Parsl: Decorators and Function Parameters

Arha Gatram
Dr. Douglas N. Friedel

ILLINOIS
NCSA | National Center for Supercomputing Applications

What happens?

- Look at this code
- Should these function calls have the same or different behavior?

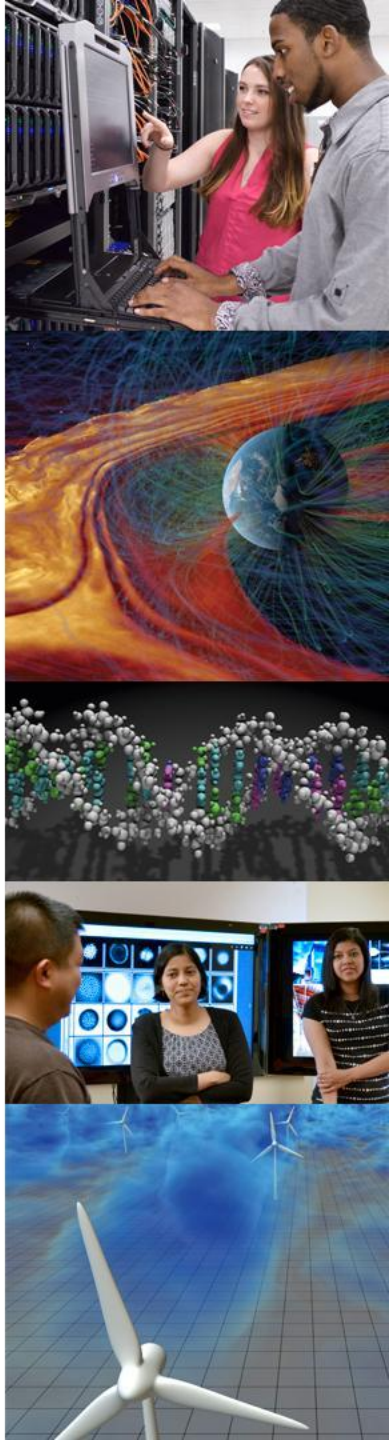
```
@python_app
def wait(walltime=1):
  import time
  time.sleep(2)
  return "success!"

future = wait(walltime=3)
print(future.result())

future = wait(3)
print(future.result())
```

Strange Behavior

- The two function calls should have the same result
- In the second call, the argument should automatically bind to the keyword argument
- Instead, we get an error: TypeError: wait() got multiple values for argument ‘walltime’
- So what gives?
- Let’s dive into how Parsl works under the hood



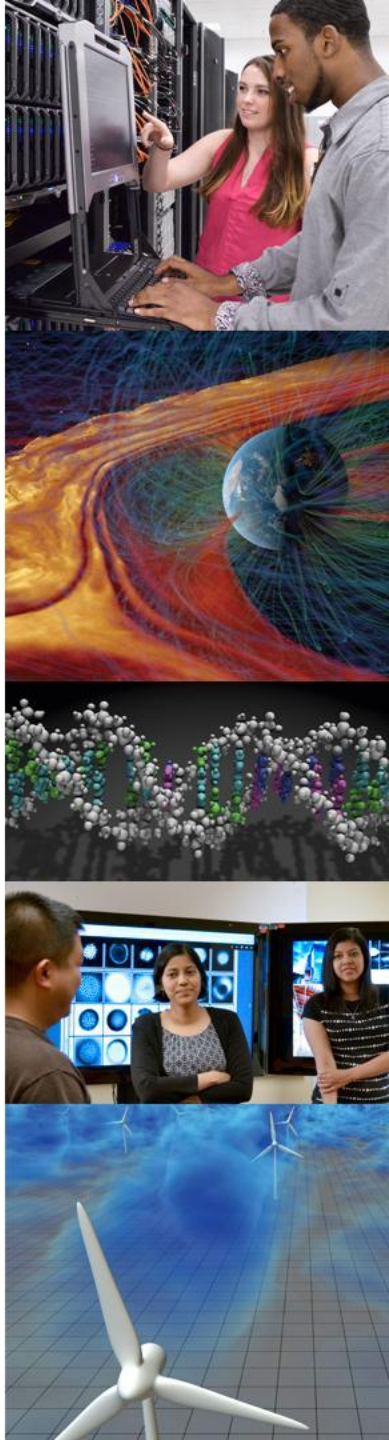
How does Parsl parallelize a function?

