

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)
```

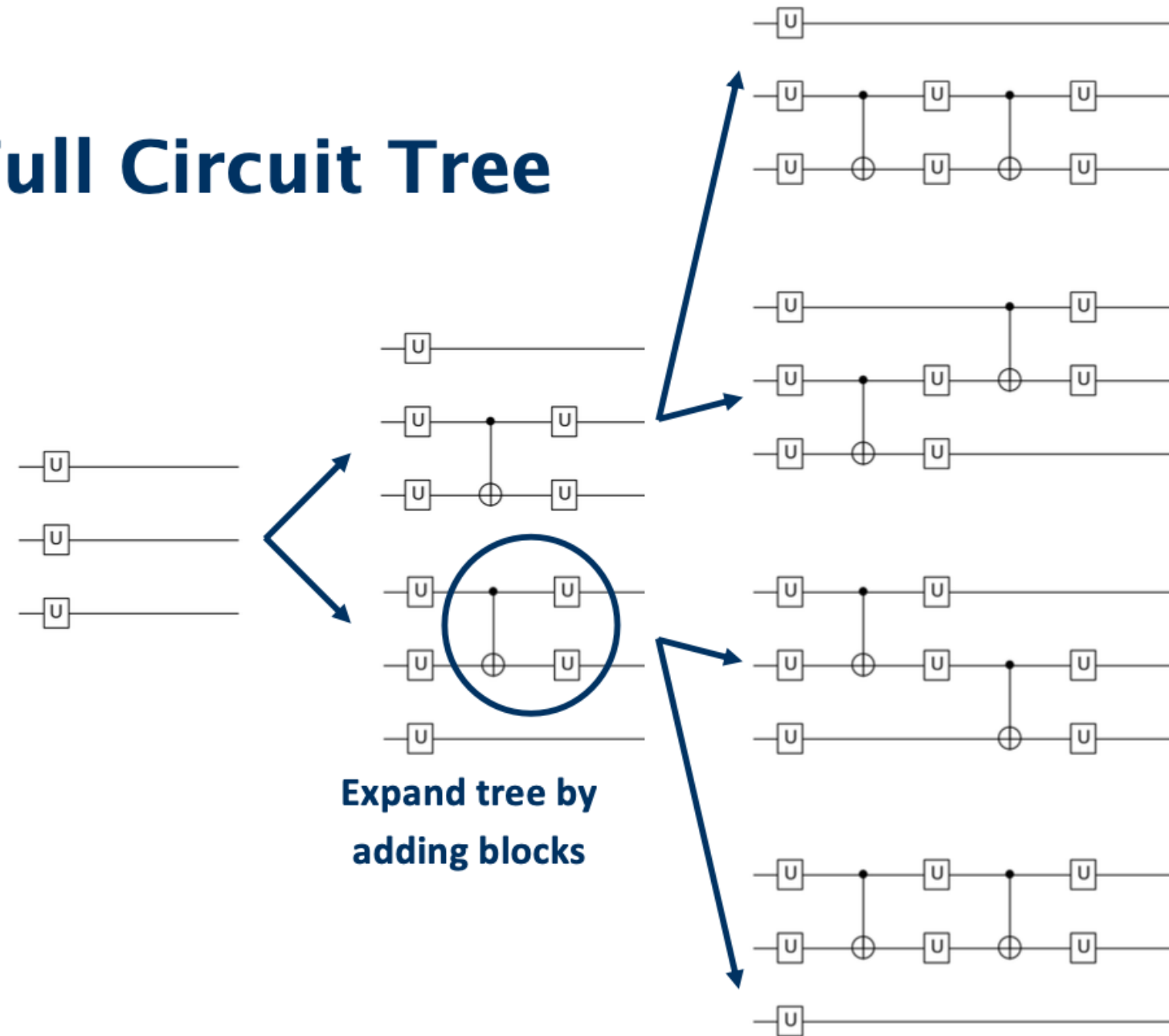
- Run a function on many different inputs
- Frequently parallel code involves parallelizing a map operation
- Many other models of parallelism (e.g. MapReduce) can be written in terms of a parallel map operation
- Its NOT everything, but its a good place to start.

Case Study: Qsearch

What it does:



Full Circuit Tree

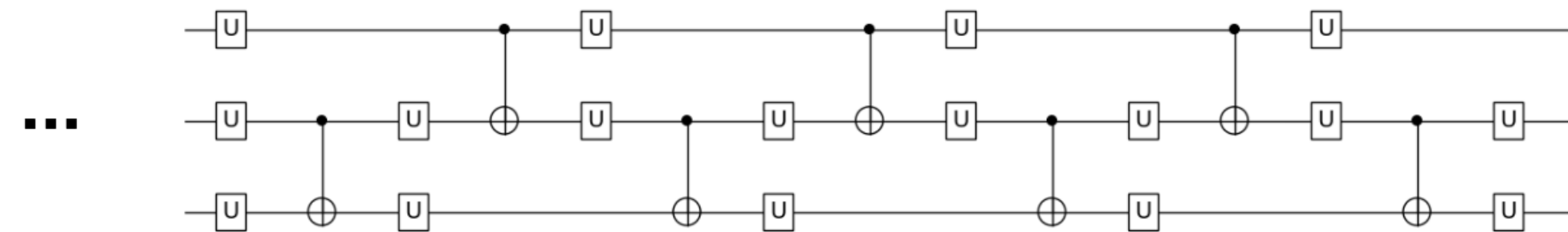


How it works:

Parameterized Circuit

$$U(\vec{x}_1 \dots \vec{x}_n) = (U_3(\vec{x}_1) \otimes \dots \otimes U_3(\vec{x}_q)) \prod_{i=q/2}^{(n-1)/2} I_1(i) \otimes (\text{CNOT}(U_3(\vec{x}_{2i}) \otimes U_3(\vec{x}_{2i+1}))) \otimes I_2(i)$$

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 leafnode 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)



Numerical optimization of quantum circuits

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
```

