Better Parallelism in Python with YieldTasks

Marc Davis

Map: A Core Parallelism Primitive

```
results = []
for data in list_of_data:
    results.append(f(data))

results = [f(data) for data in list_of_data]

results = map(f, list_of_data)
```

- Run a function on many different inputs
- Frequently parallel code involves parallelizing a map operation
- Many other models of parallelism (e.g. MapReduce) can be written in terms of a parallel map operation
- Its NOT everything, but its a good place to start.

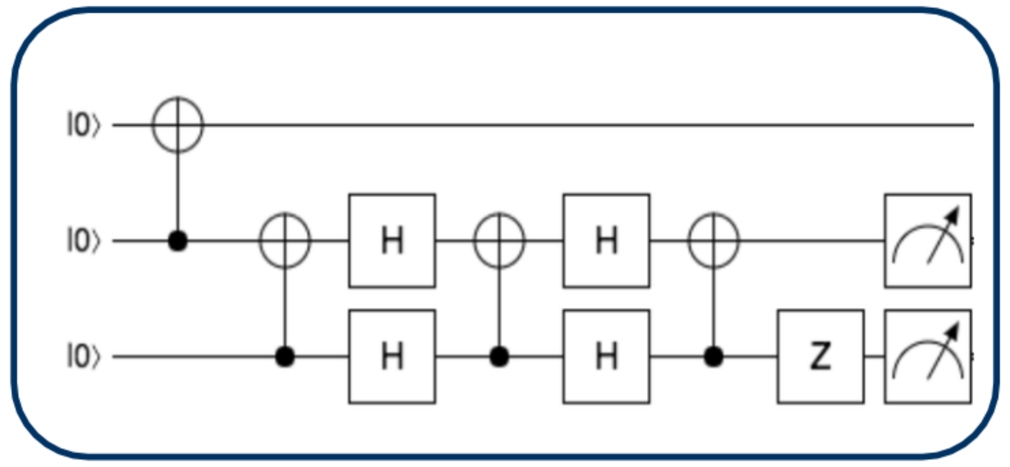
Case Study: Qsearch

What it does:

