LEAGER PROGRAMMING

Anthony Chyr

UUCS-19-002

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

February 26, 2019

Abstract

Leager programming, a portmanteau of “lazy” and “eager” or “limit” and “eager,” is an evaluation strategy that mixes lazy evaluation and eager evaluation. This evaluation strategy allows iterators to precompute the next value in a separate thread, storing the result in a cache until it is needed by the caller. Leager programming often takes the form of an iterator, which alone allows data to be prefetched, and when chained together can be used to form concurrent pipelines. We found a dramatic reduction in latency on par with code written with asynchronous callbacks, while making minimal modifications to the initial sequential code.

LEAGER PROGRAMMING

by

Anthony Chyr

A senior thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Bachelor of Computer Science

School of Computing
The University of Utah

May 2019

Approved:

____________________ / _______
Matthew Flatt
Supervisor

____________________ / _______
H. James de St. Germain
Director of Undergraduate Studies
School of Computing

____________________ / _______
Ross Whitaker
Diretor
School of Computing

ABSTRACT

Leager programming, a portmanteau of "lazy" and "eager" or "limit" and "eager," is an evaluation strategy that mixes lazy evaluation and eager evaluation. This evaluation strategy allows iterators to precompute the next value in a separate thread, storing the result in a cache until it is needed by the caller. Leager programming often takes the form of an iterator, which alone allows data to be prefetched, and when chained together can be used to form concurrent pipelines. We found a dramatic reduction in latency on par with code written with asynchronous callbacks, while making minimal modifications to the initial sequential code.

CONTENTS

LIST OF CODE LISTINGS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Many programming tasks have sequential implementations that are pleasant to read, but fail to exploit the opportunities for better performance through the concurrency and parallelism inherent to the task. For example, files can be fetched from a server by issuing requests in a sequential loop, but looking ahead and prefetching the request in parallel can speed up the download by hiding the latency. Similarly, a file processing task can be expressed naturally by looping over a list of files with a sequence of steps that load one file, apply a filter, and save the file back to disk — but streaming files through separate parallel processes for those steps can improve performance using the CPU and I/O in parallel.

Programming language designers and implementers have long recognized the potential for performance improvements by parallelizing otherwise sequential operations. Fully automatic parallelization would be ideal [14], but automatic parallelization has so far succeeded only for certain kinds of problems. Instead of fully automatic parallelization, programmers can recast their problem using features such as asynchronous programming [24], futures [23], and parallel for loops [28]. In those cases, the programmer must adopt a slightly different mental model of the computation, but hopefully one that is not too far from the sequential model.

This text presents another mental model for parallel programming with a particular emphasis on staying close to sequential constructs. It focuses on exploiting the opportunities for concurrency and parallelism inherent in the evaluation of expressions by exploring the space in between two well known evaluation strategies: eager evaluation and lazy evaluation — which I call *leager programming*.

Leager programming is about when: when an expression should be evaluated. In eager evaluation, also called "greedy evaluation" or "strict evaluation," the expression is evaluated as soon as it is bound to a variable. This presents a problem, for example, when iterating over an expression that enumerates the natural numbers. In eager evaluation, the entire list is evaluated before iteration begins, which would require infinite time and memory. In lazy evaluation, also called “call-by-need,” the expression is evaluated only when it is actually used by another expression. This allows, for example, iteration over an enumeration of the natural numbers without requiring infinite space, and allows the caller to begin consuming the natural numbers without having to wait an infinite amount of time (but enumerating all elements in the iterator would still take an infinite amount of time).

Leager programming seeks to straddle the space in between these two evaluation strategies. It can be thought of as lazy evaluation that eagerly precomputes the next value, or another way to look at it is it limits the eagerness to only the next value ahead of the consuming caller. For this reason, the term, “leager,” is a portmanteau of “lazy” and “eager” or “limits” and “eager.”

The precomputation in leager programming takes place in a separate thread or threads, concurrent to the consuming caller (or callers). Structured in this way, the thread or threads tasked with producing the precomputed values can be thought of as the eager portion of the leager iterator. Likewise, the thread or threads consuming the precomputed values can be thought of as the lazy portion of the a leager iterator, and the degree to which the leager iterator is lazy or eager can be precisely controlled through thread based synchronization primitives such as locks and signals.

To demonstrate a proof of concept of this evaluation strategy and to ground the concepts and terminology, leager programming was implemented in the Python programming language, specifically CPython 3.6.4, as a library of higher order functions and classes whose source code is given in Appendix A on pg. 45. This implementation encapsulates the complexity of managing the thread overhead, and allows the programmer to convert lazy Python generators and eager Python functions (mapped over an iterator) into a leager iterator by passing them to one of the higher order functions or classes in the leager programming library.

CHAPTER 2

BACKGROUND

Leager programming straddles the space between eager evaluation and lazy evaluation. While some texts restrict the definition of eager and lazy to only refer to when the expressions passed into functions are evaluated [12], this text uses a broader definition that can be applied to all expressions, which is similar to how other texts within the Python community apply the concept [13].

2.1 Eager evaluation vs lazy evaluation

Eager evaluation, also called "greedy evaluation" or "strict evaluation," evaluates an expression as soon as it is bound to a variable. Lazy evaluation, also called "call-by-need," defers the evaluation of an expression until when it is actually used by another expression. A prototypical example that highlights the difference is to construct a function that does not use its formal argument as shown in Listing 2.1.

```
f = lambda x: None
f(1/0)
```

Listing 2.1: A prototypical test used to determine whether the language is eager or lazy.

In languages that eagerly evaluate argument expressions, when the function is called with the expression `1/0` as its argument, the expression is immediately evaluated, which results in a divide by zero exception. On the other hand, in languages that lazily evaluates argument expressions (such as in Haskell), no exception is raised; the expression is never evaluated because the function never uses its formal argument. Because argument expressions are eagerly evaluated in Python, Listing 2.1 results in a divide by zero exception.

Even though Python is an eager language, laziness can be introduced by wrapping the expression in a lambda, also called *thunking*. While the lambda itself is eagerly evaluated, the programmer can explicitly control when its contents, the original expression, is evaluated by calling the function — thus creating laziness as shown in Listing 2.2.

```
f = lambda x: None
f(lambda: 1/0)
```

Listing 2.2: Creating laziness in an eager language by *thunking* the expression.

2.2 Generators

In Python, generators can be viewed as another kind of explicit evaluation-control construct, similar to *thunks*, where a generators is created eagerly but its body is evaluated on demand (lazily). An example of these generators is given in Listing 2.3, which implements an enumeration of the natural numbers.

```
# generator function
def natural_numbers():
    i = 1
    while True:
        yield i
        i += 1

# generator expression
nat_num = (i for i in natural_numbers())
```

Listing 2.3: An enumeration of the natural numbers expressed as a generator function and as a generator expression.

To evaluate the body of a generator, both generator functions (after it has been applied) and generator expressions are iterators, which means they can be used in a `for` loop or by calling the Python built-in function `next` as shown in Listing 2.4.

```
for i in natural_numbers():
    print(i)
    if i >= 3:
        break
# 1
# 2
# 3

next(nat_num) # 1
next(nat_num) # 2
next(nat_num) # 3
```

Listing 2.4: Extracting values from generator functions and generator expressions.

