Ray Design Patterns

Created: November 2020

⭐ This is a community maintained document; suggested edits and comments are welcome!

This document is a collection of common design patterns (and anti-patterns) for Ray programs. It is meant as a handbook for both:

- New users trying to understand how to get started with Ray, and
- Advanced users trying to optimize their Ray applications

This document is not meant as an introduction to Ray. For that and any further questions that arise from this document, please refer to A Gentle Introduction to Ray, the Ray GitHub, and the Ray Slack. Highly technical users may also want to refer to the Ray 1.0 Architecture whitepaper.

The patterns below are organized into "Basic Patterns," which are commonly seen in Ray applications, and "Advanced Patterns," which are less common but may be invaluable for certain use cases.

⭐ This is a community maintained document; suggested edits and comments are welcome! **1**

Basic Patterns **2**
- Pattern: Tree of Actors 2
- Pattern: Tree of Tasks 3
- Pattern: Map and Reduce 5
- Pattern: Using ray.wait to limit the number of in flight tasks 7

Basic Antipatterns **8**
- Antipattern: Accessing Global Variable in Tasks/Actors 8
- Antipattern: Too fine-grained tasks 9
- Antipattern: Unnecessary call of ray.get in a task 11
- Antipattern: Closure capture of large / unserializable object 13
- Antipattern: Calling ray.get in a loop 14

Advanced Patterns **15**
- Pattern: Overlapping computation and communication 15
- Pattern: Fault Tolerance with Actor Checkpointing 17
- Pattern: Concurrent operations with async actor 19

Advanced Antipatterns **21**
- Antipattern: Redefining task or actor in loop 21
- Antipattern: Unnecessary stateful workers 22
- Antipattern: Processing results in submission order using ray.get 22

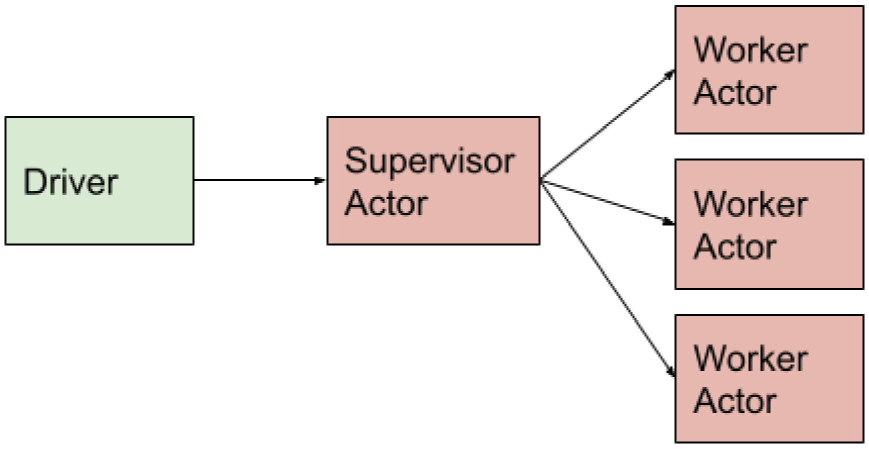
Antipattern: Fetching too many results at once with ray.get 24
More patterns (contributions welcome) 25

Basic Patterns

Pattern: Tree of Actors

Example use case
You want to train 3 models at the same time, while being able to checkpoint/inspect its state.

In this pattern, a collection of Ray worker actors is managed by a supervisory Ray actor.



A single call to the supervisor actor triggers the dispatch of multiple method calls to child actors. The supervisor can process results or update child actors prior to returning.

Notes

- If the supervisor dies (or the driver), the worker actors are automatically terminated thanks to actor reference counting.
- Actors can be nested to multiple levels to form a tree.