Opportunities for Parallelism:

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

```
marcdavis -- marc@AtlasSkyCore: ~ -- ssh skycore -- 183x44
1[|||||] 9[|||||] 17[|||||] 25[|||||]
2[|||||] 10[|||||] 18[|||||] 26[|||||]
3[|||||] 11[|||||] 19[|||||] 27[|||||]
4[|||||] 12[|||||] 20[|||||] 28[|||||]
5[|||||] 13[|||||] 21[|||||] 29[|||||]
6[|||||] 14[|||||] 22[|||||] 30[|||||]
7[|||||] 15[|||||] 23[|||||] 31[|||||]
8[|||||] 16[|||||] 24[|||||] 32[|||||]
Mem[|||||] 2.78G/12.6G Tasks: 487, 602 thr; 32 running
Swp[|||||] 0K/29.0G Load average: 113.01 68.69 31.33
Uptime: 01:05:35

PID USER PRI NI VIRT RES SHR S CPU%MEM% TIME+ Command
175982 marc 20 0 5279M 47200 8080 S 8.4 0.0 0:00.13 python3 average_time_os.py
175983 marc 20 0 5279M 47200 8080 S 8.4 0.0 0:00.13 python3 average_time_os.py
175985 marc 20 0 5279M 47200 8080 S 8.4 0.0 0:00.13 python3 average_time_os.py
175986 marc 20 0 5279M 47192 8072 R 8.4 0.0 0:00.13 python3 average_time_os.py
175985 marc 20 0 5279M 47268 8140 R 8.4 0.0 0:00.13 python3 average_time_os.py
175986 marc 20 0 5279M 47268 8140 S 8.4 0.0 0:00.13 python3 average_time_os.py
175984 marc 20 0 5279M 47200 8080 S 7.8 0.0 0:00.12 python3 average_time_os.py
175991 marc 20 0 5279M 47268 8140 R 7.8 0.0 0:00.12 python3 average_time_os.py
175992 marc 20 0 5279M 47268 8140 R 7.8 0.0 0:00.12 python3 average_time_os.py
175994 marc 20 0 5279M 47268 8140 S 7.8 0.0 0:00.12 python3 average_time_os.py
175990 marc 20 0 5279M 47268 8140 S 7.8 0.0 0:00.12 python3 average_time_os.py
176001 marc 20 0 5279M 47268 8140 R 7.8 0.0 0:00.12 python3 average_time_os.py
176003 marc 20 0 5279M 47268 8140 R 7.8 0.0 0:00.12 python3 average_time_os.py
175989 marc 20 0 5279M 47268 8140 R 7.1 0.0 0:00.11 python3 average_time_os.py
12567 marc 20 0 5279M 70608 31488 S 6.5 0.1 0:25.69 python3 average_time_os.py
175976 marc 20 0 5279M 47200 8080 S 6.5 0.0 0:00.11 python3 average_time_os.py
175970 marc 20 0 5279M 47200 8080 S 6.5 0.0 0:00.12 python3 average_time_os.py
176004 marc 20 0 5279M 47268 8140 S 6.5 0.0 0:00.10 python3 average_time_os.py
175977 marc 20 0 5279M 47192 8072 R 5.8 0.0 0:00.10 python3 average_time_os.py
175979 marc 20 0 5279M 47200 8080 S 5.8 0.0 0:00.11 python3 average_time_os.py
175980 marc 20 0 5279M 47200 8080 S 5.8 0.0 0:00.11 python3 average_time_os.py
175981 marc 20 0 5279M 47200 8080 S 5.8 0.0 0:00.12 python3 average_time_os.py
176007 marc 20 0 5279M 47268 8140 S 5.8 0.0 0:00.09 python3 average_time_os.py
175987 marc 20 0 5279M 47192 8072 S 5.2 0.0 0:00.08 python3 average_time_os.py
175988 marc 20 0 5279M 47268 8140 S 5.2 0.0 0:00.08 python3 average_time_os.py
175990 marc 20 0 5279M 47268 8140 S 5.2 0.0 0:00.08 python3 average_time_os.py
175993 marc 20 0 5279M 47268 8140 S 5.2 0.0 0:00.08 python3 average_time_os.py
175997 marc 20 0 5279M 47268 8140 S 5.2 0.0 0:00.08 python3 average_time_os.py
175999 marc 20 0 5279M 47268 8140 S 5.2 0.0 0:00.08 python3 average_time_os.py
```