2.3 The space in between eager evaluation and lazy evaluation

In eager evaluation, expressions are evaluated whether they are needed or not, and values that may not be needed are computed anyway, consuming both time and memory. This can be disastrous, as expressions that represent, for example, an enumeration of the natural numbers consume infinite time and memory if eagerly evaluated. On the other hand, lazy evaluation does eliminate the waste from computing expressions whose values may not be needed, but deferring the evaluation requires book keeping overhead in addition to losing out on the opportunity of computing the value and having it already available when it is needed.

The space in between eager evaluation and lazy evaluation exists in the time between when an expression is bound to a variable and when it is needed. Leager programming seeks to straddle this space. Similar to generators, leager programming lets the programmer explicitly delay the evaluation, but evaluation is more eager than generators, allowing some values to be computed before they are needed. This takes advantage of the opportunity of having a value already available before it is needed, while also being able to express sequences that have infinite size and limiting the waste of computing values that may not be needed.

CHAPTER 3

RELATED WORK

Improving performance by optimizing when work should be done is not novel, and numerous examples exist within computing and outside of it. While these examples implement aspects of leager programming, the goal of this text is to distill these concepts into its own library, independent of specific application, that expands on public libraries in the Python Package Index (PyPI) [20].

3.1 The Toyota Production System

The initial inspiration for leager programming came from the Toyota Production System. The Toyota Production System, also referred to as lean manufacturing, focuses on the "absolute elimination of waste" [15]. It was a response to the manufacturing practices of the early 20th century, which is often referred to as the "push" method of manufacturing that shares many similarities with eager evaluation.

In push manufacturing, parts are produced in large batches with little regard for the capacity at each stage of the manufacturing process, let alone whether the product will be wanted by the consumer in the end. This created a situation where parts piled into large warehouses, which cost factories valuable space and resulted a logistical nightmare of maintaining massive inventories that in some cases never made it to the customer.

Replace push with eager, warehouse with memory, and process with program — and the similarities between manufacturing parts and computing objects become clear. In response to push manufacturing, the Toyota Production System seeks to approach zero inventory, storing only enough raw materials for that day's production and emphasizing a "pull" system where only what is needed should be produced [15]. Leager programming is the Toyota Production System applied to computing. It seeks to maintain a cache between stages of computing only big enough for the expected demand for data. For this reason, leager programming tips to the side of lazy evaluation where only what is needed should be computed (as opposed to trying compute as much as possible within the constraints of time and memory).

3.2 Analogies in hardware and software

In digital logic design, a sequential circuit is comprised of latches that pass through a block of combinational logic back into latches [5]. This can be viewed programmatically as a block of memory feeding into a pure function back into a block of memory. Attach a clock to the circuit, and the circuit starts looking like a generator function consuming an iterator. Extend the sequential circuit into a pipeline and allow for prefetching [17], and how to apply leager programming begins to become apparent. Imperatively, leager programming works by layering a cache between function applications in the same way a sequential circuit layers latches between blocks of combinational logic.

Unix pipes redirect the standard output from one program through a cache and into the standard input of another program [27]. Graphics pipelines pass data through a series of computations with memory shared in between stages [1]. Browsers can overcome the effects of network latency by preloading web pages into caches [10]. Similar to how sequential circuits layer latches between blocks of combinational logic, these specific applications across different domains of computing all share a commonality: the layering of functionality and memory that regulates when work is done — which I claim can be parameterized and distilled into a library of its own.

3.3 Automatic parallelization and parallelism by annotation

While hardware already introduces concurrency and parallelism in machine code through instruction level parallelism [17], another approach for introducing concurrency and parallelism to larger programming constructs would be to use an automatic parallelization tool. These tools often use either compile time or run-time techniques that analyze the source code detecting dependencies and identifying sections of code that can be broken into tasks and run concurrently or in parallel. Such techniques, however, have only succeeded for certain kinds of problems. For this reason, most attention in recent years have been in tools where the programmer annotates parallelizable sections of the program in addition to providing other hints to help the automatic parallelization tool parallelize the program [14].

One such tool is OpenMP, which uses parallelism by annotation as shown in Listing 3.1. OpenMP implements multithreading through a fork-join model where starting with a master thread, which is executed sequentially, OpenMP forks a number of slave threads that divide a problem that can be run in parallel before rejoining the master thread [16]. This approach is shared by leager programming, which likewise uses both parallelism by annotation and worker threads forked from a caller thread.

```
int main()
{
    const int SIZE = 1000;
    int a[SIZE];

    #pragma omp parallel for
    for (int i = 0; i < SIZE; i++)
    {
        a[i] = i;
    }

    return 0;
}
```

Listing 3.1: Example of parallelism by annotation in OpenMP.

Where automatic parallelization tools such as OpenMP differ from leager programmming is in its evaluation strategy. Automatic parallelization tools seek to introduce parallelism into what would otherwise be sequential code, retaining the evaluation strategy of the underlying code. Leager progrmaming on the other hand seeks to introduce a new evaluation strategy that uses concurrency and parallelism.

3.4 Futures (programming construct)

Futures — also called promises, delays, deferred, and eventuals — are perhaps the programming construct most similar to leager programming. In fact, applying the definition by Prasad and others, “a future or promise can be thought of as a value that will eventually become available” [18], leager programming may be considered a type of future. More importantly, however, a future takes advantage of the time between when a value is needed, and when it can be evaluated. For example, the expression passed as an argument to a function is known (and can begin execution) at the start of a function’s execution, but may not be used until much later in the function’s execution.

Futures have a long history with unique challenges. Evolving from *thunks* (zero argument functions constructed to delay the evaluation of an expression) in argument expressions [18], futures were first implemented in the Multilisp language as an annotation [7]. One challenge facing futures is scheduling when it should be evaluated. Quite often this is done by the operating system, regulated only by limiting the number of futures that can be evaluated at a time to a particular thread pool [23].

Scheduling futures is a difficult problem — one that has been explored since the original paper on futures [2] to as recently as something Kostyukov faced while developing the finagle library at Twitter [11]. The problem with futures isn't in situations where the future performs some operation, usually blocking, and then returns. A scheduler, however simple, will still complete such tasks, even if done suboptimally. The problem is scheduling futures that depend on other futures, such as in recursive functions.

Recursive functions, such as quick sort (as shown in an example in Multilisp [7]), have an attractive property where the parallelism grows at each level of the recursion. This is a double edged sword however, and the challenge is two fold. The first is if the implementation does not restrict the number of futures active at any one time, the stack or heap could overflow as the number of futures grows at each level of recursion. The second is if the implementation does restrict the number of futures that can be active at any one time, the program could deadlock as the futures already active depend on the values of additional futures (that cannot be started due the restriction on the number of futures that can be active at any one time) to unblock. This latter problem is highlighted in Listing 3.2.

```python
from concurrent.futures import ThreadPoolExecutor
executor = ThreadPoolExecutor(max_workers=4)


def fib(n):
    assert n >= 0, 'n cannot be less than zero'
    if n < 2:
        return n
    fib_1 = executor.submit(fib, n-1)
    fib_2 = executor.submit(fib, n-2)
    return fib_1.result() + fib_2.result()
```

Listing 3.2: A recursively defined Fibonnaci function using futures. Deadlocks at `fib(5)`.

Kostyukov resolved this problem by prioritizing futures along a specific branch of a recursion tree [11]. This is analogous to resolving the recursion tree using depth first search as opposed to bread first search (which can cause the stack or heap to overflow). However, this scheduling technique, if applied too aggressively, causes any benefit arising from the concurrency that futures provide to evaporate as the recursion tree would be resolved sequentially. In contrast, leager programming side steps this problem by constraining recursion to iteration, in which the restraints on concurrency (which may be understood as a restraint on eagerness) can be better defined.