Code example

```
@ray.remote(num_cpus=1)
class Worker:
    def work(self):
        return "done"

@ray.remote(num_cpus=1)
class Supervisor:
    def __init__(self):
        self.workers = [Worker.remote() for _ in range(3)]
    def work(self):
        return ray.get([w.work.remote() for w in self.workers])

ray.init()
sup = Supervisor.remote()
print(ray.get(sup.work.remote()))  # outputs ['done', 'done', 'done']
```

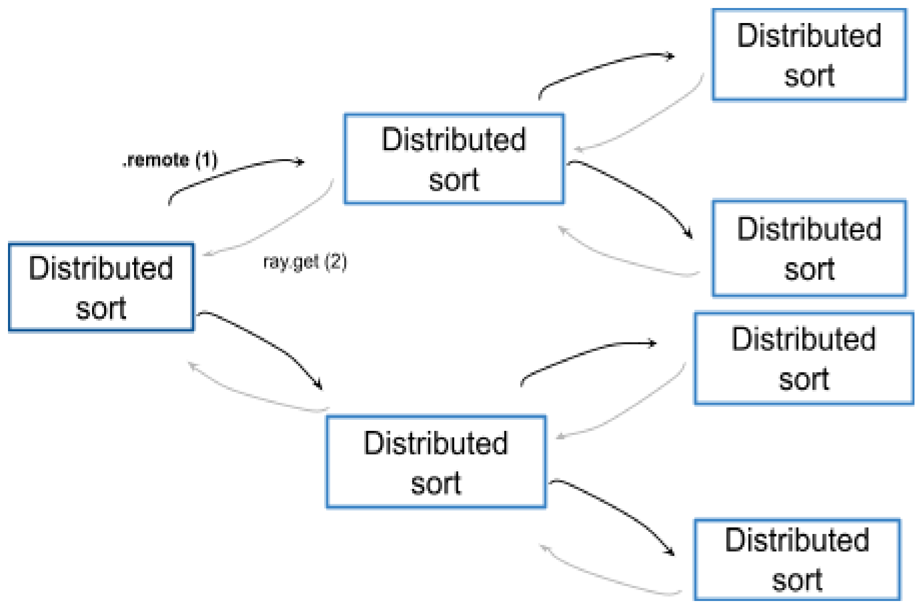
Pattern: Tree of Tasks

In this pattern, remote tasks are spawned in a recursive fashion to sort a list. Within the definition of a remote function, it is possible to invoke itself (quick_sort_distributed.remote). A single call to the task triggers the dispatch of multiple tasks.

Example use case
You have a large list of independent sub-problems that you need to solve recursively (i.e., sorting).

We call ‘ray.get’ after both ray function invocations take place. This allows you to maximize parallelism in the workload.

```
lesser = quick_sort_distributed.remote(lesser)
greater = quick_sort_distributed.remote(greater)
ray.get(lesser) + [pivot] + ray.get(greater)
```



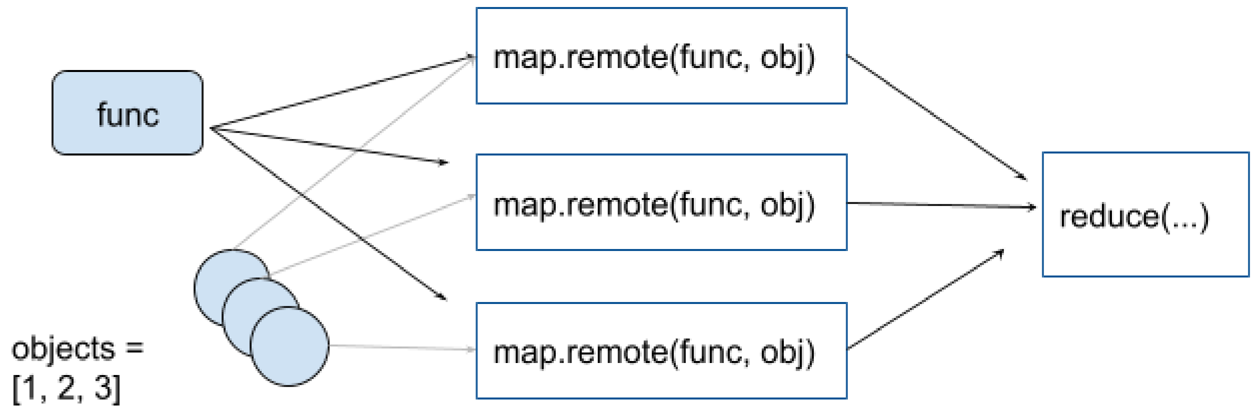
Code example

```
import ray
ray.init()
def partition(collection):
    # Use the last element as the first pivot
    pivot = collection.pop()
    greater, lesser = [], []
    for element in collection:
        if element > pivot:
            greater.append(element)
        else:
            lesser.append(element)
    return lesser, pivot, greater

@ray.remote
def quick_sort_distributed(collection):
    # Tiny tasks are an antipattern.
    # Thus, in our example we have a "magic number" to
    # toggle when distributed recursion should be used vs
    # when the sorting should be done in place. The rule
    # of thumb is that the duration of an individual task
    # should be at least 1 second.
    if len(collection) <= 200000:  # magic number
        return sorted(collection)
    else:
        lesser, pivot, greater = partition(collection)
        lesser = quick_sort_distributed.remote(lesser)
        greater = quick_sort_distributed.remote(greater)
        return ray.get(lesser) + [pivot] + ray.get(greater)

if __name__ == "__main__":
    from numpy import random
    import time
    unsorted = random.randint(1000000, size=(4000000)).tolist()
    s = time.time()
    quick_sort(unsorted)
    print("Sequential execution: " + str(time.time() - s))
    s = time.time()
    ray.get(quick_sort_distributed.remote(unsorted))
    print("Distributed execution: " + str(time.time() - s))
```