Unitary

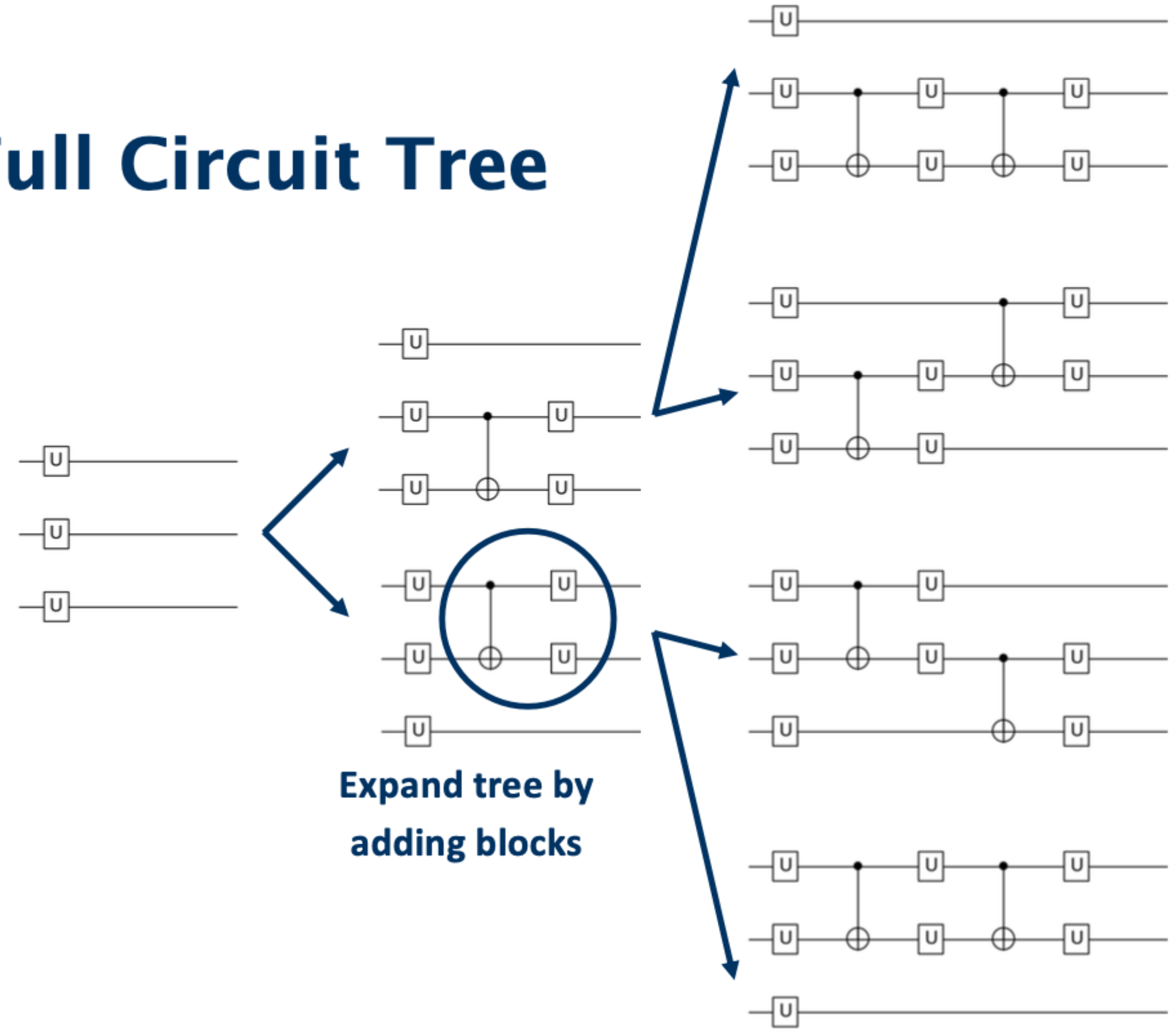
$$U_{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Circuit



How it works:

Full Circuit Tree

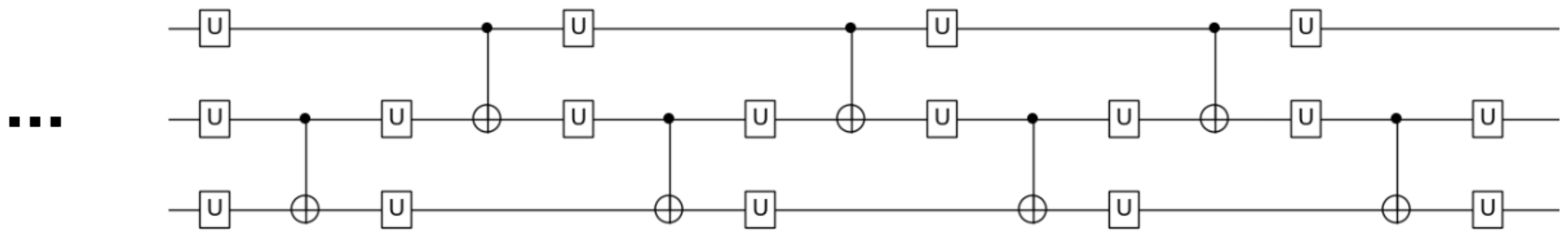


Parameterized Circuit

$$U(\vec{x_1} \dots \vec{x_n}) = \left(U_3(\vec{x_1}) \otimes \cdots \otimes U_3(\vec{x_q})\right) \prod_{i=q/2}^{(n-1)/2} I_1(i) \otimes \left(CNOT\left(U_3(\vec{x_{2i}}) \otimes U_3(\vec{x_{2i+1}})\right)\right) \otimes I_2(i)$$

q represents the number of qubits and n represents the total number of single-qubit gates

$I_1(i)$ and $I_2(i)$ are functions that represent the needed placements of identities to create the specified structure.