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

```
marcdavis -- marc@AtlasSkyCore: ~ -- ssh skycore -- 183x44
1[|||||] 9[|||||] 17[|||||] 25[|||||]
2[|||||] 10[|||||] 18[|||||] 26[|||||]
3[|||||] 11[|||||] 19[|||||] 27[|||||]
4[|||||] 12[|||||] 20[|||||] 28[|||||]
5[|||||] 13[|||||] 21[|||||] 29[|||||]
6[|||||] 14[|||||] 22[|||||] 30[|||||]
7[|||||] 15[|||||] 23[|||||] 31[|||||]
8[|||||] 16[|||||] 24[|||||] 32[|||||]
Mem[|||||] 1.73G/12.6G Tasks: 150, 561 thr; 5 running
Swp[|||||] 0K/29.0G Load average: 2.02 31.55 36.28
Uptime: 01:15:05

PID USER PRI NI VIRT RES SHR S CPU%MEM% TIME+ Command
290252 marc 20 0 5402M 40336 9080 R 100.0 0.0 0:04.47 python3 average_time_os.py
290253 marc 20 0 5402M 40364 9080 R 100.0 0.0 0:04.33 python3 average_time_os.py
290257 marc 20 0 5402M 40900 9140 R 100.0 0.0 0:04.59 python3 average_time_os.py
290266 marc 20 0 5402M 40824 9140 R 99.5 0.0 0:06.32 python3 average_time_os.py
290283 marc 20 0 5402M 40400 9140 S 88.9 0.0 0:03.04 python3 average_time_os.py
290285 marc 20 0 5402M 40300 9140 S 85.3 0.0 0:05.27 python3 average_time_os.py
290282 marc 20 0 5402M 40900 9144 S 83.0 0.0 0:05.68 python3 average_time_os.py
290272 marc 20 0 5402M 40960 9140 S 72.5 0.0 0:06.49 python3 average_time_os.py
290269 marc 20 0 5402M 40940 9140 S 63.2 0.0 0:04.93 python3 average_time_os.py
290254 marc 20 0 5402M 40344 9080 S 57.3 0.0 0:04.28 python3 average_time_os.py
290278 marc 20 0 5402M 40920 9140 S 52.0 0.0 0:04.14 python3 average_time_os.py
290263 marc 20 0 5402M 41016 9140 S 32.9 0.0 0:05.02 python3 average_time_os.py
290276 marc 20 0 5402M 40940 9140 S 31.0 0.0 0:07.49 python3 average_time_os.py
290262 marc 20 0 5402M 41016 9140 S 30.3 0.0 0:05.63 python3 average_time_os.py
290274 marc 20 0 5402M 40440 9140 S 28.3 0.0 0:03.09 python3 average_time_os.py
290268 marc 20 0 5402M 40940 9140 S 22.4 0.0 0:05.92 python3 average_time_os.py
290260 marc 20 0 5402M 41000 9140 S 18.5 0.0 0:05.16 python3 average_time_os.py
290280 marc 20 0 5402M 40476 9140 S 9.2 0.0 0:02.91 python3 average_time_os.py
290281 marc 20 0 5402M 40456 9140 S 7.9 0.0 0:02.73 python3 average_time_os.py
290279 marc 20 0 5402M 40428 9140 S 6.6 0.0 0:04.55 python3 average_time_os.py
290267 marc 20 0 5402M 40508 9140 S 4.6 0.0 0:04.52 python3 average_time_os.py
4307 root 20 0 13012 6684 3620 R 0.7 0.0 0:32.71 htop
290068 marc 20 0 5402M 64568 32024 S 0.7 0.0 0:08.79 python3 average_time_os.py
290256 marc 20 0 5402M 40400 9080 S 0.7 0.0 0:05.91 python3 average_time_os.py
290275 marc 20 0 5402M 40428 9140 S 0.7 0.0 0:04.45 python3 average_time_os.py
1 root 20 0 165M 14584 8320 S 0.0 0.0 0:01.18 init splash
653 root 10 -1 103M 91000 90360 S 0.0 0.1 0:00.32 systemd-journald
765 root 20 0 27976 8256 4640 S 0.0 0.0 0:00.22 systemd-udev
1456 systemd-o 20 0 14028 6212 5424 S 0.0 0.0 0:03.10 systemd-ndm
```