Pattern: Map and Reduce

For "map", this example uses Ray tasks to execute a given function multiple times in parallel (on a separate process). We then use ray.get to fetch the results of each of these functions.

You can have many "map" stages and many "reduce" stages.



Example use case
Implement generic map and reduce functionality with Ray tasks. "map" applies a function to a list of elements.

Code Example
Single-threaded map:

```
items = list(range(100))
```

```
map_func = lambda i : i*2
output = [map_func(i) for i in items]
```

Ray parallel map:

```
@ray.remote
def map(obj, f):
      return f(obj)

items = list(range(100))
map_func = lambda i : i*2
output = ray.get([map.remote(i, map_func) for i in items])
```

Single-threaded reduce:

```
items = list(range(100))
map_func = lambda i : i*2
output = sum([map_func(i) for i in items])
```

Ray parallel reduce:

```
@ray.remote
def map(obj, f):
      return f(obj)
@ray.remote
def sum_results(*elements):
    return np.sum(elements)

items = list(range(100))
map_func = lambda i : i*2
remote_elements = [map.remote(i, map_func) for i in items]

# simple reduce
remote_final_sum = sum_results.remote(*remote_elements)
result = ray.get(remote_final_sum)

# tree reduce
intermediate_results = [sum_results.remote(
    *remote_elements[i * 20: (i + 1) * 20]) for i in range(5)]
remote_final_sum = sum_results.remote(*intermediate_results)
result = ray.get(remote_final_sum)
```

Pattern: Using ray.wait to limit the number of in flight tasks

When you submit a ray task or actor call, Ray will make sure the data is available to the worker. However, if you submit too many tasks rapidly, the worker might be overloaded and run out of memory. You should use ray.wait to block until a certain number of tasks are ready.

Ray Serve uses this pattern to limit the number of in flight queries for each worker.

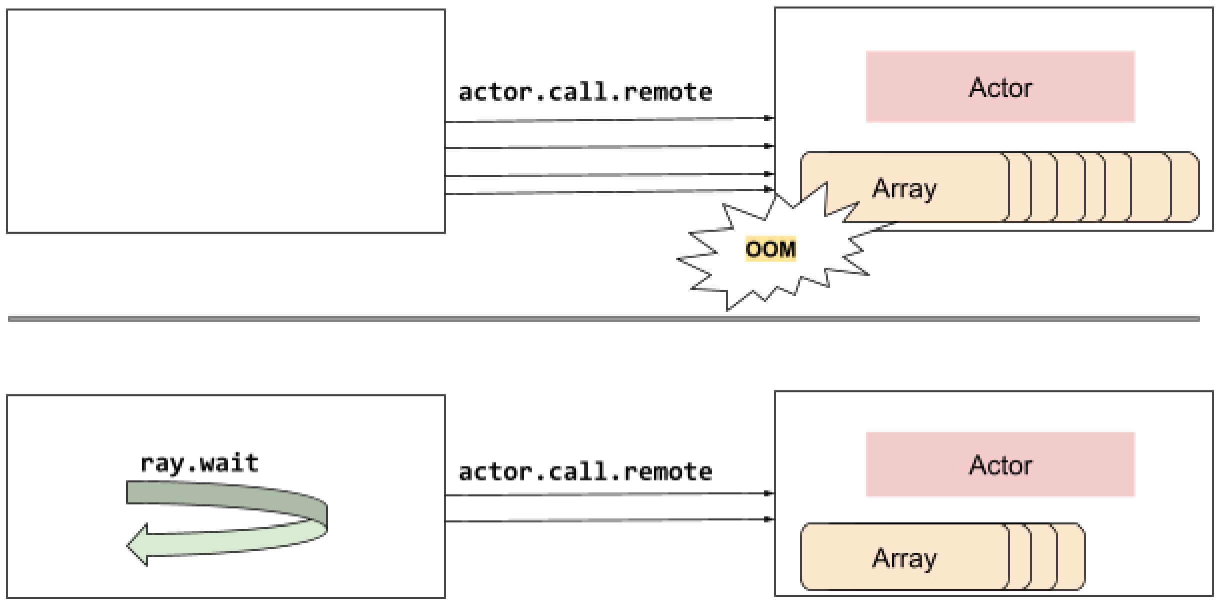
Code example

Without backpressure

```
@ray.remote
class Actor:
      def heavy_compute(self, large_array):
            # taking a long time...

actor = Actor.remote()
result_refs = []
for i in range(1_000_000):
      large_array = np.zeros(1_000_000)
      result_refs.append(actor.heavy_compute.remote(large_array))
results = ray.get(result_refs)
```

With backpressure

```
result_refs = []
for i in range(1_000_000):
      large_array = np.zeros(1_000_000)

      # Allow 1000 in flight calls
      # For example, if i = 5000, this call blocks until that
      # 4000 of the object_refs in result_refs are ready
      # and available.
      if len(result_refs) > 1000:
          num_ready = i-1000
          ray.wait(result_refs, num_returns=num_ready)

      result_refs.append(actor.heavy_compute.remote(large_array))
```



Basic Antipatterns

Antipattern: Accessing Global Variable in Tasks/Actors

TL;DR: Don't modify global variables in remote functions. Instead, encapsulate the global variables into actors.

Ray tasks and actors decorated by @ray.remote are running in different processes that don't share the same address space as ray driver (Python script that runs ray.init). That says if you define a global variable and change the value inside a driver, changes are not reflected in the workers (a.k.a tasks and actors).

Code example

Antipattern:

```
import ray
global_v = 3

@ray.remote
class A:
    def f(self):
        return global_v + 3
```

```
actor = A.remote()
global_v = 4
# This prints 6, not 7. It is because the value  change of global_v inside
a driver is not
# reflected to the actor because they are running in different processes.
print(ray.get(actor.f.remote()))
```

Better approach: Use an actor's instance variables to hold the global state that needs to be modified / accessed by multiple workers (tasks and actors).

```
import ray

@ray.remote
class GlobalVarActor:
    def __init__(self):
        self.global_v = 3
    def set_global_v(self, v):
        self.global_v = v
    def get_global_v(self):
        return self.global_v

@ray.remote
class A:
    def __init__(self, global_v_registry):
        self.global_v_registry = global_v_registry
    def f(self):
        return ray.get(self.global_v_registry.get_global_v.remote()) + 3

global_v_registry = GlobalVarActor.remote()
actor = A.remote(global_v_registry)
ray.get(global_v_registry.set_global_v.remote(4))
# This will print 7 correctly.
print(ray.get(actor.f.remote()))
```

Note that using class variables to update/manage state between instances of the same class is not currently supported. Each actor instance is instantiated across multiple processes, so each actor will have its own copy of the class variables.

Antipattern: Too fine-grained tasks

TL;DR: Avoid over-parallelizing. Parallelizing tasks has higher overhead than using normal functions.

Parallelizing or distributing tasks usually comes with higher overhead than an ordinary function call. Therefore, if you parallelize a function that executes very quickly, the overhead could take longer than the actual function call!

To handle this problem, we should be careful about parallelizing **too** much. If you have a function or task that's too small, you can use a technique called **batching** to make your tasks do more meaningful work in a single task.

