funcX: A Federated Function Serving Fabric for Science

Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ben Galewsky, Daniel S. Katz, Ian Foster, Kyle Chard
Serverless computing

Provider runs infrastructure and manages allocation of resources

Function as a Service (FaaS)

- Pick a runtime (Python/JS/R etc.)
- Write function code
- Run (and scale)

Low latency, on-demand, elastic scaling

Combine functions (e.g., workflows) to solve complex problems
Function as a service in science?

1. Support new workloads by decomposing applications into functions
   • Real-time, interactive, stream processing
   • Simplify development, maintenance, testing
2. Facilitate use of diverse compute resources
   • Abstract compute infrastructure
3. Enable fluid function execution across the heterogeneous computing continuum
   • Containers enable portability and sandboxing

➔ funcX: high performance and federated function as a service
Scientific workloads are becoming more granular

(a) tabular file extraction
(b) MNIST digit prediction
(c) DIALS stills process
(d) tomographic preview
(e) correlation spectroscopy
Using existing computing infrastructure has significant barriers

- Complex queuing systems with unpredictable delays
- Coarse allocation blocks
  - Not designed for short-duration tasks with minimal resource needs
- High learning curve and lack of portability
  - Translation to different schedulers (and update when they inevitably break)
- Heterogeneous architectures
- Different modules and source code
- Different container technology

There is an impedance mismatch between short duration function workloads and existing infrastructure available to scientific users
Specialization demands distribution

- As we face the end of Moore’s law we are seeing increasing specialization
  - Establishes a *continuum* of computing capacity where flexible devices can run many types of tasks poorly and specialized devices can few tasks very well
- Increasing specialization leads to distribution => remote and portable computing
Computation should be fluid: Trigger analysis in high energy physics

Top quark jet tagging and neutrino event classification (based on ResNet)

FPGA: 30 msec

CPU: 2 sec

Local: 2000 msec

30 + 10 + 10 = 50 msec

40x acceleration

Remote execution is not new …

• We have long strived to compute wherever it makes the most sense:
  • Resource availability, data location, analysis time, wait time, software licenses, etc.

• Remote computing has always been complex and expensive, however we now have:
  • High speed networks
  • Universal trust fabrics
  • Containers
**FuncX: a function serving ecosystem for science**

**Functions:**
- Register once, run anywhere, any time

**Endpoints:**
- Dynamically provision resources, deploy containers, and execute functions
- Exploit local architecture/accelerators

**funcX Service:**
- Register and share endpoints
- Register, share, run functions

**Turn any machine into a function serving endpoint**

**Route functions to remote endpoints**
- Closest, cheapest, fastest, accelerators ...
Transform clouds, clusters, and supercomputers into high-performance function serving systems
Register functions for execution on any funcX endpoint.

Defining a function:

```python
def funcx_sum(items):
    return sum(items)
```

EP(x) registry
Register functions for execution on any funcX endpoint

Registration

\[ f(x) + \text{dependencies} \]

```python
def funcx_sum(items):
    return sum(items)
```

Register

\[ f(x) \quad g(x) \quad h(x) \quad k(x) \]

repo2docker

EP(x) registry
Reliably and scalably execute registered functions on any funcX endpoint

Execution

\[
\begin{align*}
&f(x) [1, 2, 3, ..] \\
g(x) ['a', 'b', 'c', ...]
\end{align*}
\]
Deploying a funcX endpoint

- Pip install funcX (e.g., using Conda)
- Authenticate and register with the funcX service
- Configure the endpoint for the local resources (using Parsl)
Coding the Computing Continuum with funcX

1. Define Python functions and register them with funcX
   - Codes are serialized and stored on the cloud
   - Registration returns a UUID for the function which is used for invocation

2. Run the function on a specified endpoint
   - args* and kwargs* are serialized and sent to funcX
   - Function code and inputs routed to endpoint

3. Retrieve Results
   - Inspect status, wait on results, retrieve outputs

```
from funcx.sdk.client import FuncXClient
fcx = FuncXClient()

def funcx_sum(items):
    return sum(items)

# Register a function
sum_func = fcx.register_function(funcx_sum)
tutorial_ep = '4b116d3c-1703-4f8f-9f6f-39921e5864df'
input_items = [1,2,3,4,5]

# Execute the function on the tutorial endpoint
res = fcx.run(input_items, endpoint_id=tutorial_ep, function_id=sum_func)

# Retrieve results
fcx.get_result(res)
```
Demo

