TaskVine: Workflow for Data Intensive and Serverless Applications

Douglas Thain and the CCL Team
University of Notre Dame
Throughput Computing 2023
Madison, WI July 2023
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.
How do I organize my work to use HTCondor?

- **Makeflow**
  - Unix-Oriented DAGs

- **Work Queue**
  - Dynamic Task Creation

- **TaskVine**
  - Dynamic Data Sharing

[https://condor.cse.nd.edu](https://condor.cse.nd.edu)
TaskVine is a system for executing **data intensive** scientific workflows on clusters, clouds, and grids from very small to massive scale.

TaskVine controls the computation **and storage** capability of a large number of workers, striving to carefully manage, transfer, and re-use data and software wherever possible.
The TaskVine manager directs workers to read data from remote sources, run tasks on that data, and share data with each other.

**TaskVine leaves data on workers in the cluster wherever possible!**
Design Goals for TaskVine

- **Make it easy** to construct *dynamic* workflows with millions of tasks running on thousands of cluster nodes.
- **Handle common failures** by detecting and recovering from worker crashes, network failures, and other unexpected events.
- **Avoid moving data** wherever possible: leave data in place until it needs to be moved or duplicated.
- **Re-use data objects** within and across workflows by tracking provenance from original sources all the way to final outputs.
- **Manage task resources** (cpu, gpu, mem, disk) carefully in order to pack in as much as we can (but not too much!) into each worker.
- **Support complex software** environments built from package managers by explicitly naming dependencies of tasks.
File = Single file or complex dir.

Manager directs all file movements and accesses.

Files are immutable and given a unique cache name.

Each task runs in a sandbox with a private namespace and an allocation of cores, memory, disk, and gpus.
Suppose you have a workflow like this: a dataset D comes from a web repository, a software package S comes from the shared filesystem. After passing through tasks 1-5, the final output F should be written to the filesystem. TaskVine aims to keep all of the data within the cluster, as follows.
The manager selects a worker for task 1, and then directs dataset D to be downloaded from the web, and software package S to be loaded from the shared filesystem.

In-Cluster Data Management
Next, task 1 is dispatched to that worker, where it reads dataset D, runs software package S, and produces file I, which stays where it is created.
Once file I is created, task 2 can run immediately on that node, producing file X. Software package S and file I are duplicated to the other worker nodes.
Now tasks 3 and 4 can run on the other worker nodes, producing files Y and Z.
Next, task 5 is dispatched to the middle worker. It consumes files X, Y, and Z, which are pulled in from peer nodes. The output file X is produced on that node.
Finally, output file F is written back to the shared filesystem, as the ultimate output of the workflow.

The manager directs the workers to delete any remaining uncacheable files.

Common input files remain to accelerate future workflows.
API: Declare Files Explicitly

```python
import ndcctools.taskvine as vine

m = vine.Manager(9123)

file = m.declareFile("mydata.txt")
buffer = m.declareBuffer("Some literal data")
url = m.declareURL("https://somewhere.edu/data.tar.gz")
temp = m.declareTemp();

data = m.declareUntar(url)
package = m.declareStarch(executable)
```
API: Connect Tasks to Files

task = vine.Task("mysim.exe -p 50 input.data -o output.data")

t.add_input(url,"input.data")
t.add_output(temp,"output.data")

t.set_cores(4)
t.set_memory(2048)
t.set_disk(100)
t.set_tag("simulator")

taskid = m.submit(t)
task = vine.PythonTask(simulate_func, molecule, parameters)

t.set_cores(4)
t.set_memory(2048)
t.set_disk(100)
t.set_tag("simulator")

taskid = m.submit(t)

... 

print(t.result)
Sample Application: NCBI Blast


query_string = "GCTAATCCA..."

software = m.declareUntar(m.declareURL(blast_url))
landmark = m.declareUntar(m.declareURL(landmark_url))

task = vine.Task("blastp -db landmark -query query.file")
task.add_input(software, "blastdir")
task.add_input(database, "landmark")
task.add_input_buffer(query_string, "query.file")
task.set_env_var("BLASTDB", value="landmark")

m.submit(task)


query_string = "GCTAATCCA..."

software = m.declareUntar(m.declareURL(blast_url))
landmark = m.declareUntar(m.declareURL(landmark_url))

task = vine.Task("blastp -db landmark -query query.file")
task.add_input(software,"blastdir")
task.add_input(database,"landmark")
task.add_input_buffer(query_string, "query.file")
task.set_env_var("BLASTDB", value="landmark")

Upshot: Common data prep done once for many tasks on a node.
New capabilities are added to the system by defining mini-tasks that use the same task infrastructure to define dependencies and execute them reproducibly:

```python
data = m.declareXRootD( "xrootd://host/path", "proxy" )
```

Which is defined as a mini-task like this:

```python
t = vine.Task("xrdcp {} output.root".format(url));
t.add_input(proxy,"proxy509.pem")
t.set_env_var("X509_USER_PROXY","proxy509.pem")
data = m.declareMiniTask(t,"output.root")
```
Manager Schedules Transfers

Uncontrolled Transfers

Controlled Transfers

500 task BLAST workflow (from above) requires both software and data from NCBI.