Code example

Antipattern:

```python
@ray.remote
def double(number):
    return number * 2

numbers = list(range(10000))

doubled_numbers = []
for i in numbers:
    doubled_numbers.append(ray.get(double.remote(i)))
```

Better approach: Use batching.

```python
@ray.remote
def double_list(list_of_numbers):
    return [number * 2 for number in list_of_numbers]

numbers = list(range(10000))
doubled_list_refs = []
BATCH_SIZE = 100
for i in range(0, len(numbers), BATCH_SIZE):
    batch = numbers[i : i + BATCH_SIZE]
    doubled_list_refs.append(double_list.remote(batch))

doubled_numbers = []
```

```python
for ref in doubled_list_refs:
    doubled_numbers.extend(ray.get(ref))
```

Antipattern: Unnecessary call of ray.get in a task

TL;DR: Avoid calling `ray.get` too frequently/for intermediate steps. Work with object references directly, and only call `ray.get` at the end to get the final result.

When ray.get is called, objects must be transferred to the worker/node that calls `ray.get`. If you don't need to manipulate the object in a task, you probably don't need to call `ray.get` on it!

Typically, it's a best practice to wait as long as possible before calling `ray.get`, or even design your program to avoid having to call `ray.get` too soon.

Notes
Notice in the first example, we call `ray.get` which forces us to transfer the large rollout to the driver, then to "reducer" after that.

In the fixed version, we only pass the reference to the object to the "reducer". The "reducer" automatically calls `ray.get` once, which means the data is passed directly from `generate_rollout` to `reduce`, avoiding the driver.

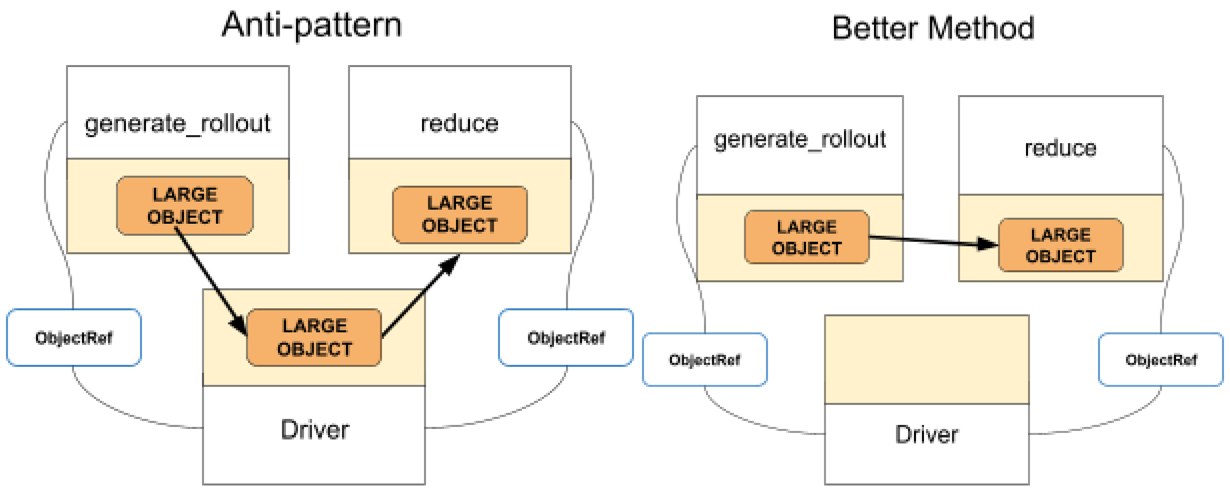
Code example

Antipattern:

```python
@ray.remote
def generate_rollout():
    return np.ones((10000, 10000))


@ray.remote
def reduce(rollout):
    return np.sum(rollout)

# `ray.get` downloads the result here.
rollout = ray.get(generate_rollout.remote())
# Now we have to reupload `rollout`
reduced = ray.get(reduce.remote(rollout))
```

Better approach:

```
# Don't need ray.get here.
rollout = generate_rollout.remote()
# Rollout object is passed by reference.
reduced = ray.get(reduce.remote(rollout))
```



Antipattern: Closure capture of large / unserializable object

TL;DR: Be careful when using large objects in `@ray.remote` functions or classes.

When you define a `ray.remote` function or class, it is easy to accidentally capture large (more than a few MB) objects implicitly in the function definition. This can lead to slow performance or `MemoryError` when attempting to define the function, since Ray is not designed to handle serialized functions or classes that are very large.

For such large objects, there are a couple options to resolve this problem:

- Use `ray.put` to put the object in the Ray object store, and then use `ray.get` to get a view of the object within the task ("better approach #1" below)
- Create the object inside the task instead of in the driver script by passing a lambda method ("better approach #2")
- The second method is the only option available for unserializable objects.

Code example

Antipattern:

```
# Create a 838 MB array, verify via: sys.getsizeof(big_array)

big_array = np.zeros(100 * 1024 * 1024)

@ray.remote
def f():
    return len(big_array) # big_array is serialized along with f!

ray.init()
ray.get(f.remote())
```

Better approach #1:

```
big_array = ray.put(np.zeros(100 * 1024 * 1024))

@ray.remote
def f():
    return len(ray.get(big_array))

ray.init()
ray.get(f.remote())
```

Better approach #2:

```
array_creator = lambda: np.zeros(100 * 1024 * 1024)

@ray.remote
def f():
    array = array_creator()
    return len(array)

ray.init()
ray.get(f.remote())
```

Antipattern: Calling ray.get in a loop

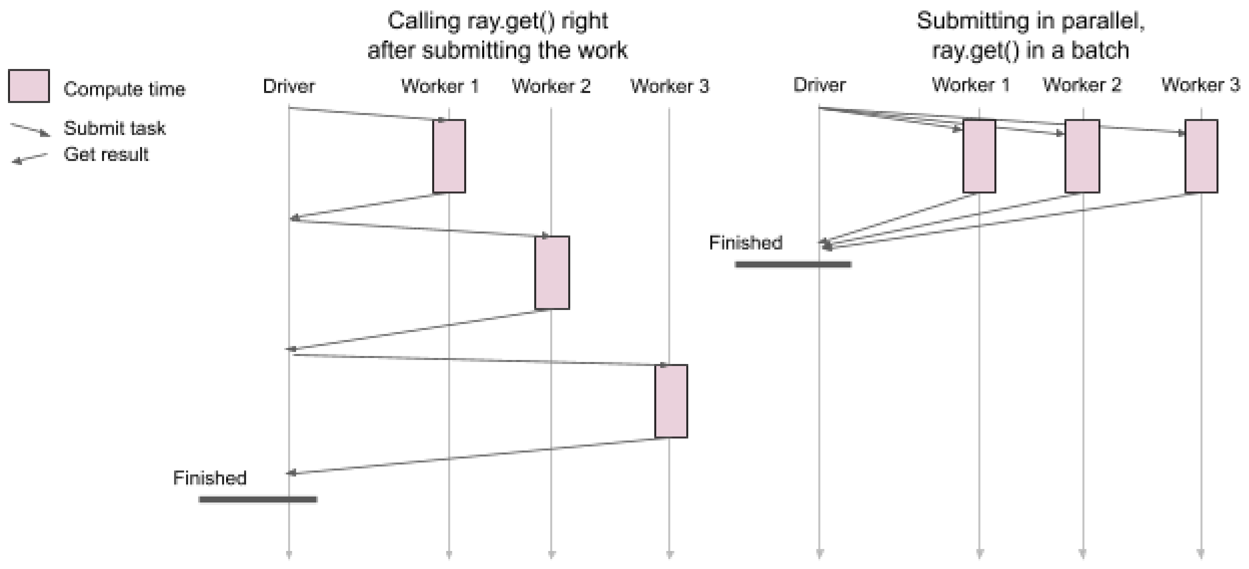
TL;DR: Avoid calling `ray.get` in a loop since it's blocking; call `ray.get` only for the final result.

A call to ray.get() fetches the results of remotely executed functions. However, it is a blocking call, which means that it always waits until the requested result is available.
If you call ray.get in a loop, the loop will not continue to run until the call to ray.get() was resolved.

If you also spawn the remote function calls in the same loop, you end up with no parallelism at all, as you wait for the previous function call to finish (because of ray.get()) and only spawn the next process in the next iteration of the loop.
The solution here is to separate the call to ray.get from the call to the remote functions. That way all remote processes are spawned before we wait for the results and can run in parallel in the background. Additionally, you can pass a list of object references to ray.get() instead of calling it one by one to wait for all of the tasks to finish.