3.5 Python `concurrent.futures`

Perhaps the closest standard library to leager programming is the `concurrent.futures` module in the Python standard library [23]. In fact, early versions of the leager programming library were built using `concurrent.futures`. Similar to how leager programming precomputes the next value of an iterator in a separate thread, `concurrent.futures` execute functions concurrently in a separate thread or process. Where the two differ is deciding when such functions should be computed.

Python `concurrent.futures` leans towards eagerness, as soon as a thread or process within its thread or process pool becomes available, the function is scheduled to be computed. On the other hand, a lazy future doesn't make sense. If a program waits until the result of a function is needed, the caller would block as the function is computed, eliminating whatever benefits `concurrent.futures` would have had over sequential code.

Leager programming is in fact a form of call-by-future, but it differs from Python `concurrent.futures` by adding a small amount of intelligence in deciding when such functions should be computed. It neither waits until when the result is needed to begin computation, nor does it attempts to compute all the values. It is somewhere in between: eagerly computing until its cache is full and then waiting until the caller consumes a value before it begins computing again.

Another area where Python `concurrent.futures` differs from leager programming is Python `concurrent.futures` has two main methods for scheduling work: `submit`, which takes a function and its arguments; and `map`, which takes a function and an iterable. While `map` in Python `concurrent.futures` works roughly the same way as `lmap` in leager programming, `submit` takes only a single function and produces only a single value. As discussed in Section 3.4 on pg. 5, for expressions that produce a single value, the degree to which it is eager or lazy depends on the scheduler, and scheduling futures is a difficult problem. For this reason, leager programming uses a higher order function, `leager`, which takes a generator function that produces multiple values instead of a function that produces a single value.

3.6 `async` and `await`

Concurrency is a core feature of leager programming, which for this reason shares similarities with `async` and `await` found in many languages. First appearing in C# 5 in 2012 [3], `async` and `await` has spread to other languages such as JavaScript (ECMA-262) [6] and Python 3.5 [24]. It is a recent revival of an old concept: cooperative multitasking (also called coroutines). In contrast to futures, which achieves concurrency through preemptive multitasking with each task assigned to a separate thread, `async` and `await` achieves concurrency through cooperative multitasking by yielding control to an event loop within a single thread. This allows concurrency without the cost of context switching between threads.

`async` and `await` are implemented in the `asyncio` package in the Python Standard Library [24], and allow tasks to yield control back to an event loop through the keyword `await` for functions tagged with `async`. This is similar to how producer threads and consumer threads in leager programming are scheduled by the operating system to allow for concurrency. Where the two differ is leager programming uses its concurrency to eagerly precompute values into a cache while `asyncio` lazily finds something else to do when it hits a blocked awaitable task. For this reason, where `concurrent.futures` leans eager, `asyncio` leans lazy, and leager programming seeks to be somewhere in between.

3.7 PyPI `prefetch_generator`, `async_prefetch`, and `pythonflow`

Perhaps the closest library to leager programming is the `prefetch_generator` package in the Python Package Index (PyPI) [9] and the `async_prefetch` recipe on Nikki Bowe's blog [4]. Both `prefetch_generator` and `async_prefetch` implement a decorator for generator functions that uses a producer thread to populate a queue with precomputed values. The leager programming library provides the same feature, but in a cleaner implementation that allows garbage collection if the caller leaves scope before the generator function terminates in addition to more precise control over the number of precomputed values at any one time.

Where the leager programming library differs the most from `prefetch_generator` and `async_prefetch`, however, is that in addition to using eagerness to precompute the values of lazy generators, laziness is used to throttle the values produced by eager functions. This is done through `lmap` and `lmap_unordered` in the leager programming library where it applies an eager Python function over an iterator.

Leager programming also expands on the prefetch use case in `prefetch_generator` and `async_prefetch` to include chaining leager iterators together to form concurrent pipelines. This pushes leager programming in the direction of data flow programming similar to the `pythonflow` package in the Python Package Index (PyPI) [8] — although not to the extreme where the programmer is required to define a directed acyclic graph of operations.

CHAPTER 4

METHODS

The leager programming library is comprised of the following higher order functions and classes intended to convert lazy Python generators and eager Python functions (mapped over an iterator) into leagerly evaluated iterators:

- `leager` and `LeagerIterator`
- `lmap`
- `lmap_unordered`.

The source code for the python implementation of leager programming is given in Appendix A on pg. 45. A helper library was developed to simplify the development of decorator functions for the Python implementation of leager programming library. The source code for this helper library is given in Appendix B on pg. 50.

4.1 Python decorators

For those who may be unfamiliar with Python decorators, this section is intended to be a review. A Python decorator is pure syntactic sugar used to modify the behavior of a function or class [26]. For example, Listing 4.1 is equivalent to Listing 4.2.

```
@decorator
def decorated():
    pass
```

Listing 4.1: Python decorators.

```
def decorated():
    pass

decorated = decorator(decorated)
```

Listing 4.2: Python decorators desugared.

Decorators are defined as high order functions that receive the decorated function as an argument, and composes a new object to be returned as the decorated function's replacement. For example, Listing 4.3 defines a decorator that modifies a function such that it prints how long it runs every time it is called.

```python
from time import time


def print_runtime(func):
    def timed_func(*args, **kwargs):
        start_time = time()
        return_value = func(*args, **kwargs)
        print(time() - start_time)
        return return_value
    return timed_func
```

Listing 4.3: Example of a decorator that modifies a function such that it prints how long it runs every time it is called.

4.2 `leager` and `LeagerIterator`

Python generators are defined as functions with one or more `yield` statements, an example is shown in Listing 4.4. It is lazily evaluated, beginning execution when a value is needed in an iteration and pausing after a value has been yielded. Because Python generators are defined as functions, to modify its behavior, `leager` was defined as a higher order function intended to be used as a Python decorator.

```python
def example_generator():
    i = 1
    while True:
        yield i
        i += 1
```

Listing 4.4: Example of a lazily evaluated Python generator.

```python
from leager import *


@leager
def example_generator():
    i = 1
    while True:
        yield i
        i += 1
```

Listing 4.5: Example of a lazily evaluated Python generator converted into a leagerly evaluated Python generator.

`leager` receives the generator function as one of its arguments, and then composes a leager version of the generator function that it then uses to replace the original generator function. Thus, converting a lazy Python generator function into a leager Python generator function as shown in Listing 4.5. The degree to which a leager generator is eager or lazy can be precisely controlled by adjusting the cache size, which is passed as an optional first argument to the `leager` decorator as shown in Listing 4.6.

```python
@leager(5)
def example_generator():
    i = 1
    while True:
        yield i
        i += 1
```

Listing 4.6: Example of a leagerly evaluated generator with a larger cache.

Inside the composition, `leager` applies the generator function to acquire its iterator, which it then uses to initialize `LeagerIterator`. `LeagerIterator` maintains the cache, which takes the from of a queue, in addition to starting two daemon threads. The first is the eager portion of the leager iterator, which precomputes values until the cache is full. The second is a custom garbage collector.

If the caller leaves scope before the eager portion of the `LeagerIterator` finishes consuming the iterator as shown in Listing 4.7, thus releasing its reference, the `LeagerIterator` will still have references in two other threads — thus preventing the object from being freed by Python's garbage collector. In order to rectify this memory leak, the custom garbage collector regularly checks the number of references to the `LeagerIterator`. If the number of references drops to the number of references in the daemon threads, the custom garbage collector signals for both daemon threads to terminate, thus freeing the references and allowing the `LeagerIterator` to be freed by Python's garbage collector.

```
def example_scope():
    for i in example_generator():
        print(i)
        if i >= 3:
            return # example_generator leaves scope, eager threads terminate

example_scope()
```

Listing 4.7: Example of a leager generator leaving scope and getting garbage collected.