Obtaining and deploying assets is part of the workflow itself!
Files have one of three lifetimes:

- single-task
- workflow (default)
- forever

"forever" cached objects are given content addressable names from a Merkle Tree of the file's provenance. If any inputs change, then so does the name of the output, and it's not the same file.
Eliminating Startup Costs

The next time a (similar) workflow is run, software and data assets are already copied, unpacked, verified, and ready to use on the cluster nodes.
# Define ordinary Python functions

def my_sum(x, y):
    return x+y

def my_mul(x, y):
    return x*y

# Create a library object from functions
L = m.create_library_from_functions("my_library", my_sum, my_mul)

# Install the library on all workers.
m.install_library(L)
# Define a function invocation and submit it

```python
for i in range(1, 100):
    t = vine.FunctionCall("my_library", "my_sum", 10, i)
```

Simply converting "import tensorflow" into the preamble of a Library task saves **1.2GB** of Python libraries, **30K** metadata system calls, and **5-10s** latency per `FunctionCall`.

David Simonetti
Multi-Modal Workflows

100x Standard Tasks
Build model from MNIST data.

For each produced model:
Deploy LibraryTask for inference.

Submit 10x FunctionCalls that
invoke each LibraryTask.

Application gradually accelerates
as standard tasks produce data
that define libraries that can then
be invoked.
Late stage data analysis for LHC CMS experiment. Search for new physics impacting associated top quark production using the framework of effective field theory (EFT). TopEFT uses Coffea HEP framework and scientific python components.
Old:
Accumulation Data Returned
TopEFT + Work Queue

Stuck on Long Tail!

New:
In-Cluster Accumulation
TopEFT + TaskVine

Tail Eliminated!
# Estimate three values for pi
a, b, c = pi\(10^{\ast\ast}6\), pi\(10^{\ast\ast}6\), pi\(10^{\ast\ast}6\)

# Compute the mean of the three estimates
mean_pi = mean(a, b, c)

# Print the results
print("a: {:.5f} b: {:.5f} c: {:.5f}".format(a.result(), b.result(), c.result()))
print("Average: {:.5f}".format(mean_pi.result())))
import dask
import dask.array as da

# Dask Task Graph

d = {'x': 1,
     'y': (inc, 'x'),
     'z': (add, 'y', 10)}

x = da.random.random((10000,10000),chunks=5000)
y = x + x.T
z = y[::2,500:].mean(axis=1)

result = z.compute()
print(result)
import ndcctools.taskvine as vine
import dask
import dask.array as da

# Create a new manager listening on port 9123
manager = vine.DaskVine(9123)

x = da.random.random((10000,10000),chunks=5000)
y = x + x.T
z = y[:2,500:].mean(axis=1)

result = z.compute(manager.get())
print(result)
Research Challenges

- Decomposing DAGs of Short Tasks
  - Dask and Parsl can produce $O(1M)$ function evals that may be less than one second each.
- Automatically Identifying Serverless Candidates
  - Can we recover the cost of deployment?
- Dynamic Resource Management
  - How to choose resources for raw functions?
- Dependency Management Challenges
  - Do you know what your code depends upon?
  - Do you want to use what others depend upon?
  - What are your expectations regarding updates?
Current Status of TaskVine

- **TaskVine** is a component of the Cooperative Computing Tools (cctools) from Notre Dame alongside Makeflow, Work Queue, Resource Monitor, etc.
- Research software with an engineering process: issues, tests, manual, examples.
- We are eager to collaborate with new users on applications and challenges!

This work was supported by NSF Award OAC-1931348

conda install -c conda-forge ndcctools

https://cctools.readthedocs.io
For more information…

https://ccl.cse.nd.edu/software/taskvine
https://dthain.github.io

This work was supported by NSF Award OAC-1931348

TaskVine is a framework for building large scale data intensive dynamic workflows that run on HPC clusters, GPU clusters, and commercial clouds. As tasks access external data sources and produce their own outputs, more and more data is pulled into local storage on workers. This data is used to accelerate future tasks and avoid re-computing existing results. Data gradually grows "like a vine" through the cluster. TaskVine is our third-generation workflow system, built on our twenty years of experience creating scalable applications in fields such as high energy physics, bioinformatics, molecular dynamics, and machine learning.
Simply converting "import tensorflow" into the preamble of a Library task saves 1.2GB of Python libraries, 30K metadata system calls, and 5-10s latency per FunctionCall. We can mix standard Tasks, Libraries, and FunctionCalls in the same workflow: