Better Nested Parallel Maps in Python with YieldTasks

Marc Davis (mgd@mit.edu)

Abstract—**The parallel map operation is an important parallelization primitive which can be used as a ”parallel for” to accelerate serial code, or to build models of parallel code such as ”Map-Reduce”. In Python, several existing libraries provide parallel map functions. However, each has drawbacks. Additionally, nested parallel map operations are typically problematic. YieldTasks improves on these situations by providing a library that separates the implementation of parallelism from the expression of parallelism, allowing the user to write code once and change which library they rely on for parallelism depending on the current scenario without re-writing a significant portion of code. YieldTasks is also specifically designed to handle nested parallelism efficiently. We discuss the motivation and design decisions of YieldTasks, and describe an implementation based on Multiprocessing. We use Qsearch, a quantum gate synthesis program, as a case study, and benchmark the original Qsearch implementation (based on Multiprocessing) against one modified to use YieldTasks for better organization of parallelism. We observe significant performance improvements from using YieldTasks in this case study.**

Source code for my implementation of YieldTasks can be found at https://github.com/WolfLink/yieldtasks [8].

I. Introduction

The parallel map operation is an important parallelization primitive that is both easy to switch to from common loop structures in serial code and as a building block for other parallel models. It involves applying one function to large set of different inputs, where each function call is independent from the other function calls within the map operation.

The map operation shows up frequently in the form of iterating over a list or other collection of objects, and performing the same operation on each object. In serial code, this often takes the form of a for loop, but can also show up as a ”map” function call. Many parallelism libraries include some form of a parallel map function call. In Python, this includes ”imap_unordered” from Multiprocessing [4], ”applymap” from dask [6], and ”map” from TensorFlow [1]. Some libraries use other models of parallelism, such as Parsl ”apps” [2], or Map Reduce [11], which does provide a similar ”map” operation but it retrieves input and stores output to a filesystem, since it is designed to be used in the specific map-reduce pattern rather than as a more general ”parallel for” operation.

However, each of these libraries have their drawbacks, and users may want to use different libraries for different use cases. For example, it is often more convenient and performant to use Multiprocessing for using one computer with a few processors, such as a personal laptop, while Dask is better suited for scaling to supercomputers. It is good practice to start with small-scale proof-of-concept code before scaling up to datacenter-size operations. However, starting with a datacenter-ready library adds overhead to beginning a project.

Currently, using a particular library for parallelism involves building the codebase around that specific library. Switching to a different parallelism library generally involves rewriting every line of code that invokes a parallel map, and often additional setup, and sometimes may require specific formatting of data [1]. For these reasons, it is desirable to have one parallelism ”frontend” which allows the expression of parallelism, separate from the ”backend” which contains the implementation of parallelism. With such a library, it is possible to switch between different implementations of parallelism for different use cases without modifying significant portions of the code [2].

Another issue with common parallel map operations is when nested for loops or map operations appear. It can be a significant performance boost to parallelize each map operation, but doing so with existing parallelization libraries may cause performance detriment or even crash.

YieldTasks solves these problems by presenting a simple parallel map ”frontend” that handles nested parallelism nicely, and allows switching the parallel ”backend” without only a small modificaiton to one line of code.

In this paper, I describe the design and motivation for YieldTasks, my backend implementation based on Multiprocessing, and validate its benefits with a case study of improving the performance of Qsearch [9], a quantum gate synthesis tool which is readily parallelizable, but difficult to do so effectively with other parallelism libraries.

II. Background

The parallel map operation is a common primitive for parallelism. In Python, there are implementations of it in many popular parallelism libraries. However, there are difficulties with properly implementing parallelism in Python due to the Global Interpreter Lock (GIL) which is present in the popular CPython implementation of Python [3], [14]. Additionally, there are more general difficulties with nested parallelism, which are present even in libraries that do implement parallel map operations in Python.

A. Importance of the Parallel Map

The parallel map operation, sometimes known as a ”parallel for”, can be defined as a function $result = map(f, data)$ that takes in a function f and a collection of data $data$, and performs f on each element of $data$, and returns a collection of the results.

The parallel map operation is an important primitive for parallelism. It can be used directly as a replacement for a for loop when loop iterations are independent of each other [15]. It also can be used to build the parallelism primitives for other parallel models. For example, both the ”map” and ”reduce” operations of Map-Reduce [11] can be implemented with a parallel map operation.

B. The Global Interpreter Lock

CPython, one of the most popular implementations of Python, has a Global Interpreter Lock (GIL) [3], [14]. The GIL prevents true thread-based parallelism. While it is possible to spawn multiple threads within a Python program, the GIL only allows one thread to go through critical parts of the execution process at a time. Multithreaded Python code can see a speedup when the primary bottleneck are filesystem or network IO, or calls to certain calls to running non-Python code which release the GIL. However, more generally the GIL prevents true thread-based parallelism. This means that most Python parallelism libraries rely on creating subprocesses instead, which has a slightly higher overhead, and has other implications such as requiring any data that the new subprocess needs to be in a format that can be sent between processes [12], [14].

C. Pickling

A common way to send data between processes involves ”Pickling” [5], a standardized way for Python objects to serialize and un-serialize. The serialized data can be written to a file, sent across a pipe, or sent across a network. The Python Pickle module implements serialization for basic types (such as integers, floats, booleans, strings, and None), functions, and collections or object instances involving only pickleable data. Users can implement their own functions to serialize and unserialize their data when the default implementation is inefficient or insufficient.

Pickled objects are binary or text data, and are readily written to files, sent across pipes, or sent over networks. For example, the Multiprocessing library uses pipes to send pickled objects between processes.

D. Existing Parallelism Libraries

There are many existing parallelism libraries for Python that provide parallel map operations. However, each has its drawbacks. I will cover several popular examples.

1) Multiprocessing: Multiprocessing is a library provided with Python in the standard library. It provides two parallel map, ”Pool.imap” and ”Pool.imap_unordered”. The difference between the two is ”imap” returns outputs in the order of the input data, while ”imap_unordered” returns outputs in the order that they finished processing. Both of these operations rely on ”Multiprocessing.Pool”, which generates and manages a pool of subprocesses. Pipes are used to send pickled data between processes.

Multiprocessing is relatively lightweight and performs well, but is not suitable for nested parallelism. The subprocesses performing ”imap” operations are not able to submit more work to the pool, and creating a pool within one of these processes creates subprocesses of the subprocess, at best creating oversubscription and at worst causing crashes related to opening too many pipes or processes.

Additionally, ”Multiprocessing.Pool” does not support more complex scenarios, such as multiple networked computers or a multi-node supercomputer.

2) Dask: Dask is an advanced parallelism library for Python that provides complex configuration and is readily applicable to scalable enviornments, such as networked computers or multi-node supercomputers. However, if not configured well, it can be less performant than multiprocessing. Also, it requires use of Dask dataframes to use the built-in ”applymap” operation.