Leaf Node:

Any circuit structure that the optimizer can find parameters for such that the distance to the target is near zero.

- By continuing to add layers, we can represent any circuit, somewhere in our tree.
- The lowest-depth leaf node represents a minimal-CNOT length circuit solution

Case Study: Qsearch

Written in:

python™ — User facing code (Python is the industry standard for Quantum Computing right now)

Rust — Numerical optimization of quantum circuits

Opportunities for Parallelism:

Pseudocode

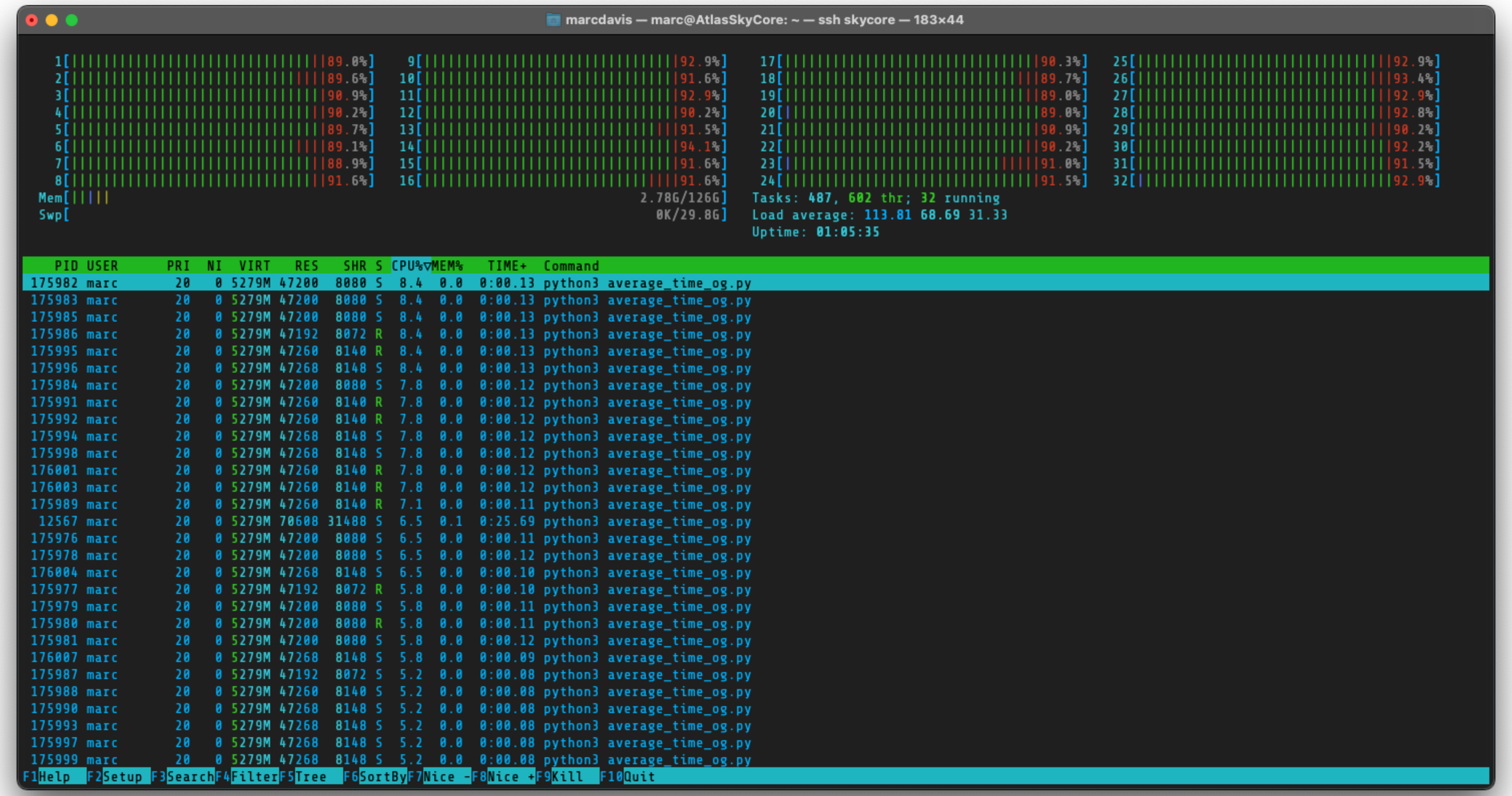
```
output = []
for unitary in unitaries_to_synthesize: # Could be parallelized
    output.append(synthesize(unitary))

def synthesize(unitary):
    best_ansatz = prepare()
    while not is_solution(best_ansatz):
        ansatz_list = get_list_of_next_ansatzs()
        result_list = []
        for new_ansatz in ansatz_list: # Could be parallelized
            result_list.append(evaluate_ansatz(new_ansatz))

        best_ansatz = choose_best_ansatz(result_list)

    return best_ansatz

def evaluate_ansatz(new_ansatz):
    data = []
    starting_points = generate_random_starting_points()
    for starting_point in starting_points: # Could be parallelized
        data.append(optimize(new_ansatz, starting_point))

    best_data = evaluate_data(data)
    return best_data
```

Even Simpler Pseudocode

```
for unitary in unitaries_to_synthesize:
    while not is_solution(best_ansatz): # This loop can't be parallelized
        for new_ansatz in get_list_of_next_ansatzs():
            for starting_point in starting_points:
                call_into_optimization_library()
```

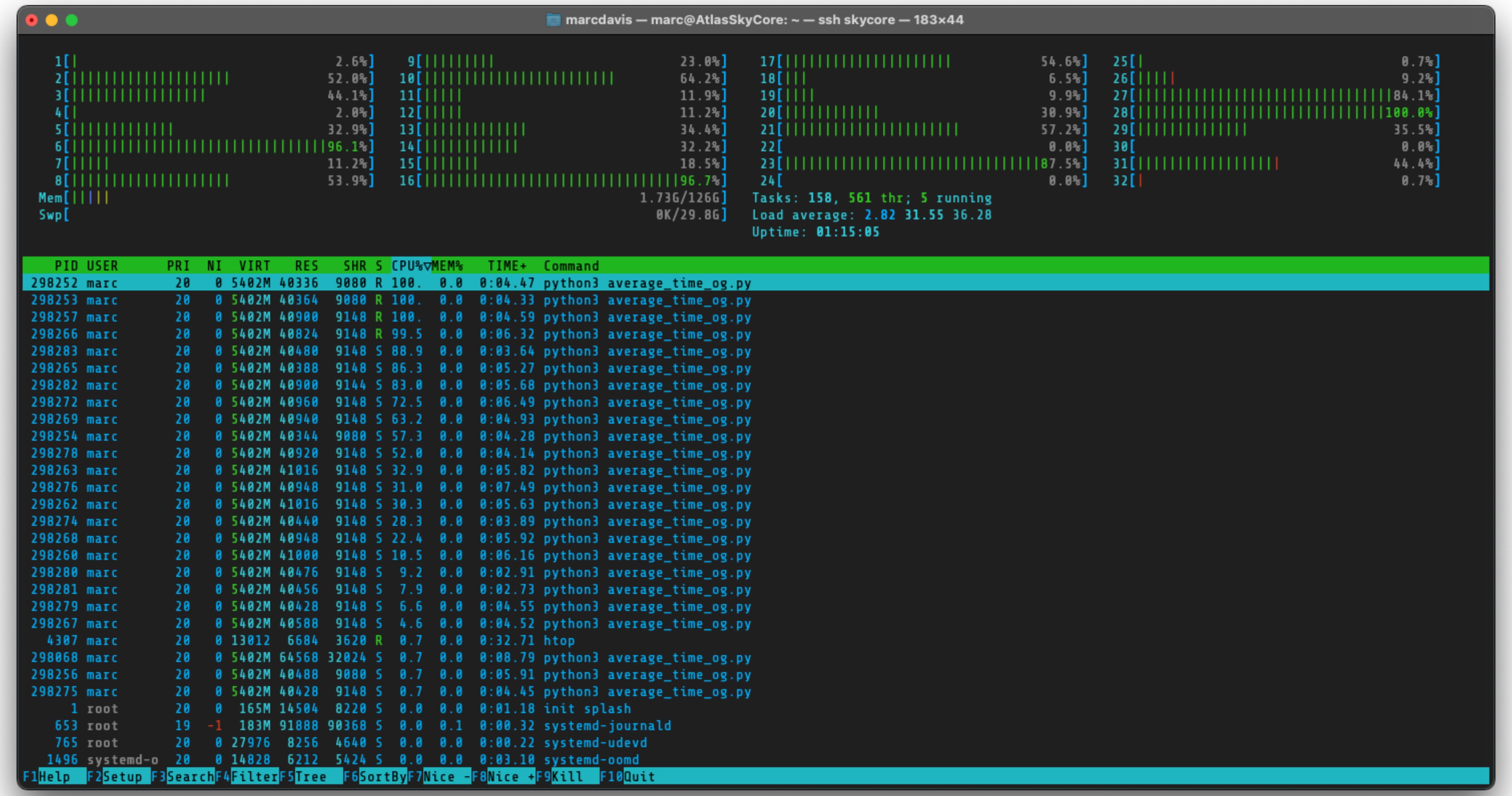
- Nested loops (one of which can't be parallel)
- Difficult to wrap into one loop
- Data transferred is "small" (KB)
- Smallest level of computation is "large" (0.01s-5s)

Oversubscription vs Undersubscription



Oversubscription

- Parallelize at every layer
- Rely on the OS scheduler to sort things out
- Performance lost due to context switching
- Sub-processes of sub-processes



Undersubscription

- Pick one layer to focus on
- Keep things simple
- Performance lost due to underutilized CPU
- One sub-process per hardware thread (usually)

Expression vs Implementation of Parallelism

```
results = parallel_map(f, data)
```

multiprocessing

```
results = pool.imap_unordered(f, data)
```

Pros:
- very simple
- lightweight

Cons:
- single-node
- no GPU

dask

```
results = data.applymap(f)
```

Pros:
- powerful

Cons:
- complex
- can have very poor performance if not configured well

TensorFlow

```
results = data.map(f)
```

Pros:
- similar API to other popular libraries

Cons:
- requires converting your data to TensorFlow formats

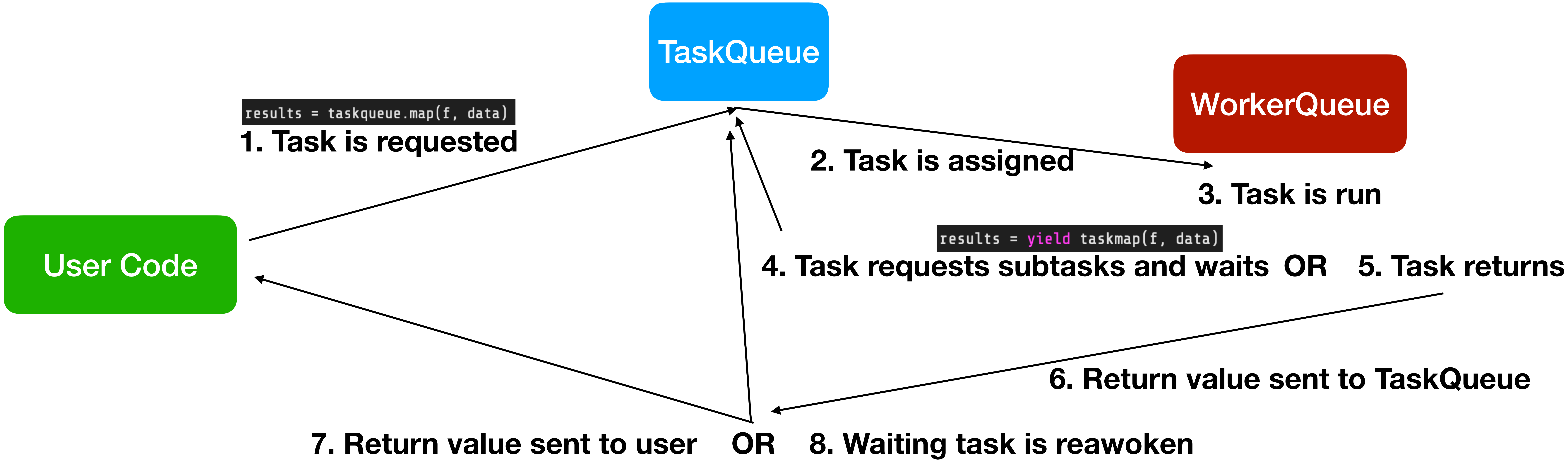
Release Qsearch:
- both over and under subscription (in different places)
- multiprocessing

BQSKit (Successor to Qsearch):
- undersubscription
- dask

Introducing Yieldtasks

```
results = taskqueue.map(f, data)
```

```
results = yield taskmap(f, data)
```



`yield` Makes a function return a **Generator**, which will store the intermediate state of that function while it waits for more data.
(This is a way of implementing **coroutines** in Python)

TaskQueue and WorkerQueue can be re-implemented in **different backends** without modifying **user code**.

```
taskqueue = DaskTaskQueue()
taskqueue = MultiprocessingTaskQueue()
```

Converting to Yieldtasks

```
output = []
for unitary in unitaries_to_synthesize: # Could be parallelized
    output.append(synthesize(unitary))

def synthesize(unitary):
    best_ansatz = prepare()
    while not is_solution(best_ansatz):
        ansatz_list = get_list_of_next_ansatzs()
        result_list = []
        for new_ansatz in ansatz_list: # Could be parallelized
            result_list.append(evaluate_ansatz(new_ansatz))

        best_ansatz = choose_best_ansatz(result_list)

    return best_ansatz

def evaluate_ansatz(new_ansatz):
    data = []
    starting_points = generate_random_starting_points()
    for starting_point in starting_points: # Could be parallelized
        data.append(optimize(new_ansatz, starting_point))

    best_data = evaluate_data(data)
    return best_data
```

```
taskqueue = MultiprocessingTaskQueue()
output = taskqueue.map(synthesize, unitaries_to_synthesize)

def synthesize(unitary):
    best_ansatz = prepare()
    while not is_solution(best_ansatz):
        ansatz_list = get_list_of_next_ansatzs()
        result_list = yield taskmap(evaluate_ansatz, ansatz_list)
        best_ansatz = choose_best_ansatz(result_list)

    return best_ansatz

def evaluate_ansatz(new_ansatz):
    starting_points = generate_random_starting_points()
    data = yield taskmap(optimize, starting_point)
    best_data = evaluate_data(data)
    return best_data
```

- Parallelized at every level
- Neither over nor under subscribes
- Switching backend can be done by changing a single line

Performance Comparison

Processor: AMD 5950X (32 threads)
Task: Perform 8 unitary synthesis tasks.
Comparison: Original Qsearch from PyPy **vs** Qsearch implemented with Yieldtasks
(Then repeat 10x to average out randomness)

	Qsearch 16 Multistarts Benchmark List x10
Undersubscribe	19m 58s
Oversubscribe	8m 21s
Yieldtasks	5m 32s

Summary

Yieldtasks Can:

- Enable nested parallel map operations
- Without oversubscribing or undersubscribing
- Potential to switch backends with 1 line of code

Implemented So Far:

- Working Yieldtasks implementation with a multiprocessing backend
- Qsearch built on Yieldtasks outperforms original implementation

Next Steps and Future Goals:

- Polish and release Yieldtasks implementation
- Improve error handling
- Implement other backends
  - Multiple computers on a network?
  - Multiple nodes in a cluster?
  - Dask?
  - GPU? (Since GPUs don't run Python, special handling will be needed…)
- Move to a syntax based on async/await