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

(result, resources) = my_function_1(0.1)
print(result, '{0}'.format({'memory': resources['memory'], 'wall_time': resources['wall_time']}))

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