3) Tensorflow: Tensorflow is a library for accelerating linear algebra, and is particularly suited for cases involving machine learning. It can automatically run parallel code on multiple cores or on a GPU. It includes a ”map” operation, however, using it requires formatting data in a Tensorflow array. These arrays are limited to containing only numerical data, as opposed to Dask dataframes which can contain any serializable (e.g. pickleable) python object.

Each of these parallelism libraries has its benefits. Multiprocessing is the simplest and most lightweight, Dask is the most versatile, and Tensorflow supports GPUs. However, Multiprocessing and Tensorflow do not readily support nested parallelism, and Dask requires extensive configuration, and introduces a large overhead.

E. Nested Parallel Maps

In some cases, code may exhibit multiple nested for loops, or nested map operations. This can become a barrier to parallelization, because performing the operations normally involved in performing a parallel map, such as spawning subprocesses, can cause reduced performance or even cause a program to crash.

The usual recommendation is to combine multiple loops into one larger loop. This technique is known as loop collapsing [13]. For example, one way to iterate over all pairs of objects between two sets would be to have two nested loops, one that iterates over objects in one set, and one that iterates over the the other set. However, another approach would be to generate all pairs and iterate over the set of all pairs. This ”collapsed” loop performs the same set of computations, but as one large loop that is easier to parallelize as it does not involve nested parallelism. However, this requires the programmer to carefully consider the implications of loops in their code, which may not always be simple or even possible.

In a project with multiple interacting parts, it may be difficult to properly collapse loops without making assumptions about how different parts will fit together. Additionally, loops that cannot be parallelized (because iterations are not independent) within a series of nested loops can make collapsing all of the loops impossible. Examples of both of these issues are discussed in Section V.

One central issue to managing nested parallel maps is that generally, the outermost layer of code is the layer that is most appropriate to implement parallelism, but inner layers may want to express parallelism, without the layers in-between needing to facilitate communication across these abstraction boundaries.

III. YieldTasks Design

The basic principle of YieldTasks is to encapsulate running code in a way that it can be stored as data while waiting for new input data. This is accomplished through the use of Python Generators and the ”yield” keyword. A system for organizing the running of code is also required.

The YieldTasks API includes two core objects: the Task, and the TaskQueue. The TaskQueue must be subclassed with an implementation based on another parallelism library. The Task object and the TaskQueue.map function should stay compatible. This allows a user to swap parallelism ”backends” by switching which subclass of TaskQueue is used, without modifying any other code.

The ”TaskQueue.map” or ”TaskQueue.run” function should be used at the outermost layer to call into YieldTasks. All other parallel map calls can be performed with lines involving the ”yield” keyword and either user-implemented subclasses of Task, or functions wrapped as Tasks using the ”taskmap” or ”taskwrap” helper functions provided in the YieldTasks library.

A. Yield

A function that uses the ”yield” keyword returns a Generator object when called. Every time ”next” is called on the generator object, the code is run up to the next ”yield” keyword, at which point the code is paused, its state is stored, and the yielded data is returned from the ”next” call. When the function finally returns, the generator raises a ”StopIteration” error with the return value attached. The ”next” function may be called with an input argument, which will be received as a return value from the ”yield” statement in the function wrapped in the Generator object.

In this way, the ”yield” keyword enables a form of coroutine that can be used to help build a parallel map.

B. Task

The Task object encapsulates code that needs to be run, and is in charge of the expression of parallelism. It's specification is simple: arguments for its function should be stored at instantiation. It must implement a function which takes no arguments called ”run”. This function performs the encapsulated function, and may or may not include ”yield” statements. If present, each statement should yield a list of other Task objects. Those sub-Tasks will be executed, and the return values will be returned together in a list from the ”yield” statement once all of the sub-Tasks are complete. The ”run” function may return anything (or None) as a return value. If the Task is a sub-Task that was yielded by some other Task, the return value will be passed to the original Task as a return value from a ”yield” statement. Otherwise the return value will be output to user code as the return value of a ”TaskQueue.map” or ”TaskQueue.run” call.

Users may implement their own Task subclasses, but several tools are provided in the YieldTasks library to allow easier wrapping of code into Task objects. The ”Partial” subclass of Task takes a function as the first argument for initialization, and any other arguments are stored. This is similar to the ”partial” function provided by ”functools” in the Python standard library. The function that is wrapped this way may include ”yield” statements in the format described previously.

The ”taskmap” and ”taskwrap” helper functions are also provided. The ”taskwrap” function simply returns ”Partial” task as the single element in a list, for more convenient usage in ”yield” statements (which must return a *list* of Tasks). The ”taskmap” function performs the parallel map operation: it takes as input a function as the first argument, a collection of input data as the second argument, and stores the rest of the arguments. Each time the function is called, one argument from the collection of input data is passed as the first argument to the function, and the rest of the stored arguments are passed afterwards. The ”taskmap” function returns a list of Tasks that are ready to be sent via a ”yield” statement.

C. TaskQueue

The TaskQueue object must organize and execute Task objects, and is in charge of the implementation of parallelism. It also has one function that must be implemented: the ”run” function, which takes a single Task as input. The TaskQueue class in my implementation of YieldTasks includes implementations of several other functions that may be useful in implementing the ”run” function.

The ”run” function of the TaskQueue should call the ”run” function of the passed Task. If that function is a generator function, it should call ”next” on the generator and then store it in a ”waiting” queue, along with information about which Tasks it is waiting on. The yielded list of Task objects should then be ”run” in the same way. When a ”run” is not a generator function, its return value should be stored alongside data of which generator is waiting on that data. When the returned data is the last piece of data that a generator was waiting on, that generator's ”next” function should be called with a list of the returned data from all of the Tasks that it waited on. This process continues until the original Task's generator returns, and that return value is returned to the user as the output from the ”run” function.

The TaskQueue function is built around storing and resuming ”Task” objects and the resulting generators. The TaskQueue accomplishes this by first assigning each Task an ID number. When a Task's ”run” function is a generator function, a ”ResumedTask” is created. This is a placeholder object that stores the generator object and identification information about the original Task and the yielded sub-Tasks. The sub-Tasks are also given a ”waiting_id” which identifies which ”ResumedTask” needs the return value of the newly created sub-Task. When a Task produces output, the TaskQueue checks its collection of ”ResumedTask” objects to find the one with an ID matching the completed Task's ”waiting_id”. The

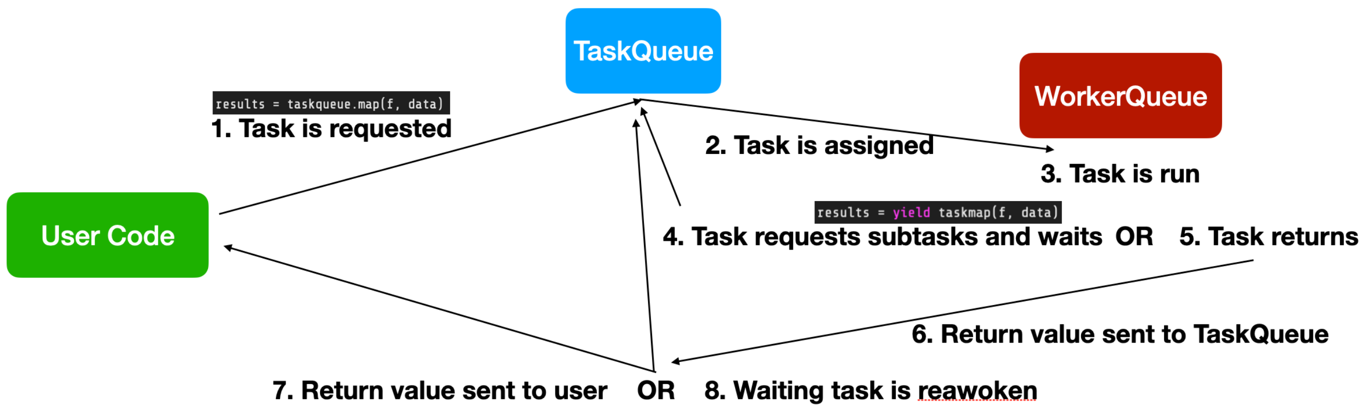

TaskQueue
WorkerQueue
User Code
results = taskqueue.map(f, data)
1. Task is requested
2. Task is assigned
3. Task is run
results = yield taskmap(f, data)
4. Task requests subtasks and waits OR 5. Task returns
6. Return value sent to TaskQueue
7. Return value sent to user OR 8. Waiting task is reawoken

Fig. 1. An image depicting the path that Tasks take during execution under the Multiprocessing-based MPQueue in YieldTasks, which is an example of the more general TaskQueue system. First, the user submits a task to the TaskQueue using either "TaskQueue.map" or "TaskQueue.run". Then, the task is assigned to a WorkerQueue, and run. The Task either returns output or yields a list of sub-Tasks. If there are sub-Tasks, they are sent to the TaskQueue to be assigned to WorkerQueues. If there is a return value, it is sent to the TaskQueue, where it is either sent to a worker with a waiting Task, potentially causing it to be reawoken, or it is returned to the user.

return value is stored on the "ResumedTask" and if this was the last needed return value, the "ResumedTask" is queued for execution. The execution of a "ResumedTask" is slightly different from that of a normal "Task" (instead of calling "run" the "next" function of the generator is called) but the behavior afterwards is the same. The only Task that does not have a "waiting_id" will be the original Task sent as input to the "run" function on the TaskQueue. When this Task (or the resulting ResumedTask) produces a return value from its "run" function, that return value will be returned to the user (as there will be no Task waiting for this data). Note that generator objects cannot be pickled, so most implementations of TaskQueue will face the restriction that the process that starts execution of a Task must maintain that generator object and perform all of that Task's execution over time.

Another type of placeholder Task called a "Placeholder-Task" is provided. This Task is meant to store the return value of another task. It's run function simply returns this stored value. It should inherit the identification information (such as the "id" and "waiting_id" of its parent).

IV. TaskQueue Implementation with Multiprocessing

For evaluation of the YieldTasks design, I implemented a YieldTasks "backend" based on Multiprocessing in the form of a "MPQueue" subclass of "TaskQueue". This implementation is available my implementation of the YieldTasks library and is available on GitHub [8]. An overview of how the Multiprocessing implementation of YieldTasks works is shown in Figure 1.

While it is more convenient to use "multiprocessing.Pool" to create a pool of workers, I need more control over the sub-processes than "Pool" provides. A critical complication is that generator objects cannot be pickled. This means that the subprocess that starts executing a Task must perform the entire execution over time. Instead, I manually create a number of "multiprocessing.Process" processes and store them as a list of workers. The main process, which runs the "run" function of the "MPQueue" is in charge of delegating work to the worker processes, communicating between the processes, and sending the final return value to the user. The worker processes run a modified "TaskQueue" called a "MPWorkerQueue" that is in charge of maintaining its list of waiting "ResumedTask" objects. Unlike a normal TaskQueue, the MPWorkerQueue waits for more Tasks from the MPQueue when it is done with its current tasks, and sends both finished return values in the form of PlaceholderTasks, and yielded Tasks to the MPQueue rather than returning them from its "run" function.

A. *MPWorkerQueue*

The MPWorkerQueue is designed to be run by a worker process. Rather than using the normal "run" function, it is started with a "start" function call, and Tasks are sent via a pipe (step 2 of Figure 1. The worker maintains its own queue of Tasks to run. For each of these tasks, it runs it as described in Section III (step 3 of Figure 1), including storing a ResumedTask and identifying Task metadata when a Task yields a list of sub-Tasks. However, instead of putting these sub-Tasks in its own queue, it sends them via pipe to the MPQueue (step 4 of Figure 1.

Because workers are in charge of maintaining Resumed-Tasks and the identifying metadata, they are also in charge of assigning IDs to Tasks, and include their own "worker_id" when they do so. This helps the MPQueue identify which data needs to go to which worker to re-queue ResumedTasks.

When a Task or ResumedTask produces a return value as output instead of yielding a list of sub-Tasks (step 5 of Figure 1, the worker first checks to see if it has the ResumedTask that needs this data. If so, it stores that data and potentially re-queues the ResumedTask as described in Section III. Otherwise, it creates a PlaceholderTask to wrap the data and sends it to the MPQueue. The MPQueue will send the PlaceholderTask to the correct worker, which will then execute the PlaceholderTask, recognize that it needs the resulting data, and process it accordingly (step 6 of Figure 1).

When the worker has finished all of the tasks in its queue, it waits to receive a new task from the MPQueue through the pipe. When it receives Tasks from the MPQueue, it repopulates its own queue and resumes work on those Tasks (step 8 of Figure 1). It continues this way until the MPQueue sends a message that lets the worker know that all Tasks are complete.

B. MPQueue

The MPQueue receives the user's initial Task (step 1 of Figure 1), as well as PlaceholderTasks and Tasks from the MPWorkerQueues and is in charge of sending them where they need to go. It does not actually execute any Tasks.

The MPQueue's run function begins with calls to Multiprocessing to create and start the worker processes and the pipes needed to communicate with them. It then enters a loop which includes sending Tasks to workers and receiving Tasks from workers, which it repeats until all workers have completed all Tasks, and no Tasks remain in the MPQueue's main queue. The MPQueue keeps track of how many Tasks it has assigned to and completed by each worker to aid in load balancing as well as part of the termination condition.

The loop starts with popping all the Tasks in its queue, and assigning them to workers (step 2 of Figure 1). It maintains data about how many tasks have been assigned to each worker, and uses this to assign new Tasks to worker with the least number of current Tasks. When the MPQueue encounters a PlaceholderTask, it instead uses the "worker_id" stored as part of the "waiting_id" to determine which worker to send it to (step 8 of Figure 1). The Tasks are sent to workers by pipe.

Once the queue is emptied, the MPQueue waits for data to become available from one of the worker's pipes. It then retrieves that data and uses it to repopulate the queue (steps 4 and 6 of Figure 1). If the received data is a PlaceholderTask with no "waiting_id" it instead stores the data to be returned to the user later.

Once the queue is empty and all workers are done, the MPQueue cleans up by sending a message to each worker that tells it to stop, and then waits for all of the sub-processe to finish. Finally, it retrieve the stored return value and returns it to the user (step 7 of Figure 1).

V. Case Study: Qsearch

Qsearch is a quantum gate synthesis tool, which is written in Python and offers an interesting opportunity for parallelization [9]. The reference implementation of Qsearch, which is available on GitHub [10] and may be installed via PyPy, includes parallelism using "multiprocessing.Pool.imap_unordered". However, this paralleization is inefficient. The specific challenges in effectively parallelizing Qsearch exemplify the benefits of YieldTasks.

A. Qsearch Background

Qsearch is a quantum gate synthesis tool, which is a technique used in quantum compiling to produce a quantum program (also known as a "quantum circuit") from a description of a desired program in the form of a unitary matrix. Quantum gate synthesis has applications in quantum circuit design and optimization [9], [10].

Qsearch performs synthesis by searching over a tree of Ansatz circuit structures. For each Ansatz circuit structure, it uses numerical optimization to find the parameters for the Ansatz circuit that best match the target unitary. Further Ansatz circuits are generated, using feedback from the numerical optimization step, and this process repeats until a sufficient match is found. An overview of this process is depicted in Figure 2.

The objective function used for numerical optimization involves simulation of small quantum circuits. This consists of a sequence of matrix multiplications. The gradient of the objective function is computed alongside the objective value by a custom implementation of forward-mode automatic differentiation that is specifically optimized to take advantage of the properties of unitary matrices, and to share computations with computing the objective as much as possible [10].

The computational cost of Qsearch primarily comes from the cost of computing the objective function for numerical optimization. This objective function is called thousands of times per optimization, and thousands of optimizations are run per synthesized unitary. Additionally, it is common to have a set of unitaries of interest rather than just one unitary to focus on.

It is worth noting that Qsearch consists of running many relatively computationally intensive function calls (quantum circuit simulation), with a relatively small variety of data being passed between function calls (the constant unitary and the different Ansatz circuit structures). This is different from the commonly-studied scenario in which a very small calculation must be performed on a large variety of data [11].

B. Opportunites for Parallelism in Qsearch

Qsearch has several opportunities for parallelism.

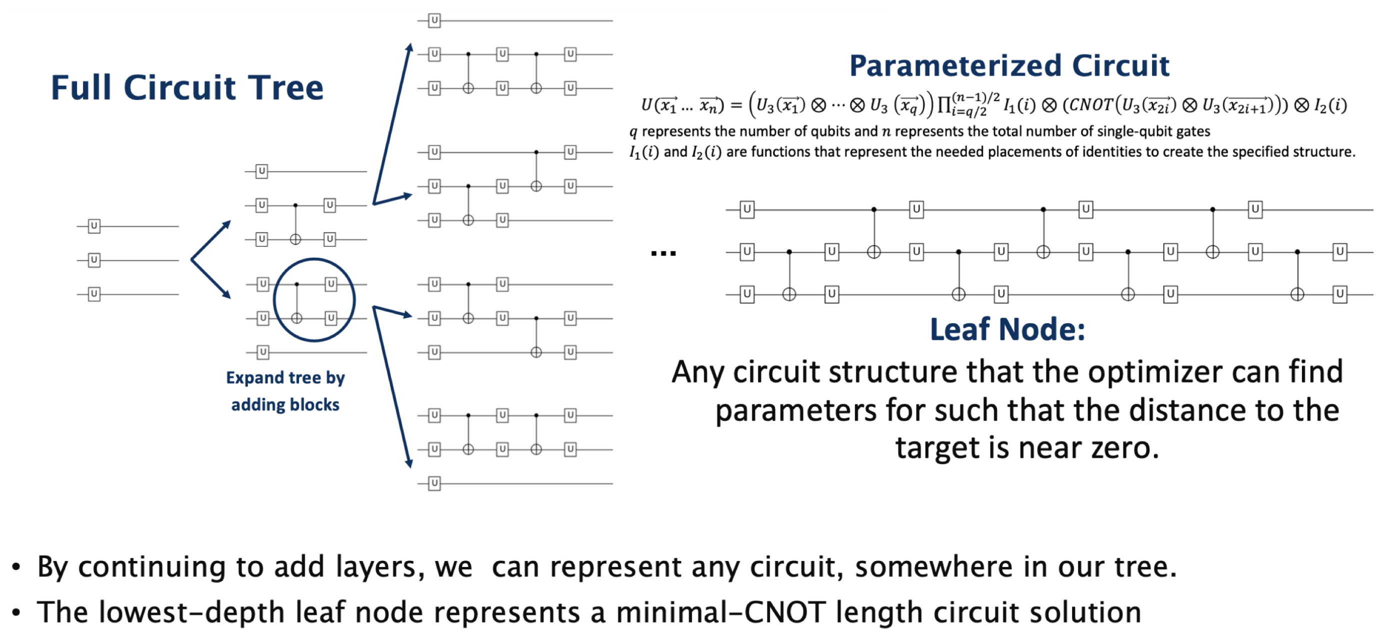
1) Multiple Unitaries: It is common to have a list of unitaries to be synthesized rather than focus on a single unitary at a time. The entire synthesis process must be run once per unitary, but each synthesis process is independent from all others.

2) Multiple Ansatz Circuits: When Qsearch generates Ansatz circuits, it typically generates multiple Ansatz circuits at a time. All of these Ansatz circuits must be numerically optimized, but the optimizations are independent of each other.

3) Multi-Start Optimization: The optimization landscape for the optimization problems involved in Qsearch have many local minima. To improve the chances of finding the global minima, Qsearch employs multi-start optimization. This involves running the numerical optimization procedure many different times from different starting points. All of these optimizations are independent of each other.

C. Difficulties with Loop Collapsing in Qsearch

The usual recommendation for handling multiple nested parallelizable loops is to collapse them into one loop. This is difficult to do in Qsearch for two reasons:



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

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.

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

Fig. 2. An image depicting the algorithm used by Qsearch to perform quantum gate synthesis. A tree of Anastz circuit structures is searched. At each node in the tree, the Ansatz circuit is instantiated by using numerical optimization to find parameters that minimize the objective function (the distance between the implemented and target unitaries). The results of this optimization are used to determine if the node is a "leaf node" (a node that represents a solution to the synthesis problem), and are also used in heuristics to guide the search.