The two daemon threads are implemented using the `threading` package in Python's standard library [21]. The producing eager thread and the consuming caller thread maintain synchronization through Python `Lock` and `Condition` objects. A shared `Lock` maintains the consistency of the cache, while `Condition` allows the producer and consumer to notify one another when the cache has been mutated. The producer blocks when the cache is full, and is notified by the consumer when it dequeues a value. Likewise, if the cache is empty, the consumer blocks, and the producer notifies the consumer when a value becomes available.

`LeagerIterator` contains a `stop` function that signals the producing eager thread to terminate, thus reverting the behavior of a `LeagerIterator` back to being lazy as shown in Listing 4.8.

```
leager_gen = example_generator()

for i in leager_gen: # leagerly evaluated
    print(i)
    if i >= 3:
        leager_gen.stop() # stop eager threads, revert behavior back to lazy
        break

for i in leager_gen: # lazily evaluated
    print(i)
    if i >= 6:
        break
```

Listing 4.8: Reverting a leager generator back to being lazily evaluated.

While `leager` adds eagerness to inherently lazy generators to produce a leager iterator, laziness can be added to inherently eager functions (applied over an iterator) to produce a leager iterator — which is the direction `lmap` and `lmap_unordered` approaches leager programming.

4.3 lmap

Similar to Python's built-in `map`, which is lazily evaluated [25], and `imap` in the `multiprocessing` module of Python's standard library, which is eagerly evaluated [22] — `lmap` leagerly applies a function over an iterator as shown in Listing 4.9. Similar to `leager`, the degree to which `lmap` is eager or lazy can be precisely controlled by adjusting the size of the cache size, which is passed as an optional third argument.

```
from random import randint
from time import sleep
from leager import *


def square(i):
    sleep(randint(0, 5))
    return i * i

for sqr in lmap(square, range(20), 5):
    print(sqr)
    input('Press [enter] to show the next perfect square.')
```

Listing 4.9: Example use of `lmap`.

Unlike `LeagerIterator`, `lmap` does not use a single eager producer thread, but instead spawns a thread for each function application. Each function application is given an index, which is used to ensure that the order in which values are yielded matches that of the iterator `lmap` is mapped over. Because each value is associated with an index, the cache uses a dictionary instead of a queue. And because each function application takes place in its own thread, a higher degree of concurrency can be achieved when compared to `leager`.

Synchronization is maintained through `Lock` and `Condition` objects, similar to `LeagerIterator`. However, since there is no single producer thread to notify, the consumer thread spawns producer threads as needed in order to maintain the cache.

4.4 lmap_unordered

Similar to `imap_unordered` in the `multiprocessing` module of Python's standard library [22], which is eagerly evaluated — `lmap_unordered` leagerly applies a function over an iterator. Similar to `lmap`, the degree to which `lmap_unordered` is eager or lazy can be precisely controlled by adjusting the size of the cache, which is passed as an optional third argument as shown in Listing 4.10.

```
from random import randint
from time import sleep
from leager import *


def square(i):
    sleep(randint(0, 5))
    return i * i

for sqr in lmap_unordered(square, range(20), 5): # may come out of order
    print(sqr)
    input('Press [enter] to show another perfect square.')
```

Listing 4.10: Example use of `lmap_unordered`.

Unlike `lmap`, order is not maintained. No function application is given an index, and adds its value to the cache as soon as it becomes available. For this reason, `lmap_unordered` can be faster than `lmap`. Because order is not maintained, the cache uses a queue instead of a dictionary similar to `LeagerIterator`.

Synchronization is maintained similar to `lmap`.

CHAPTER 5

RESULTS

A summary of the run times comparing the different programming styles for the examples discussed in this chapter is given in Table 5.1.

Style	Prefetch example[1]	Pipeline example[2]	CPU-bound example[3]
Regular Python	0.67	6.75	2.16
Leager programming	0.12	0.86	2.16
`concurrent.futures`	0.12	0.85	2.16
`async` and `await`	0.01	0.88	2.16
`prefetch_generator`	0.12	5.63	2.16
`pythonflow`	0.67	6.76	2.16

Table 5.1: Run times in seconds comparing example programs written using different styles.

[1]See Section 5.1 on pg. 20.

[2]See Section 5.2 on pg. 28.

[3]See Section 5.3 on pg. 36.

5.1 Example: prefetching data

Leager programming can be used to prefetch data, which may dramatically improve the performance of programs with blocking operations. An example program is shown in Listing 5.1, which retrieves and prints archived articles on Wikipedia with random prime ids. This program is then accelerated using leager programming to prefetch articles in Listing 5.2, which results in a decrease in latency from 0.67 seconds to 0.12 seconds.

As shown in Listing 5.3, this decrease in latency is similarly observed when using `concurrent.futures` to accelerate the program, taking 0.12 seconds. However, `concurrent.futures` requires a more extensive rewrite of the sequential code in Listing 5.1 when compared to leager programming. Using `async` and `await` as shown in Listing 5.4 resulted in the lowest latency of 0.01 seconds by avoiding thread overhead, but requires the entire program to be rewritten using coroutines. As shown in Listing 5.5, `prefetch_generator` does not come with a `map`, but one can easily be created by composing `map` in Python with `background` in `prefetch_generator`. Using `prefetch_generator` resulted in the same latency of 0.12 seconds as leager programming. `pythonflow` is not a library aimed at improving performance or creating concurrency. It instead focuses on introducing dataflow programming to Python. For this reason, `pythonflow`, as shown in Listing 5.6 and with a latency of 0.67, did not result in any improvement when compared to the regular python for this example.

```python
from random import randint
from urllib.request import urlopen
from time import time

base_url = 'https://en.wikipedia.org/w/index.php?oldid=%s'


def is_prime(num):
    for i in range(2, num):
        if not num % i:
            return False
    return True


def random_prime():
    while True:
        num = randint(0, 1000000)
        if is_prime(num):
            yield num


def main():
    start_time = time()
    for req in map(urlopen, (base_url % oldid for oldid in random_prime())):
        print(req.read())
        print('Execution time %s seconds' % (time() - start_time))
        input('Press [enter] to show the next article.\n')
        start_time = time()


if __name__ == '__main__':
    main()
```

Listing 5.1: Example program retrieving articles on Wikipedia.

```python
from random import randint
from urllib.request import urlopen
from time import time
from leager import *

base_url = 'https://en.wikipedia.org/w/index.php?oldid=%s'


def is_prime(num):
    for i in range(2, num):
        if not num % i:
            return False
    return True


@leager
def random_prime():
    while True:
        num = randint(0, 1000000)
        if is_prime(num):
            yield num


def main():
    start_time = time()
    for req in lmap(urlopen, (base_url % oldid for oldid in random_prime())):
        print(req.read())
        print('Execution time %s seconds' % (time() - start_time))
        input('Press [enter] to show the next article.\n')
        start_time = time()


if __name__ == '__main__':
    main()
```

Listing 5.2: Using leager programming to prefetch archived articles on Wikipedia.

```python
from random import randint
from urllib.request import urlopen
from time import time
from concurrent.futures import ThreadPoolExecutor

pool = ThreadPoolExecutor()
base_url = 'https://en.wikipedia.org/w/index.php?oldid=%s'


def is_prime(num):
    for i in range(2, num):
        if not num % i:
            return False
    return True


def random_prime():
    next_num = pool.submit(randint, 0, 1000000)
    while True:
        num = next_num.result()
        next_num = pool.submit(randint, 0, 1000000)
        if is_prime(num):
            yield num


def main():
    random_prime_generator = random_prime()
    next_req = pool.submit(urlopen, base_url % next(random_prime_generator))
    while True:
        start_time = time()
        req = next_req.result()
        next_req = pool.submit(urlopen, base_url % next(random_prime_generator))
        print(req.read())
        print('Execution time %s seconds' % (time() - start_time))
        input('Press [enter] to show the next article.\n')


if __name__ == '__main__':
    main()
```

Listing 5.3: Using `concurrent.futures` to prefetch archived articles on Wikipedia.

```
from random import randint
from urllib.request import urlopen
from time import time
import asyncio

oldid_queue = asyncio.Queue(1)
request_queue = asyncio.Queue(1)
base_url = 'https://en.wikipedia.org/w/index.php?oldid=%s'


def is_prime(num):
    for i in range(2, num):
        if not num % i:
            return False
    return True


def random_prime():
    while True:
        num = randint(0, 1000000)
        if is_prime(num):
            yield num


async def oldid_producer(loop):
    rp = random_prime()
    while True:
        oldid = await loop.run_in_executor(None, lambda: next(rp))
        await oldid_queue.put(oldid)


async def request_producer(loop):
    while True:
        oldid = await oldid_queue.get()
        request = await loop.run_in_executor(None, urlopen, base_url % oldid)
        await request_queue.put(request)


async def user_prompt(loop):
    while True:
        start_time = time()
        request = await request_queue.get()
        print(request.read())
        print('Execution time %s seconds' % (time() - start_time))
        print('Press [enter] to show the next article.')
        await loop.run_in_executor(None, input)


def main():
    loop = asyncio.get_event_loop()
    loop.create_task(oldid_producer(loop))
    loop.create_task(request_producer(loop))
    loop.create_task(user_prompt(loop))
```

```
    loop.run_forever()
    loop.close()


if __name__ == '__main__':
    main()
```

Listing 5.4: Using `async` and `await` to prefetch archived articles on Wikipedia.

```python
from random import randint
from urllib.request import urlopen
from time import time
from prefetch_generator import background

pgmap = lambda f, it, max_size=1: background(max_size)(map)(f, it)
base_url = 'https://en.wikipedia.org/w/index.php?oldid=%s'

def is_prime(num):
    for i in range(2, num):
        if not num % i:
            return False
    return True


@background()
def random_prime():
    while True:
        num = randint(0, 1000000)
        if is_prime(num):
            yield num


def main():
    start_time = time()
    for req in pgmap(urlopen, (base_url % oldid for oldid in random_prime())):
        print(req.read())
        print('Execution time %s seconds' % (time() - start_time))
        input('Press [enter] to show the next article.\n')
        start_time = time()


if __name__ == '__main__':
    main()
```

Listing 5.5: Using `prefetch_generator` to prefetch archived articles on Wikipedia.

```python
from random import randint
from urllib.request import urlopen
from time import time
import pythonflow as pf

base_url = 'https://en.wikipedia.org/w/index.php?oldid=%s'


def is_prime(num):
    for i in range(2, num):
        if not num % i:
            return False
    return True


def random_prime():
    while True:
        num = randint(0, 1000000)
        if is_prime(num):
            yield num


def main():
    with pf.Graph() as graph:
        oldid = pf.placeholder(name='oldid')
        url = pf.map_(lambda oldid: base_url % oldid, oldid)
        request = pf.map_(urlopen, url)

    start_time = time()
    for req in graph(request, oldid=random_prime()):
        print(req.read())
        print('Execution time %s seconds' % (time() - start_time))
        input('Press [enter] to show the next article.\n')
        start_time = time()


if __name__ == '__main__':
    main()
```

Listing 5.6: Using `pythonflow` to retrieve archived articles on Wikipedia.

5.2 Example: building a pipeline

Leager programming can be used to form concurrent pipelines. An example program is shown in Listing 5.7 as archived articles are downloaded from Wikipedia and saved to local storage. This program is then accelerated using leager programming to form concurrent pipelines allowing for a high degree of both horizontal and vertical parallelism as shown in Listing 5.8. This results in a decrease in run time from 6.75 seconds to 0.86 seconds.

As shown in Listing 5.9, this decrease in run time is similarly observed when using `concurrent.futures`, taking 0.85 seconds, but with a more extensive rewrite of the sequential code when compared to leager programming in Listing 5.8. The decrease in runtime is also shared when using `async` and `await` as shown in Listing 5.10, taking 0.88 seconds, but it resulted in the entire program being rewritten using coroutines. `prefetch_generator`, as shown in Listing 5.11, did result in a decrease in run time taking 5.63 seconds, but the high level of horizontal concurrency found in leager programming could not be created by composing `background` in `prefetch_generator` with `map` in Python. `pythonflow`, as shown in Listing 5.12, is not a performance library and did not result in an improvement in performance when compared to regular python, taking 6.76 seconds.

```
from urllib.request import urlopen
from time import time

base_url = 'https://en.wikipedia.org/w/index.php?oldid=%s'


def make_request(oldid):
    req = urlopen(base_url % oldid)
    req.oldid = oldid
    return req


def save_request(req):
    fn = 'scratch/%s.htm' % req.oldid
    with open(fn, 'wb') as f:
        return 'fn: %s, bytes_written: %s' % (fn, f.write(req.read()))


def main():
    start_time = time()

    stage_1 = map(make_request, range(16))
    stage_2 = map(save_request, stage_1)

    for status_message in stage_2:
        print(status_message)

    print('Execution time: %s seconds' % (time() - start_time))


if __name__ == '__main__':
    main()
```

Listing 5.7: Example program downloading archived articles on Wikipedia to local storage.

```python
from urllib.request import urlopen
from time import time
from leager import *

base_url = 'https://en.wikipedia.org/w/index.php?oldid=%s'


def make_request(oldid):
    req = urlopen(base_url % oldid)
    req.oldid = oldid
    return req


def save_request(req):
    fn = 'scratch/%s.htm' % req.oldid
    with open(fn, 'wb') as f:
        return 'fn: %s, bytes_written: %s' % (fn, f.write(req.read()))


def main():
    start_time = time()

    stage_1 = lmap_unordered(make_request, range(16), 16)
    stage_2 = lmap_unordered(save_request, stage_1, 16)

    for status_message in stage_2:
        print(status_message)

    print('Execution time: %s seconds' % (time() - start_time))


if __name__ == '__main__':
    main()
```

Listing 5.8: Using leager programming to form a concurrent pipeline to download archived articles on Wikipedia to local storage.

```
from urllib.request import urlopen
from time import time
from concurrent.futures import ThreadPoolExecutor

pool = ThreadPoolExecutor()
base_url = 'https://en.wikipedia.org/w/index.php?oldid=%s'


def make_request(oldid):
    req = urlopen(base_url % oldid)
    req.oldid = oldid
    return req


def save_request(req):
    fn = 'scratch/%s.htm' % req.oldid
    with open(fn, 'wb') as f:
        return 'fn: %s, bytes_written: %s' % (fn, f.write(req.read()))


def main():
    start_time = time()

    stage_1 = pool.map(make_request, range(16))
    stage_2 = pool.map(save_request, stage_1)

    for status_message in stage_2:
        print(status_message)

    print('Execution time: %s seconds' % (time() - start_time))


if __name__ == '__main__':
    main()
```

Listing 5.9: Using `concurrent.futures` to form a concurrent pipeline to download archived articles on Wikipedia to local storage.

```
from urllib.request import urlopen
from time import time
import asyncio

OBJ_COUNT = 16
stage_1_queue = asyncio.Queue(OBJ_COUNT)
stage_2_queue = asyncio.Queue(OBJ_COUNT)
base_url = 'https://en.wikipedia.org/w/index.php?oldid=%s'


def make_request(oldid):
    req = urlopen(base_url % oldid)
    req.oldid = oldid
    return req


def save_request(req):
    fn = 'scratch/%s.htm' % req.oldid
    with open(fn, 'wb') as f:
        return 'fn: %s, bytes_written: %s' % (fn, f.write(req.read()))


async def stage(loop, func, args, out_queue):
    result = await loop.run_in_executor(None, func, *args)
    await out_queue.put(result)


async def stage_1(loop):
    for oldid in range(OBJ_COUNT):
        loop.create_task(stage(loop, make_request, (oldid,), stage_1_queue))


async def stage_2(loop):
    items_processed = 0
    while items_processed < OBJ_COUNT:
        request = await stage_1_queue.get()
        loop.create_task(stage(loop, save_request, (request,), stage_2_queue))
        items_processed += 1


async def consumer(loop):
    items_processed = 0
    while items_processed < OBJ_COUNT:
        status_message = await stage_2_queue.get()
        await loop.run_in_executor(None, print, status_message)
        items_processed += 1
    loop.stop()


def main():
    start_time = time()

    loop = asyncio.get_event_loop()
```

```
    loop.create_task(stage_1(loop))
    loop.create_task(stage_2(loop))
    loop.create_task(consumer(loop))
    loop.run_forever()
    loop.close()

    print('Execution time: %s seconds' % (time() - start_time))


if __name__ == '__main__':
    main()
```

Listing 5.10: Using `async` and `await` to form a concurrent pipeline to download archived articles on Wikipedia to local storage.

```python
from urllib.request import urlopen
from time import time
from prefetch_generator import background

pgmap = lambda f, it, max_size=1: background(max_size)(map)(f, it)
base_url = 'https://en.wikipedia.org/w/index.php?oldid=%s'


def make_request(oldid):
    req = urlopen(base_url % oldid)
    req.oldid = oldid
    return req


def save_request(req):
    fn = 'scratch/%s.htm' % req.oldid
    with open(fn, 'wb') as f:
        return 'fn: %s, bytes_written: %s' % (fn, f.write(req.read()))


def main():
    start_time = time()

    stage_1 = pgmap(make_request, range(16), 16)
    stage_2 = pgmap(save_request, stage_1, 16)

    for status_message in stage_2:
        print(status_message)

    print('Execution time: %s seconds' % (time() - start_time))


if __name__ == '__main__':
    main()
```

Listing 5.11: Using `prefetch_generator` to form a concurrent pipeline to download archived articles on Wikipedia to local storage.

```python
from urllib.request import urlopen
from time import time
import pythonflow as pf

base_url = 'https://en.wikipedia.org/w/index.php?oldid=%s'


def make_request(oldid):
    req = urlopen(base_url % oldid)
    req.oldid = oldid
    return req


def save_request(req):
    fn = 'scratch/%s.htm' % req.oldid
    with open(fn, 'wb') as f:
        return 'fn: %s, bytes_written: %s' % (fn, f.write(req.read()))


def main():
    with pf.Graph() as graph:
        oldid = pf.placeholder(name='oldid')
        stage_1 = pf.map_(make_request, oldid)
        stage_2 = pf.map_(save_request, stage_1)

    start_time = time()

    for status_message in graph(stage_2, oldid=range(16)):
        print(status_message)

    print('Execution time: %s seconds' % (time() - start_time))


if __name__ == '__main__':
    main()
```

Listing 5.12: Using `pythonflow` to download archived articles on Wikipedia to local storage.

5.3 Python Global Interpreter Lock (GIL)

An important limitation in the Python implementation of leager programming is the Python Global Interpreter Lock (GIL), which allows only one thread to execute in CPython 3.6.4 at a time [19]. This prevents any improvement for CPU bound tasks to be made as both the sequential implementation in Listing 5.13 and the leager implementation execute in about 2.16 seconds.

For the same reason, `concurrent.futures` as shown in Listing 5.15, `async` and `await` as shown in Listing 5.16, `prefetch_generator` as shown in Listing 5.17, and `pythonflow` as shown in Listing 5.18 — all resulted in the same run time as the sequential implementation, taking about 2.16 seconds.

```python
from time import time


def cpu_bound_task(i):
    n = 1000000
    while n > 0:
        n -= 1
    return i


def main():
    start_time = time()

    stage_1 = map(cpu_bound_task, range(10))
    stage_2 = map(cpu_bound_task, stage_1)

    for i in stage_2:
        print(i)

    print('Execution time: %s seconds' % (time() - start_time))


if __name__ == '__main__':
    main()
```

Listing 5.13: Example program with a CPU bound task.

```python
from time import time
from leager import *


def cpu_bound_task(i):
    n = 1000000
    while n > 0:
        n -= 1
    return i


def main():
    start_time = time()

    stage_1 = lmap_unordered(cpu_bound_task, range(10), 4)
    stage_2 = lmap_unordered(cpu_bound_task, stage_1, 4)

    for i in stage_2:
        print(i)

    print('Execution time: %s seconds' % (time() - start_time))


if __name__ == '__main__':
    main()
```

Listing 5.14: Using leager programming to break a CPU bound task into a concurrent pipeline. Due to the Global Interpreter Lock, no improvement is made.

```python
from time import time
from concurrent.futures import ThreadPoolExecutor

pool = ThreadPoolExecutor()


def cpu_bound_task(i):
    n = 1000000
    while n > 0:
        n -= 1
    return i


def main():
    start_time = time()

    stage_1 = pool.map(cpu_bound_task, range(10))
    stage_2 = pool.map(cpu_bound_task, stage_1)

    for i in stage_2:
        print(i)

    print('Execution time: %s seconds' % (time() - start_time))


if __name__ == '__main__':
    main()
```

Listing 5.15: Using `concurrent.futures` to break a CPU bound task into a concurrent pipeline. Due to the Global Interpreter Lock, no improvement is made.

```python
from time import time
import asyncio

OBJ_COUNT = 10
stage_1_queue = asyncio.Queue(OBJ_COUNT)
stage_2_queue = asyncio.Queue(OBJ_COUNT)


def cpu_bound_task(i):
    n = 1000000
    while n > 0:
        n -= 1
    return i


async def stage(loop, func, args, out_queue):
    result = await loop.run_in_executor(None, func, *args)
    await out_queue.put(result)


async def stage_1(loop):
    for i in range(OBJ_COUNT):
        loop.create_task(stage(loop, cpu_bound_task, (i,), stage_1_queue))


async def stage_2(loop):
    items_processed = 0
    while items_processed < OBJ_COUNT:
        result = await stage_1_queue.get()
        loop.create_task(stage(loop, cpu_bound_task, (result,), stage_2_queue))
        items_processed += 1


async def consumer(loop):
    items_processed = 0
    while items_processed < OBJ_COUNT:
        result = await stage_2_queue.get()
        await loop.run_in_executor(None, print, result)
        items_processed += 1
    loop.stop()


def main():
    start_time = time()

    loop = asyncio.get_event_loop()
    loop.create_task(stage_1(loop))
    loop.create_task(stage_2(loop))
    loop.create_task(consumer(loop))
    loop.run_forever()
    loop.close()

    print('Execution time: %s seconds' % (time() - start_time))
```

```


if __name__ == '__main__':
    main()
```