- Uses the Python construct of decorators, but not quite
- Parsl actually gives you an object, but still defines a decorator function
- Gives you the syntax sugar, but allows for some additional internal processing
- How does it still work like a function?

```
def python_app(function: Optional[Callable] = None,
    def decorator(func: Callable) -> Callable:
        def wrapper(f: Callable) -> PythonApp:
            return PythonApp(f,
                             data_flow_kernel=data_flow_kernel,
                             cache=cache,
                             executors=executors,
                             ignore_for_cache=ignore_for_cache,
                             join=False)
        return wrapper(func)
    if function is not None:
        return decorator(function)
    return decorator
```

Classes as Functions

- In C++ you might overload the operator()
- In Python you can define a special class method __call__ which does essentially the same thing
- But how do we match this to a function signature that we only know at runtime?
- Take in (*args, **kwargs)

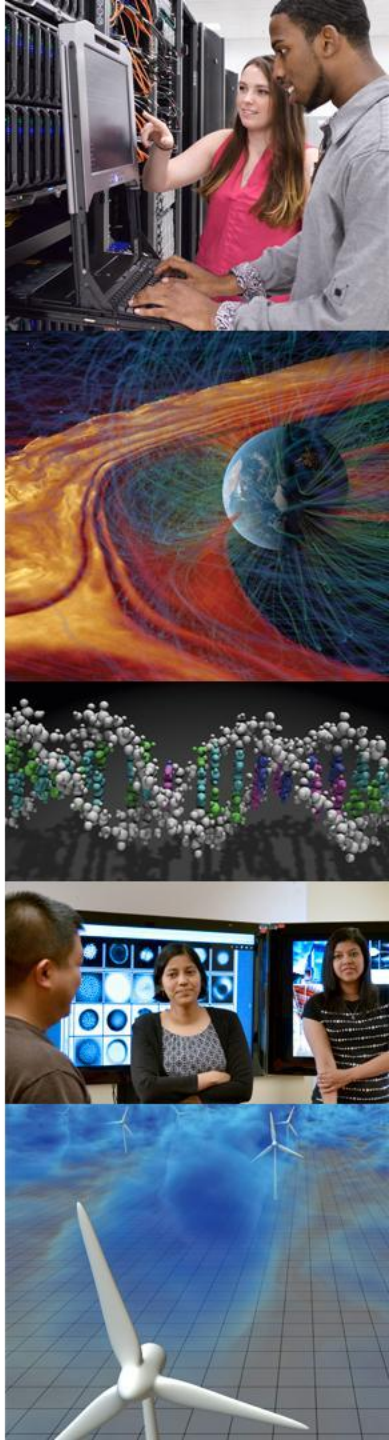


What's the deal with walltime?

- Parsl has certain "special parameters"
- There's some introspection that Parsl does to handle these differently than normal parameters
- This introspection is why Parsl returns a class
- The constructor looks through the parameters in the signature for these special parameters and registers them

Is that really necessary?

- If these special parameters have default values, they would only be reconciled when you call the original function
- The __call__ resolves into a call to the DataFlowKernel which passes the function and parameters along
- This will change behavior according to those special parameters, so you need this introspection

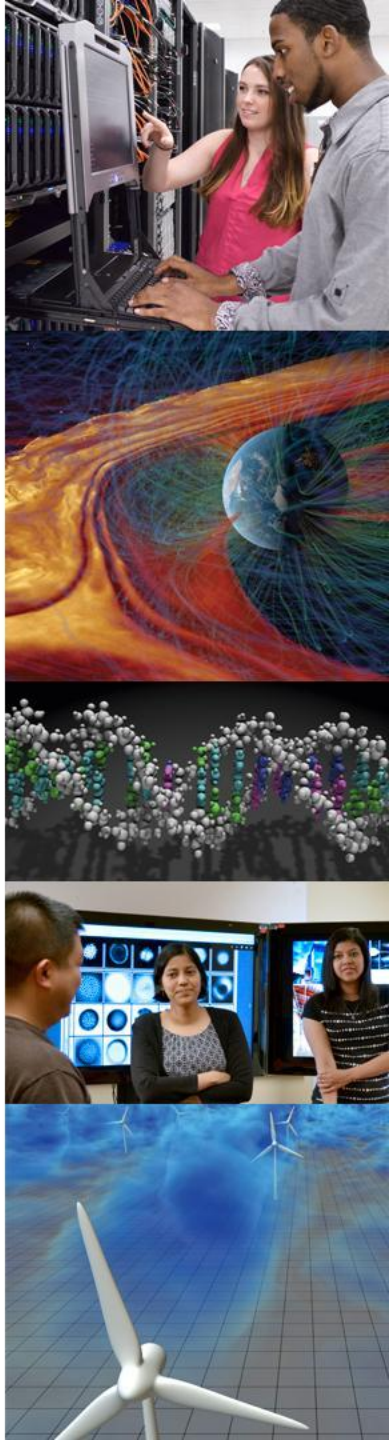


Where's the bug?

- The internal processing stores a dictionary of default arguments for special keywords
- This is updated for **kwargs passed in so that the introspection has access to it
- But if the parameter is not passed in as a keyword argument, it's passed in through *args
- When the function is finally called, the parameter is bound to whatever is passed in with *args but also has the default value added to **kwargs

Moral of the Story

- This is fixable if you do binding earlier in the process with inspect.signature.bind functionality
- This would still require significant refactoring of how Parsl handles these special keyword arguments
- Bind considers only **kwargs as actual keyword arguments so what we were doing earlier are considered default arguments which get resolved to regular arguments
- Extra work to find special parameters now, so still problematic



Thank you! Questions?