1) Serial Loops Nested with Parallel Loops: Not all of the loops in Qsearch are parallelizable. Within each synthesis procedure, there are multiple rounds of generating set of Ansatz circuits. Each Ansatz generation round is dependent on the previous one, because the results from optimization are used to determine when a solution has been found, and in heuristics to guide the generation of future Ansatzes [9], [16]. This makes it impossible to collapse the loops of optimizing multiple Ansatz circuits and performing multi-start optimization together with the loop of synthesizing multiple unitaries.

Additionally, the numerical optimization process involves a serial loop. However, because this process forms the innermost loop, it is not a significant barrier to parallelization because it may simple be treated as one computationally expensive function call.

2) Abstraction Barriers: Qsearch is intended to be used in a wide variety of use cases, ranging from synthesis of large, individual unitaries to synthesis of many small unitaries. Qsearch has to be ready to perform one synthesis routine or many, to perform individual optimization or multi-start optimization, and to have a modified Ansatz circuit generation process.

Furthermore, there are variants of synthesis (such as targeting a discrete gate set [7]) that introduce new loops, which may or not be parallelizable, to the program.

The uncertainity of which loops will be present and parallelizable makes it difficult to write a loop-collapsed Qsearch that can handle every scenario without significant modification. The current implementation of Qsearch instead implements pieces of the program that can be swapped for variants, with only certain pieces implementing parallelism.

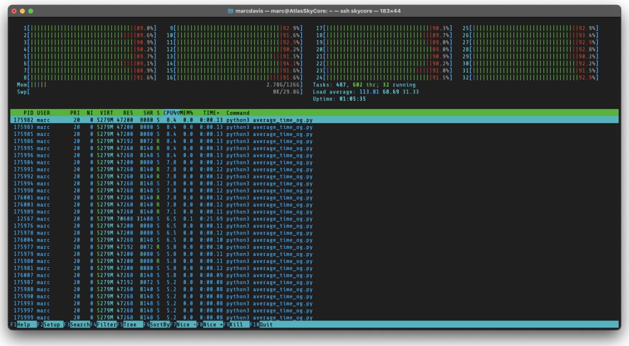
D. Parallelism in the Existing Implementation of Qsearch

The reference implementation of Qsearch [10] includes parallelism implemented via "multiprocessing.Pool.imap_unordered". This parallelism is applied at two locations: running multiple Ansatz circuits in parallel, and performing multi-start optimization. Running multiple Ansatz circuits in parallel is a productive use of parallelism in most scenarios, but can underutilize the available resources when the circuit is small and there are many available hardware threads. Performing multi-start optimization is not always applicable, but when it is used, it is always productive to do so in parallel. For CPUs with many hardware threads, multi-start optimization is unlikely to fully utilize the CPU.

It is also worth mentioning BQSKit [17], which is a quantum circuit optimization library that includes an implementation of Qsearch as well as several other quantum synthesis techniques and other quantum circuit optimization tools. BQSKit implements parallelism only at the multiple-unitary level.

Qsearch is versatile enough to tweak how parallelism is implemented. Depending on how it is configured, it will use parallelism at the multiple-Ansatz level, the multi-start level, both, or neither. (User-customized Qsearch code can introduce further paralleization). The choice of parallelization options can result in either undersubscription or oversubscription. The difference between undersubscription and oversubscription is illustrated with screenshots of "htop" in Figure 3.

1) Undersubscription: Undersubscription is when the code generates fewer parallel processes than there are available hardware threads in the CPU. This is generally sub-optimal, as there are unused CPU resources. Qsearch performs undersubscription when using just one of the its two main optimization

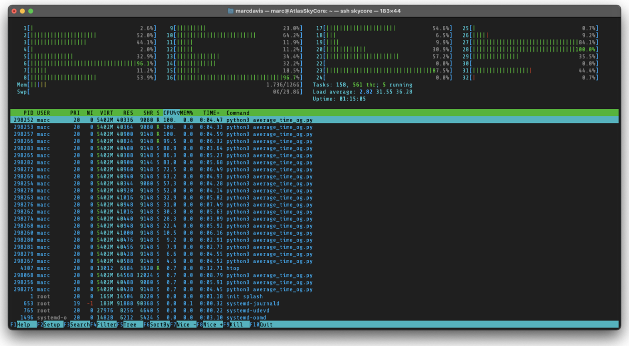


```
marcdavis — marc@AtlasSkyCore: ~ — ssh skycore — 183×44

  1[   89.0%]   9[   92.9%]  17[   90.3%]  25[   92.9%]
  2[   89.6%]  10[   91.6%]  18[   89.7%]  26[   93.4%]
  3[   90.9%]  11[   92.9%]  19[   89.0%]  27[   92.9%]
  4[   90.2%]  12[   90.2%]  20[   89.0%]  28[   92.8%]
  5[   89.7%]  13[   91.5%]  21[   90.9%]  29[   90.2%]
  6[   89.1%]  14[   94.1%]  22[   90.2%]  30[   92.2%]
  7[   88.9%]  15[   91.6%]  23[   91.0%]  31[   91.5%]
  8[   91.6%]  16[   91.6%]  24[   91.5%]  32[   92.9%]
 Mem[              2.78G/126G]   Tasks: 487, 602 thr; 32 running
 Swp[               0K/29.8G]   Load average: 113.81 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_og.py
 175983 marc        20   0 5279M 47200  8080 S  8.4  0.0  0:00.13 python3 average_time_og.py
 175985 marc        20   0 5279M 47200  8080 S  8.4  0.0  0:00.13 python3 average_time_og.py
 175986 marc        20   0 5279M 47192  8072 R  8.4  0.0  0:00.13 python3 average_time_og.py
 175995 marc        20   0 5279M 47260  8140 R  8.4  0.0  0:00.13 python3 average_time_og.py
 175996 marc        20   0 5279M 47268  8148 S  8.4  0.0  0:00.13 python3 average_time_og.py
 175984 marc        20   0 5279M 47200  8080 S  7.8  0.0  0:00.12 python3 average_time_og.py
 175991 marc        20   0 5279M 47260  8140 R  7.8  0.0  0:00.12 python3 average_time_og.py
 175992 marc        20   0 5279M 47260  8140 R  7.8  0.0  0:00.12 python3 average_time_og.py
 175994 marc        20   0 5279M 47268  8148 S  7.8  0.0  0:00.12 python3 average_time_og.py
 175998 marc        20   0 5279M 47268  8148 S  7.8  0.0  0:00.12 python3 average_time_og.py
 176001 marc        20   0 5279M 47260  8140 R  7.8  0.0  0:00.12 python3 average_time_og.py
 176003 marc        20   0 5279M 47260  8140 R  7.8  0.0  0:00.12 python3 average_time_og.py
 175989 marc        20   0 5279M 47260  8140 R  7.1  0.0  0:00.11 python3 average_time_og.py
  12567 marc        20   0 5279M 70608 31488 S  6.5  0.1  0:25.69 python3 average_time_og.py
 175976 marc        20   0 5279M 47200  8080 S  6.5  0.0  0:00.11 python3 average_time_og.py
 175978 marc        20   0 5279M 47200  8080 S  6.5  0.0  0:00.12 python3 average_time_og.py
 176004 marc        20   0 5279M 47268  8148 S  6.5  0.0  0:00.10 python3 average_time_og.py
 175977 marc        20   0 5279M 47192  8072 R  5.8  0.0  0:00.10 python3 average_time_og.py
 175979 marc        20   0 5279M 47200  8080 S  5.8  0.0  0:00.11 python3 average_time_og.py
 175980 marc        20   0 5279M 47200  8080 R  5.8  0.0  0:00.11 python3 average_time_og.py
 175981 marc        20   0 5279M 47200  8080 S  5.8  0.0  0:00.12 python3 average_time_og.py
 176007 marc        20   0 5279M 47268  8148 S  5.8  0.0  0:00.09 python3 average_time_og.py
 175987 marc        20   0 5279M 47192  8072 S  5.2  0.0  0:00.08 python3 average_time_og.py
 175988 marc        20   0 5279M 47260  8140 S  5.2  0.0  0:00.08 python3 average_time_og.py
 175990 marc        20   0 5279M 47268  8148 S  5.2  0.0  0:00.08 python3 average_time_og.py
 175993 marc        20   0 5279M 47268  8148 S  5.2  0.0  0:00.08 python3 average_time_og.py
 175997 marc        20   0 5279M 47268  8148 S  5.2  0.0  0:00.08 python3 average_time_og.py
 175999 marc        20   0 5279M 47268  8148 S  5.2  0.0  0:00.08 python3 average_time_og.py
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Oversubscription

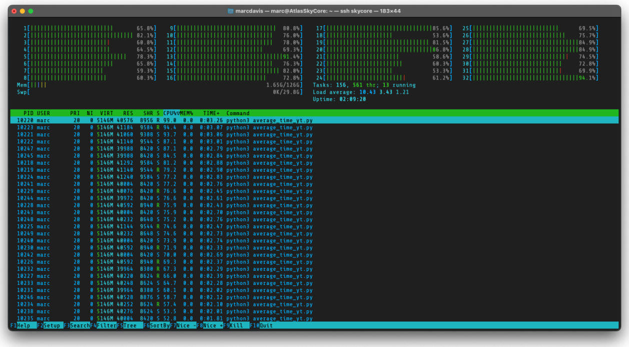