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

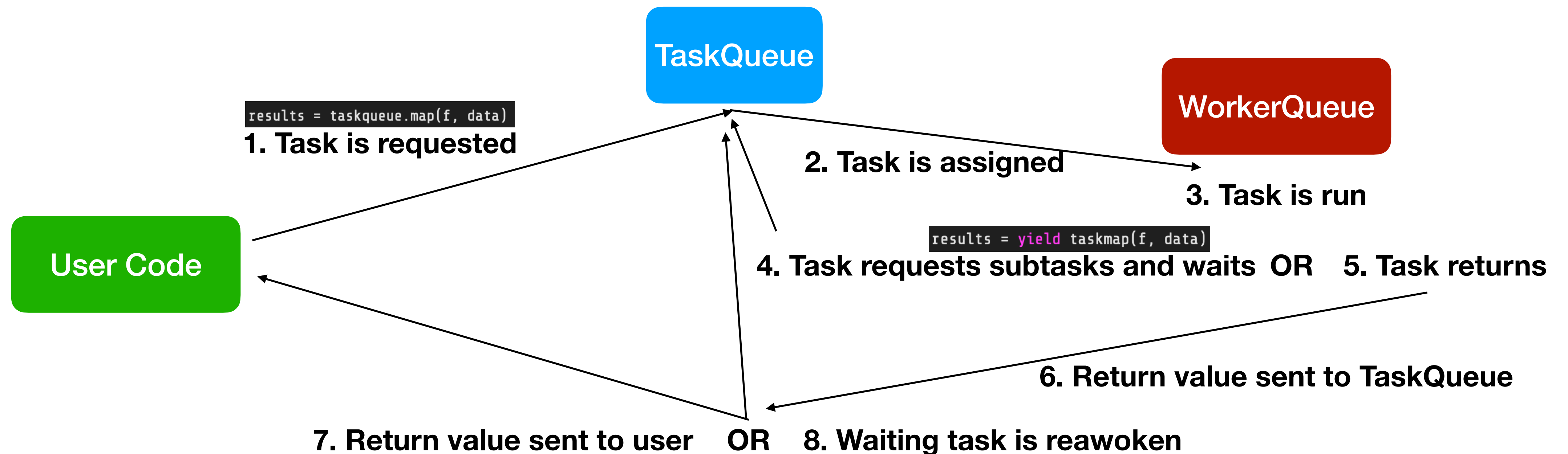
BQSKit (Successor to Qsearch):

- 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))

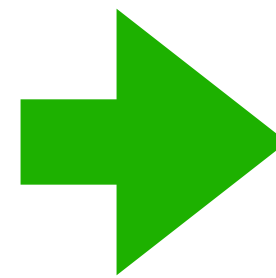
def synthesize(unitary):
    best_ansatz = prepare()
    while not is_solution(best_ansatz):
        ansatz_list = get_list_of_next_ansatzs()
        result_list = []
        for new_ansatz in ansatz_list: # Could be parallelized
            result_list.append(evaluate_ansatz(new_ansatz))

        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