Setup an endpoint
$ conda create –n funcx python=3.6
$ pip install funcx
$ funcx-endpoint configure <ENDPOINT_NAME>
$ funcx-endpoint start <ENDPOINT_NAME>

Run a function
from funcx.sdk.client import FuncXClient
fxc = FuncXClient()
def funcx_sum(items):
    return sum(items)
func_uuid = fxc.register_function(funcx_sum)
res = fxc.run(items, endpoint_id=<UUID>, function_id=func_uuid)
fxc.get_result(res)
funcX service: fire-and-forget managed function execution

REST Web interface
• Register and manage endpoints
• Publish and invoke Python functions
• Globus Auth for authn/z

Redis store
• Store and share functions
• Track and allocate tasks
• Reliable endpoint task queues

Endpoint forwarders
• Forward serialized functions and inputs for execution
funcX endpoint: high performance function execution on arbitrary computers

Secure communication

- Securely connect **out** to forwarder for registration
- ZeroMQ for low latency comm.
- Retrieve and queue tasks

Compute abstraction

- Acquire nodes from diverse compute resources (using **Parsl**)  
- Deploy workers inside containers to nodes

Endpoint

- Report state, usage, and liveness
funcX scales to 100K+ workers

- funcX endpoints deployed on ALCF Theta and NERSC Cori
- Strong scaling (100K concurrent functions) shows good scaling up to 2K containers even with short sleep tasks
- Weak scaling (10 tasks per container) shows scaling to 131K concurrent containers (1.3M tasks)
Elastic execution irrespective of underlying system

• funcX agent deployed on a Kubernetes cluster
• Each function is registered in a container and allowed to use 0-10 pods (unit of execution)
• FuncX elastically scales active pods (bottom) based on workload (top)
funcX recovers from worker, manager, and endpoint failures
Optimizing performance: prefetching and Batching

![Graph showing completion time vs. prefetch count per node]

- no-op (Theta)
- 1ms sleep (Theta)
- 10ms sleep (Theta)
- 100ms sleep (Theta)

![Graph showing throughput vs. number of batches]

- 1 worker
- 2 workers
- 4 workers
- 8 workers
- 16 workers
- 32 workers

![Graph showing average latency per request vs. batch size]

- DIALS
- Metadata
- MNIST
- Neurocartography
- XPCS
Scheduling heterogenous tasks over heterogenous endpoints

- Experimenting with scheduling across heterogenous funcX endpoints
  - Raspberry Pis, Desktops, Cloud instances, GPUs

- Three scheduling algorithms
  - Round robin, Fastest endpoint, smallest ETA

- Three function types of three sizes
  - Matrix multiplication, map reduce, file I/O

- Smaller tasks distributed across slower endpoints
Example application: Manufacturing

1. Instrument sensors stream data to the MDML
2. Use FaaS to analyze data on-demand
3. FaaS tasks distributed across the computing continuum
4. Results are used to guide the experiment
Example application: Serial Crystallography
Example application: DLHub
Lessons learned applying funcX to science use cases

✓ Abstracts the complexity of using diverse compute resources
✓ Simplicity: automatic scaling, single interface
✓ Flexible web-based authentication model
✓ Enables event-based processing and automated pipelines
✓ Increases portability between sites, systems, etc.
✓ Resources can be used efficiently and opportunistically
✓ Enables secure function sharing with collaborators

✗ FaaS is not suitable for some applications
✗ Ratio of data size to compute has to be reasonable
✗ Containerization does not always provide entirely portable codes
✗ Coarse allocation models do not map well to fine grain/short functions
✗ Decomposing applications isn’t always easy (or possible)
**Apps** define opportunities for parallelism

- Python apps call Python functions
- Bash apps call external applications

Apps return “futures”: a proxy for a result that might not yet be available

Apps run concurrently respecting data dependencies. Natural parallel programming!

Parsl scripts are independent of where they run. Write once run anywhere!

```
@python_app
def hello():
    return 'Hello World!'

print(hello().result())
```

Hello World!

```
@bash_app
def echo_hello(stdout='echo-hello.stdout'):
    return 'echo "Hello World!"

echo_hello().result()
```

```
with open('echo-hello.stdout', 'r') as f:
    print(f.read())
```

Hello World!

Try Parsl: https://mybinder.org/v2/gh/Parsl/parsl-tutorial/master
funcX creates a federated FaaS ecosystem for science

funcX is a federated FaaS system designed to meet the requirements of scientific computing

Enables fluid execution by dispatching functions to wherever makes the most sense

Initial deployments scale to 130K+ concurrent workers and >1.2M functions

http://github.com/funcx-faas