Listing 5.16: Using `async` and `await` to break a CPU bound task into a concurrent pipeline. Due to the Global Interpreter Lock, no improvement is made.

```python
from time import time
from prefetch_generator import background

pgmap = lambda f, it, max_size=1: background(max_size)(map)(f, it)


def cpu_bound_task(i):
    n = 1000000
    while n > 0:
        n -= 1
    return i


def main():
    start_time = time()

    stage_1 = pgmap(cpu_bound_task, range(10), 4)
    stage_2 = pgmap(cpu_bound_task, stage_1, 4)

    for i in stage_2:
        print(i)

    print('Execution time: %s seconds' % (time() - start_time))


if __name__ == '__main__':
    main()
```

Listing 5.17: Using `prefetch_generator` to break a CPU bound task into a concurrent pipeline. Due to the Global Interpreter Lock, no improvement is made.

```python
from time import time
import pythonflow as pf


def cpu_bound_task(i):
    n = 1000000
    while n > 0:
        n -= 1
    return i


def main():
    with pf.Graph() as graph:
        it = pf.placeholder(name='it')
        stage_1 = pf.map_(cpu_bound_task, it)
        stage_2 = pf.map_(cpu_bound_task, stage_1)

    start_time = time()

    for result in graph(stage_2, it=range(10)):
        print(result)

    print('Execution time: %s seconds' % (time() - start_time))


if __name__ == '__main__':
    main()
```

Listing 5.18: Using `pythonflow` decompose a CPU bound task into a directed acyclic graph of operations. Due to the Global Interpreter Lock, no improvement is made.

CHAPTER 6

CONCLUSIONS

Leager programming is about: when an expression should be evaluated. It straddles the space in between eager evaluation and lazy evaluation, improving performance by introducing concurrency into what would otherwise be sequential code.

While improving performance by optimizing when work should be done is not novel, the goal of the leager programming library is to distill these concepts into its own library, independent of specific applications, that expand on publicly available libraries in the Python Package Index (PyPI).

Built using the `threading` package in Python's standard library, it is comprised of higher order functions and classes taking the form of decorators and substitutions of well known functions in Python. Used in asynchronous callbacks, leager programming can be used to prefetch data; and chained together, it can be used to form concurrent pipelines.

APPENDIX A

PYTHON IMPLEMENTATION OF LEAGER PROGRAMMING

```python
"""Tools that intelligently combine eager evaluation and lazy evaluation."""

from collections import deque, abc
from functools import wraps
from sys import getrefcount
from threading import Lock, Condition, Thread, current_thread
from time import sleep

from funchelp import default, unwrap


__all__ = ['leager', 'LeagerIterator', 'lmap', 'lmap_unordered']


@default
@unwrap
def leager(gen_func, max_size=1):
    """
    Allow generator function to precompute the next value in a separate thread.

    >>> @leager
    ... def example_generator():
    ... from time import sleep
    ... while True:
    ... sleep(5) # example blocking operation
    ... yield

    :param gen_func: generator function
    :param max_size: maximum number of values to precompute
    :return: generator function that precomputes
    """
    @wraps(gen_func)
    def leager_gen_func(*args, **kwargs):
        return LeagerIterator(gen_func(*args, **kwargs), max_size)
    return leager_gen_func


class LeagerIterator(abc.Iterator):
    """
    Create a leager iterator object that precomputes the next value from a
    provided iterator object in a separate thread.
```

```
    """
    def __init__(self, iterable, max_size=1):
        self.DAEMON_REF_COUNT = 5 # number of references in daemon threads
        self.GC_CHECK_INTERVAL = 1 # seconds between garbage collection checks

        assert max_size > 0, 'invalid max_size'
        self.iterator = iter(iterable)
        self.max_size = max_size

        self.cache = deque()

        self.stop_signal = False
        self.worker_stopped = False

        self.mutex = Lock()
        self.consumer = Condition(self.mutex)
        self.producer = Condition(self.mutex)

        self.worker = Thread(target=self._worker, daemon=True)
        self.worker.start()

        self.gc = Thread(target=self._gc, daemon=True)
        self.gc.start()

    def stop(self):
        """
        Signal all worker threads to stop precomputing values.

        :return: None
        """
        with self.mutex:
            self._signal_stop()

    def _signal_stop(self):
        self.stop_signal = True
        self.producer.notify_all()

    def _stop_worker(self):
        self.worker_stopped = True
        self.consumer.notify_all()

    def _worker(self):
        while True:
            with self.producer:
                if self.stop_signal:
                    self._stop_worker()
                    return
                while len(self.cache) >= self.max_size:
                    self.producer.wait()
                    if self.stop_signal:
                        self._stop_worker()
                        return

            try:
```

```
                item = next(self.iterator)
            except StopIteration:
                with self.producer:
                    self._stop_worker()
                    return

            with self.producer:
                self.cache.append(item)
                self.consumer.notify()

    def _gc(self):
        while True:
            sleep(self.GC_CHECK_INTERVAL)
            with self.mutex:
                if self.worker_stopped:
                    return
                if getrefcount(self) <= self.DAEMON_REF_COUNT:
                    self._signal_stop()

    def __iter__(self):
        return self

    def __next__(self):
        with self.consumer:
            while not len(self.cache):
                if self.worker_stopped:
                    return next(self.iterator)
                self.consumer.wait()

            self.producer.notify()
            return self.cache.popleft()


class lmap(abc.Iterator):
    """
    Create a leager map that precomputes the next value in a separate thread.
    The order in which values are returned is preserved.
    """
    class _Counter:
        def __init__(self):
            self.count = 0

        def __call__(self):
            self.count += 1
            return self.count

    def __init__(self, func, iterable, max_size=1):
        assert max_size > 0, 'invalid max_size'
        self.func = func
        self.iterator = iter(iterable)
        self.max_size = max_size

        self.produced = self._Counter()
        self.consumed = self._Counter()
```

```

        self.cache = {}
        self.workers = set()

        self.mutex = Lock()
        self.consumer = Condition(self.mutex)

        with self.mutex:
            self._spawn_workers()

    def _spawn_workers(self):
        while len(self.cache) + len(self.workers) < self.max_size:
            try:
                item = next(self.iterator)
            except StopIteration:
                if not len(self.workers):
                    self.consumer.notify_all()
                return

            idx = self.produced()
            worker = Thread(target=self._worker, args=(item, idx), daemon=True)
            self.workers.add(worker)
            worker.start()

    def _worker(self, item, idx):
        value = self.func(item)
        with self.mutex:
            self.cache[idx] = value
            self.workers.remove(current_thread())
            self.consumer.notify_all()

    def __iter__(self):
        return self

    def __next__(self):
        with self.consumer:
            idx = self.consumed()
            while idx not in self.cache:
                if not self.cache and not self.workers:
                    return self.func(next(self.iterator))
                self.consumer.wait()

            return_value = self.cache.pop(idx)
            self._spawn_workers()
            return return_value


class lmap_unordered(abc.Iterator):
    """
    Create a leager map that precomputes the next value in a separate thread.
    The order in which values are returned is arbitrary.
    """
    def __init__(self, func, iterable, max_size=1):
        assert max_size > 0, 'invalid max_size'
```

```
        self.func = func
        self.iterator = iter(iterable)
        self.max_size = max_size

        self.cache = deque()
        self.workers = set()

        self.mutex = Lock()
        self.consumer = Condition(self.mutex)

        with self.mutex:
            self._spawn_workers()

    def _spawn_workers(self):
        while len(self.cache) + len(self.workers) < self.max_size:
            try:
                item = next(self.iterator)
            except StopIteration:
                if not len(self.workers):
                    self.consumer.notify_all()
                return

            worker = Thread(target=self._worker, args=(item,), daemon=True)
            self.workers.add(worker)
            worker.start()

    def _worker(self, item):
        value = self.func(item)
        with self.mutex:
            self.cache.append(value)
            self.workers.remove(current_thread())
            self.consumer.notify()

    def __iter__(self):
        return self

    def __next__(self):
        with self.consumer:
            while not len(self.cache):
                if not self.workers:
                    return self.func(next(self.iterator))
                self.consumer.wait()

            return_value = self.cache.popleft()
            self._spawn_workers()
            return return_value
```