Code example

```python
import ray
ray.init()

def f():
    pass

# Antipattern: no parallelism due to calling ray.get inside of the loop.
returns = []
for i in range(100):
    returns.append(ray.get(f.remote(i)))

# Better approach: parallelism because the tasks are spawned in parallel.
refs = []
for i in range(100):
    refs.append(f.remote(i))

returns = ray.get(refs)
```


Calling ray.get() right after submitting the work
Submitting in parallel, ray.get() in a batch
Compute time
Submit task
Get result
Driver
Worker 1
Worker 2
Worker 3
Finished

When calling ray.get() right after scheduling the remote work, the loop blocks until the item is received. We thus end up with sequential processing.
Instead, we should first schedule all remote calls, which are then processed in parallel. After scheduling the work, we can then request all the results at once.

Advanced Patterns

Pattern: Overlapping computation and communication

Sometimes a component of your application will need to do both compute-intensive work and communicate with other processes. Ideally, you want to overlap computation and communication to minimize the time spent not doing compute-intensive work.

If you block waiting for remote tasks to return, that blocking process will be idle, thereby likely reducing the overall throughput of the system.

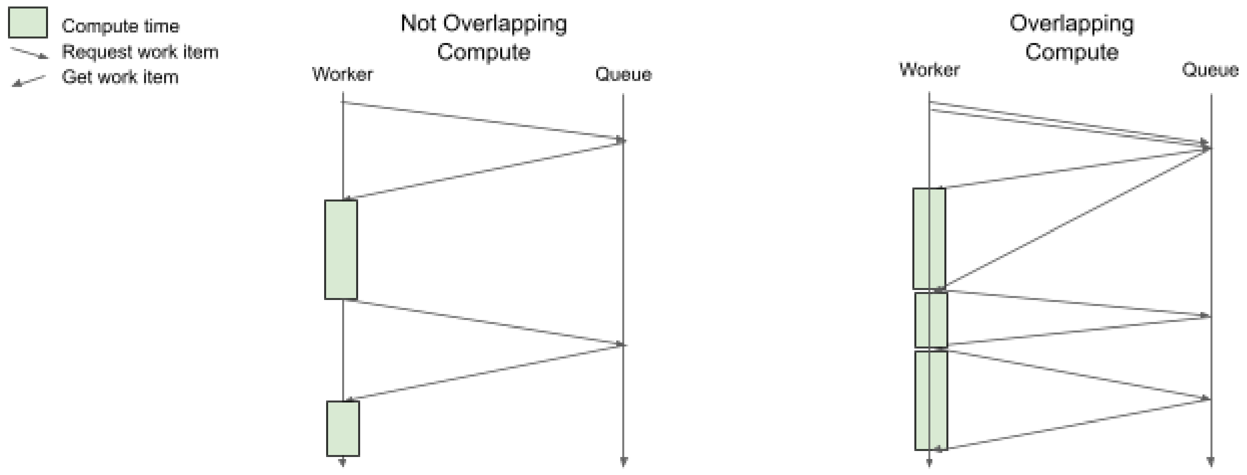
Notes
There are some cases where this behavior is not desirable. For example:

- If computing a work item takes much longer than the RTT time in the system, this is unlikely to have significant benefits.
- If the time to compute each task (or work item in this example) is highly variable, you may increase the latency for small tasks by blocking them behind large ones.

Code example

In the example below, a worker actor pulls work off of a queue and then does some computation on it. In the "bad" code example, we call ray.get() immediately after requesting a work item, so we block while that RPC is in flight, causing idle CPU time. In the corrected example, we instead preemptively request the next work item before processing the current one, so we can use the CPU while the RPC is in flight.

```python
# Bad: No overlapping of computation with communication.
@ray.remote
class Worker:
    def __init__(self, work_queue):
        self.work_queue = work_queue

    def run(self):
        while True:
            # Get work from the queue.
            work_item = ray.get(self.work_queue.get_work_item.remote())

            # Do work.
            self.process(work_item)

# Good: Overlapping computation with communication.
@ray.remote
class Worker:
    def __init__(self, work_queue):
        self.work_queue = work_queue

    def run(self):
        self.work_future = self.work_queue.get_work_item.remote()
        while True:
            # Get work from the queue.
            work_item = ray.get(self.work_future)
            self.work_future = self.work_queue.get_work_item.remote()

            # Do work.
            self.process(work_item)
```



On the left we have the first code example where we synchronously call ray.get(get_work_item.remote()) to get a new work item to process. Because we have to wait for the RPC to return the next item, we have idle periods where we are not performing computation (represented as gaps in the green boxes).