```
marcdavis — marc@AtlasSkyCore: ~ — ssh skycore — 183×44

  1[    2.6%]   9[   23.0%]  17[   54.6%]  25[    0.7%]
  2[   52.0%]  10[   64.2%]  18[    6.5%]  26[    9.2%]
  3[   44.1%]  11[   11.9%]  19[    9.9%]  27[   84.1%]
  4[    2.0%]  12[   11.2%]  20[   30.9%]  28[  100.0%]
  5[   32.9%]  13[   34.4%]  21[   57.2%]  29[   35.5%]
  6[   96.1%]  14[   32.2%]  22[    0.0%]  30[    0.0%]
  7[   11.2%]  15[   18.5%]  23[   87.5%]  31[   44.4%]
  8[   53.9%]  16[   96.7%]  24[    0.0%]  32[    0.7%]
 Mem[              1.73G/126G]   Tasks: 158, 561 thr; 5 running
 Swp[               0K/29.8G]   Load average: 2.82 31.55 36.28
                                 Uptime: 01:15:05

    PID USER       PRI  NI  VIRT   RES   SHR S CPU%▽MEM%   TIME+  Command
 298252 marc        20   0 5402M 40336  9080 R 100.  0.0  0:04.47 python3 average_time_og.py
 298253 marc        20   0 5402M 40364  9080 R 100.  0.0  0:04.33 python3 average_time_og.py
 298257 marc        20   0 5402M 40900  9148 R 100.  0.0  0:04.59 python3 average_time_og.py
 298266 marc        20   0 5402M 40824  9148 R 99.5  0.0  0:06.32 python3 average_time_og.py
 298283 marc        20   0 5402M 40480  9148 S 88.9  0.0  0:03.64 python3 average_time_og.py
 298265 marc        20   0 5402M 40388  9148 S 86.3  0.0  0:05.27 python3 average_time_og.py
 298282 marc        20   0 5402M 40900  9144 S 83.0  0.0  0:05.68 python3 average_time_og.py
 298272 marc        20   0 5402M 40960  9148 S 72.5  0.0  0:06.49 python3 average_time_og.py
 298269 marc        20   0 5402M 40940  9148 S 63.2  0.0  0:04.93 python3 average_time_og.py
 298254 marc        20   0 5402M 40344  9080 S 57.3  0.0  0:04.28 python3 average_time_og.py
 298278 marc        20   0 5402M 40920  9148 S 52.0  0.0  0:04.14 python3 average_time_og.py
 298263 marc        20   0 5402M 41016  9148 S 32.9  0.0  0:05.82 python3 average_time_og.py
 298276 marc        20   0 5402M 40948  9148 S 31.0  0.0  0:07.49 python3 average_time_og.py
 298262 marc        20   0 5402M 41016  9148 S 30.3  0.0  0:05.63 python3 average_time_og.py
 298274 marc        20   0 5402M 40440  9148 S 28.3  0.0  0:03.89 python3 average_time_og.py
 298268 marc        20   0 5402M 40948  9148 S 22.4  0.0  0:05.92 python3 average_time_og.py
 298260 marc        20   0 5402M 41000  9148 S 10.5  0.0  0:06.16 python3 average_time_og.py
 298280 marc        20   0 5402M 40476  9148 S  9.2  0.0  0:02.91 python3 average_time_og.py
 298281 marc        20   0 5402M 40456  9148 S  7.9  0.0  0:02.73 python3 average_time_og.py
 298279 marc        20   0 5402M 40428  9148 S  6.6  0.0  0:04.55 python3 average_time_og.py
 298267 marc        20   0 5402M 40588  9148 S  4.6  0.0  0:04.52 python3 average_time_og.py
   4307 marc        20   0 13012  6684  3620 R  0.7  0.0  0:32.71 htop
 298068 marc        20   0 5402M 64568 32024 S  0.7  0.0  0:08.79 python3 average_time_og.py
 298256 marc        20   0 5402M 40488  9080 S  0.7  0.0  0:05.91 python3 average_time_og.py
 298275 marc        20   0 5402M 40428  9148 S  0.7  0.0  0:04.45 python3 average_time_og.py
      1 root        20   0  165M 14504  8220 S  0.0  0.0  0:01.18 init splash
    653 root        19  -1  183M 91888 90368 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-udevd
   1496 systemd-o   20   0 14828  6212  5424 S  0.0  0.0  0:03.10 systemd-oomd
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Undersubscription



```
marcdavis — marc@AtlasSkyCore: ~ — ssh skycore — 183×44

  1[   65.8%]   9[   80.8%]  17[   85.6%]  25[   69.5%]
  2[   82.1%]  10[   76.0%]  18[   53.6%]  26[   75.7%]
  3[   60.8%]  11[   78.0%]  19[   81.5%]  27[   84.9%]
  4[   64.5%]  12[   69.3%]  20[   86.8%]  28[   84.9%]
  5[   78.3%]  13[   91.4%]  21[   58.6%]  29[   74.5%]
  6[   65.8%]  14[   76.3%]  22[   60.3%]  30[   72.8%]
  7[   59.3%]  15[   82.0%]  23[   53.3%]  31[   69.9%]
  8[   60.3%]  16[   72.8%]  24[   61.2%]  32[   94.1%]
 Mem[              1.65G/126G]   Tasks: 156, 561 thr; 13 running
 Swp[               0K/29.8G]   Load average: 10.43 3.43 1.21
                                 Uptime: 02:09:20

    PID USER       PRI  NI  VIRT   RES   SHR S CPU%▽MEM%   TIME+  Command
  10220 marc        20   0 5146M 40576  8956 R 99.0  0.0  0:03.26 python3 average_time_yt.py
  10223 marc        20   0 5146M 41184  9584 R 94.4  0.0  0:03.07 python3 average_time_yt.py
  10221 marc        20   0 5146M 41060  9388 S 93.7  0.0  0:03.06 python3 average_time_yt.py
  10222 marc        20   0 5146M 41140  9544 S 87.1  0.0  0:03.01 python3 average_time_yt.py
  10247 marc        20   0 5146M 39988  8420 S 87.1  0.0  0:02.79 python3 average_time_yt.py
  10245 marc        20   0 5146M 39988  8420 S 84.5  0.0  0:02.84 python3 average_time_yt.py
  10218 marc        20   0 5146M 41292  9584 S 81.2  0.0  0:02.88 python3 average_time_yt.py
  10219 marc        20   0 5146M 41140  9544 R 79.2  0.0  0:02.90 python3 average_time_yt.py
  10224 marc        20   0 5146M 41240  9584 S 77.2  0.0  0:02.83 python3 average_time_yt.py
  10241 marc        20   0 5146M 40004  8420 S 77.2  0.0  0:02.76 python3 average_time_yt.py
  10229 marc        20   0 5146M 40076  8420 R 76.6  0.0  0:02.45 python3 average_time_yt.py
  10244 marc        20   0 5146M 39972  8420 S 76.6  0.0  0:02.61 python3 average_time_yt.py
  10228 marc        20   0 5146M 40592  8940 R 75.9  0.0  0:02.43 python3 average_time_yt.py
  10243 marc        20   0 5146M 40004  8420 S 75.9  0.0  0:02.70 python3 average_time_yt.py
  10248 marc        20   0 5146M 40232  8648 S 75.2  0.0  0:02.76 python3 average_time_yt.py
  10225 marc        20   0 5146M 41144  9544 R 74.6  0.0  0:02.47 python3 average_time_yt.py