Listing A.1: leager.py

APPENDIX B

HELPER LIBRARIES

```python
from functools import wraps

__all__ = ['default', 'unwrap']


def default(dec_func):
    """
    Allows a decorator function with parameters where all the parameters are
    optional to use the unapplied form. For example, by specifying

    >>> @default
    ... def decorator(*args, **kwargs):
    ... def real_decorator(func):
    ... @wraps(func)
    ... def wrapped(*a, **kw):
    ... func(a, kw, args, kwargs)
    ... return wrapped
    ... return real_decorator

    allows

    >>> @decorator
    ... def decorated():
    ... pass

    to be equivalent to

    >>> @decorator()
    ... def decorated():
    ... pass

    Note that this creates a situation where if the first and only parameter is
    callable, the behavior is undefined. For example, avoid

    >>> f = lambda *x: x
    >>> @decorator(f)
    ... def decorate():
    ... pass

    :param dec_func: decorator function
    :return: decorator function that will worked in the unapplied form
    """
    @wraps(dec_func)
```

```
    def default_dec(*args, **kwargs):
        return dec_func()(args[0]) \
            if len(args) == 1 and callable(args[0]) and not kwargs else \
            dec_func(*args, **kwargs)
    return default_dec


def unwrap(dec_func):
    """
    Unwraps a decorator function with parameters forcing the decorated function
    to be passed as the first argument followed by the decorator's parameters.

    For example, it allows

    >>> def decorator(*args, **kwargs):
    ... def real_decorator(func):
    ... @wraps(func)
    ... def wrapper(*a, **kw):
    ... func(a, kw, args, kwargs)
    ... return wrapper
    ... return real_decorator

    to be written as

    >>> @unwrap
    ... def decorator(func, *args, **kwargs):
    ... @wraps(func)
    ... def wrapper(*a, **kw):
    ... func(a, kw, args, kwargs)
    ... return wrapper

    reducing the nested function depth.

    :param dec_func: decorator function
    :return: unwrapped decorator function
    """
    @wraps(dec_func)
    def unwrap_dec(*args, **kwargs):
        return wraps(dec_func)(lambda func: dec_func(func, *args, **kwargs))
    return unwrap_dec
```

Listing B.1: funchelp.py

REFERENCES

[1] Akenine-Moller, T., Haines, E., and Hoffman, N. *Real-Time Rendering*, 3rd ed. CRC Press, 2008.

[2] Baker, H. C., and Hewitt, C. The Incremental Garbage Collection of Processes. *SIGPLAN Not. 12*, 8 (8 1977), 55–59.

[3] Blewett, R., and Clymer, A. *Pro Asynchronous Programming with .NET*. Apress, 2014.

[4] Bowe, N. Hiding IO latency in generators by async prefetching. http://niki.code-karma.com/2011/05/hiding-io-latency-in-generators-by-async-prefetching/, 2011.

[5] Brown, S., and Vranesic, Z. *Fundamentals of Digital Logic with Verilog Design*, 3rd ed. McGraw-Hill, New York City, 2014.

[6] Ecma Internationl. ECMAScript 2017 Language Specification (ECMA-262, 8th edition, June 2017). https://www.ecma-international.org/ecma-262/8.0/#sec-async-function-definitions, 2017.

[7] Halstead, R. H. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems (TOPLAS) 7*, 4 (October 1985), 501–538.

[8] Hoffmann, T. PyPI — pythonflow. https://pypi.org/project/pythonflow/, 2018.

[9] justheuristic. PyPI — prefetch_generator. https://pypi.org/project/prefetch_generator/, 2016.

[10] Kapadia, R., and Brinza, B. Getting to the Content You Want, Faster in IE11. https://blogs.msdn.microsoft.com/ie/2013/12/04/getting-to-the-content-you-want-faster-in-ie11/, 2013.

[11] Kostyukov, V. scale.bythebay.io: Vladimir Kostyukov, Futures: Twitter vs Scala. https://www.youtube.com/watch?v=jiYe-LdPrS0, 2017.

[12] Krishnamurthi, S. *Programming Languages: Application and Interpretation*, 2nd ed. Brown University, Providence, 2017.

[13] Lott, S. F. *Functional Python Programming: Discover the power of functional programming, generator functions, lazy evaluation, the build-in itertools library, and monads*, 1st ed. Packt Publishing, Birmingham, 2015.

[14] Midkiff, S. P. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Morgan & Claypool Publishers, 2012.

[15] Ohno, T. *Toyota Production System: Beyond Large-Scale Production*. CRC Press, New York City, 1988, ch. Starting from Need.

[16] OpenMP Architecture Review Board. *OpenMP Application Programming Interface Version 4.5.* OpenMP, November 2015.

[17] Patterson, D. A., and Hennessy, J. L. *Computer Organization and Design: The Hardware/Software Interface.* Elsevier, Oxford, 2014.

[18] Prasad, K., Patil, A., and Miller, H. *Programming Models for Distributed Computing.* http://dist-prog-book.com/chapter/2/futures.html, ch. Futures and Promises.

[19] Python Software Foundation. Initialization, Finalization, and Threads. https://docs.python.org/3/c-api/init.html, 2018.

[20] Python Software Foundation. PyPI — the Python Package Index. https://pypi.org, 2018.

[21] Python Software Foundation. The Python Standard Library: 17.1. threading — Thread-based parallelism. https://docs.python.org/3/library/threading.html, 2018.

[22] Python Software Foundation. The Python Standard Library: 17.2. multiprocessing — Process-based parallelism. https://docs.python.org/3/library/multiprocessing.html, 2018.

[23] Python Software Foundation. The Python Standard Library: 17.4. concurrent.futures — Launching parallel tasks. https://docs.python.org/3/library/concurrent.futures.html, 2018.

[24] Python Software Foundation. The Python Standard Library: 19.5. asyncio — Asynchronous I/O, event loop, coroutines and tasks. https://docs.python.org/3/library/asyncio.html, 2018.

[25] Python Software Foundation. The Python Standard Library: 2. Built-in Functions. https://docs.python.org/3/library/functions.html, 2018.

[26] Python Software Foundation. The Python Standard Library: 8. Compound statements. https://docs.python.org/3/reference/compound_stmts.html, 2018.

[27] The Linux Information Project. Pipes: A Brief Introduction. http://www.linfo.org/pipe.html, 2006.

[28] The MathWorks Inc. Parallel for loop - MATLAB parfor. https://www.mathworks.com/help/matlab/ref/parfor.htmll, 2018.