  10249 marc        20   0 5146M 40232  8648 S 74.6  0.0  0:02.73 python3 average_time_yt.py
  10240 marc        20   0 5146M 40004  8420 S 73.9  0.0  0:02.74 python3 average_time_yt.py
  10230 marc        20   0 5146M 40592  8940 R 71.9  0.0  0:02.33 python3 average_time_yt.py
  10242 marc        20   0 5146M 40004  8420 S 70.0  0.0  0:02.69 python3 average_time_yt.py
  10226 marc        20   0 5146M 40592  8940 R 69.3  0.0  0:02.37 python3 average_time_yt.py
  10232 marc        20   0 5146M 39964  8380 R 67.3  0.0  0:02.29 python3 average_time_yt.py
  10227 marc        20   0 5146M 40220  8624 R 66.0  0.0  0:02.39 python3 average_time_yt.py
  10233 marc        20   0 5146M 40248  8624 S 64.7  0.0  0:02.28 python3 average_time_yt.py
  10231 marc        20   0 5146M 39964  8380 S 60.1  0.0  0:02.02 python3 average_time_yt.py
  10246 marc        20   0 5146M 40528  8876 S 58.7  0.0  0:02.12 python3 average_time_yt.py
  10234 marc        20   0 5146M 40252  8624 R 57.4  0.0  0:02.10 python3 average_time_yt.py
  10238 marc        20   0 5146M 40276  8624 S 53.5  0.0  0:02.01 python3 average_time_yt.py
  10235 marc        20   0 5146M 40004  8420 S 52.8  0.0  0:01.81 python3 average_time_yt.py
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

YieldTasks

Fig. 3. Screenshots of ”htop” during the benchmarking process, illustrating the difference between undersubscription, oversubscription, and the behavior of YieldTasks (which seems to underuse the CPU, but only slightly).

options. BQSKit performs undersubscription when there are fewer unitaries to perform than the number of available hardware threads. One technique that Qsearch uses to combat undersubscription is to naively predict what Ansatzes will be needed soon so as to increase the number of Ansatzes generated at once. The extra Ansatzes generated this way are not always useful, so while this approach will better utilize a many-core CPU, it will be performing work that ends up not being useful, and wastes resources doing so.

2) *Oversubscription:* Oversubscription is when the code generates more parallel processes than there are available hardware threads in the CPU. Qsearch performs oversubscription when parallelizing on both the multiple-Ansatz and multi-start levels at the same time. When that happens, Qsearch will generate one ”multiprocessing.Pool” for parallelizing the Ansatzes, and each of those sub-processes will create its own ”Pool” of sub-sub-processes for performing multi-start. This results in a total number of working processes equal to Ansatzes $\times$ multi-starts. For example, a common scenario would be 3 Anastzes and 16 multi-starts for a total of 48 processes. This technique will fully utilize the CPU, but the OS will constantly switch between working threads, and performance will be lost due to this context switching.

E. *Accelerating Qsearch with YieldTasks*

To accelerate Qsearch with YieldTasks, I started with the Qsearch library and, in particular, the code for ”SearchCompiler”, which performs the core quantum synthesis task. I wrote a function containing the contents of ”SearchCompiler.compile” and replaced the call to ”solve_circuit_parallel” line (which calls into Qsearch’s wrapper over Multiprocessing) with an equivalent line consisting of ”yield” and ”taskmap” from YieldTasks. I also had to write a function which wraps Qsearch’s calls to numerical optimization, and a function to perform multi-start optimization with ”yield” and ”taskmap” rather than by creating a ”multiprocessing.Pool”. Finally, when I wish to synthesize multiple unitaries at a time, I do so using ”TaskQueue.map” rather than serially as one would do with the reference Qsearch implementation.

F. *Benchmarking Technique*

I ran the ”average_time.py” benchmarking script from the Qsearch GitHub repo [10], which performs a set of 8 different unitary synthesis tasks of varying difficulty 10 times each, and records the average time it took for each synthesis task. I modified the script slightly to report only the average total time for completing all 8 tasks, rather than timing each one individually, and to include 16 multi-starts during optimization.

Benchmarks were run on a computer with a Ryzen 9 5950X processor, which has 16 cores (32 threads), and 128GB of RAM.

I then wrote 3 variants of the benchmarking script: one that only uses parallelism at the multi-Ansatz level (undersubscription), one that uses parallelism at both the multi-Ansatz and multi-start levels (oversubscription), and one that uses

Technique	Time
Undersubscription	19m 58s
Oversubscription	8m 21s
YieldTasks	5m 32s

TABLE I

Summary of the results of benchmarking Qsearch re-written with YieldTasks against the reference Qsearch implementation written with Multiprocessing. The times presented are wall-clock times recorded using time ”timeit” package from the Python standard library. The task being benchmarked was to perform a set of 8 unitary synthesis tasks, using 16 mult-starts, and to repeat this task 10 times. Undersubscription refers to using the reference implementation of Qsearch and parallelizing only on the multi-Ansatz level, leaving the CPU underutilized. Oversubscription refers to using the reference implementation of Qsearch and parallelizing at both the multi-Ansatz and multi-start levels. YieldTasks refers to using a modified version of Qsearch built on YieldTasks to parallelize at the multi-Anatz, multi-start, and multi-unitary levels without oversubscribing. The benchmarks were all run on a computer with an AMD Ryzen 9 5950X CPU which has 16 cores (32 threads) and 128GB of RAM. The time presented is the total time to perform all tasks 10 times, not the average time per set of 8 tasks.

YieldTasks to parallelize at the multi-Ansatz, multi-start, and multi-unitary levels. The results are presented in Table I.

G. Benchmarking Discussion

The results from my benchmarking experiment showed that oversubscription is better than undersubscription. The undersubscription approach took 19m 58s to complete but the oversubscription approach only took 8m 21s. YieldTasks performed even better, completing the tasks in 5m 32s.

Examining screenshots of ”htop”, a program that monitors CPU usage on a per-thread basis, we can visually see the difference in behavior between oversubscription, undersubscription, and YieldTasks. These screenshots are presented in Figure 3. It appears that YieldTasks did underutilize the CPU, but only slightly.

This experiment clearly demonstrates the value of the YieldTasks approach in parallelizing a program that is difficult to parallelize with other libraries.

VI. Conclusion and Next Steps

My YieldTasks implementation was able to achieve an advantage over Multiprocessing in the difficult-to-parallelize scenario of Qsearch. It implements the parallel map operation in a way that can be nested efficiently, while also separating the expression of parallelsim from the implementation of parallelism such that it is possible to change the parallel backend without rewriting large portions of code. YieldTasks is currently publicly available on GitHub. However, there are further improvements to be made.

Examining the behavior of YieldTasks as illustrated in the ”htop” screenshot in Figure 3, it appears that YieldTasks slightly underutilized the CPU (although it still outperformed over-utilization). It is possible that there is more performance to be squeezed out of this situation. This might be fixed by improving the load-balancing algorithm in MPQueue by tracking more details about the status of workers, or by preferentially sending Tasks to the workers that are waiting on them, reducing the amount of communication that needs to occur. An investigation of the workload balance would be useful to improve this performance.

One of the core ideas of YieldTasks is to allow changing the parallelism backend without changing much code. To accomplish this, I need to make more implementations of the TaskQueue, such as one backed by Dask, which would allow running on a multi-node supercomputer, or one that takes advantage of Multiprocessing’s features to share information between processes on different computers over a network. Another possibility would be to have a highly customized TaskQueue that could perform specific Tasks on a GPU. This, combined with a TaskQueue that can delegate between multiple sub-TaskQueues, similar to the MPQueue, could enable the usage of both the CPU and GPU at the same time.

Another improvement would be to re-write the code using the ”async” and ”await” syntax from Asyncio. Asyncio does not actually enable parallelism (it enables multithreading, but is limited by the GIL), but it may be able to be combined with a parallelism backend such as Dask or Multiprocessing to enable an implementation of YieldTasks that uses the ”async” and ”await” syntax instead of the ”yield” syntax, which might look nicer to users.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Ian Foster, Michael Wilde, and Kyle Chard. Scalable parallel programming in python with parsl. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, PEARC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[3] CPython Development Team. Cpython. https://github.com/python/cpython.

[4] CPython Development Team. multiprocessing: Process-based parallelism. https://docs.python.org/3/library/multiprocessing.html.

[5] CPython Development Team. pickle: Python object serialization. https://docs.python.org/3/library/multiprocessing.html.

[6] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016.

[7] Marc G. Davis. Numerical synthesis of arbitrary multi-qubit unitaries with low t-count. Master's thesis, Massachusetts Institute of Technology, 2023.

[8] Marc G. Davis. Yieldtasks. https://github.com/WolfLink/yieldtasks, 2023.

[9] Marc G. Davis, Ethan Smith, Ana Tudor, Koushik Sen, Irfan Siddiqi, and Costin Iancu. Towards optimal topology aware quantum circuit synthesis. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 223–234, 2020.

[10] Marc G. Davis, Ethan Smith, and Ed Younis. qsearch. https://github.com/BQSKit/qsearch, 2020.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[12] Roger Eggen and Maurice Eggen. Thread and process efficiency in python, 2019. Copyright - Copyright The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp) 2019; Last updated - 2020-01-27.

[13] Adrian Jackson and Orestis Agathokleous. Dynamic loop parallelisation, 2012.

[14] Remigius Meier and Armin Rigo. A way forward in parallelising dynamic languages. In *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE*, ICOOOLPS '14, New York, NY, USA, 2014. Association for Computing Machinery.

[15] Remigius Meier and Armin Rigo. A way forward in parallelising dynamic languages. In *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE*, ICOOOLPS '14, New York, NY, USA, 2014. Association for Computing Machinery.

[16] Ethan Smith, Marc Grau Davis, Jeffrey Larson, Ed Younis, Lindsay Bassman Oftelie, Wim Lavrijsen, and Costin Iancu. Leap: Scaling numerical optimization based synthesis using an incremental approach. *ACM Transactions on Quantum Computing*, 4(1), feb 2023.

[17] Ed Younis, Ethan Smith, and Mathias Weiden. BQSKit. https://github.com/BQSKit/bqskit, 2020.