On the right, we overlap the communication and computation by spawning a task to fetch the next work item as we work on the current one. This allows us to more efficiently use the CPU because we don't have idle periods waiting for RPCs to return.

Pattern: Fault Tolerance with Actor Checkpointing

Ray offers support for task and actor fault tolerance. Specifically for actors, you can specify max_restarts to automatically enable restart for Ray actors. This means when your actor or the node hosting that actor crashed, the actor will be automatically reconstructed. However, this doesn't provide ways for you to restore application level states in your actor. You checkpoint your actor periodically and read from the checkpoint if possible.

There are several ways to checkpoint:
- Write the state to local disk. This can cause troubles when actors are instantiated in multi-node clusters.
- Write the state to local disk and use cluster launcher to sync file across cluster
- Write the state to ray internal kv store. (this is an experimental feature and not suitable for large files)
- Write the state to a Ray actor placed on head node (using custom resource constraints)

Code example

```
# max_restarts tells Ray to restart the actor infinite times
# max_task_retries tells Ray to transparently retries actor call when you
```

```
call ray.get(actor.process.remote())
@ray.remote(max_restarts=-1, max_task_retries=-1)
class ImmortalActor:
      def __init__(self):
            if os.path.exists("/tmp/checkpoint.pkl"):
                  self.state = pickle.load(open("/tmp/checkpoint.pkl"))
            else:
                  self.state = MyState()

      def process(self):
            ....
```

You can also achieve the same result just using regular Ray actors and some custom logic:

```
@ray.remote
class Worker:
      def __init__(*args, **kwargs):
            self.state = {}

      def perform_task(*args, **kwargs):
            # This task might fail.
            ...

      def get_state():
            # Returns actor state.
            return self.state


      def load_state(state):
            # Loads actor state.
            self.state = state

class Controller:
      def create_workers(num_workers):
            self.workers = [Worker.remote(...) for _ in range(num_workers)]

      def perform_task_with_fault_tol(max_retries, *args, **kwargs):
            # Perform tasks in a fault tolerant manner.
            for _ in range(max_retries):
                  worker_states = ray.get(
                      [w.get_state.remote() for w in self.workers])
                  success, result = self.perform_task_on_all_workers(
                      *args, **kwargs)
                  if success:
                        return result
                  else:
                        self.create_workers()
                        ray.get(
                            [w.load_state.remote(state)
                                  for w, state in zip(
                                      self.workers, worker_states)])
            return None


      def perform_task_on_all_workers(*args, **kwargs):
            futures = [
               w.perform_task.remote(
                   *args, **kwargs) for w in self.workers]
            output = []
            unfinished = futures
            try:
                  while len(unfinished) > 0:
                        finished, unfinished = ray.wait(unfinished)
                        output.extend(ray.get(finished))
            except RayActorError:
                  return False, None

            return True, output
```

Pattern: Concurrent operations with async actor

Sometimes, we'd like to have IO operations to other actors/tasks/components (e.g., DB) periodically within an actor (long polling). Imagine a process queue actor that needs to fetch data from other actors or DBs.

This is problematic because actors are running within a single thread. One of the solutions is to use a background thread within an actor, but you can also achieve this by using Ray's async actors APIs.

Let's see why it is difficult by looking at an example.

```
class LongPollingActor:
    def __init__(self, data_store_actor):
        self.data_store_actor = data_store_actor

    def run(self):
        while True:
            data = ray.get(self.data_store_actor.fetch.remote())
            self._process(data)

    def other_task(self):
        return True

    def _process(self, data):
        # Do process here...
        pass
```

There are 2 issues here. 1. Since a long polling actor has a run method that runs forever with while True, it cannot run any other actor task (because the thread is occupied by the while loop). That says

```
l = long_polling_actor.remote()
# Actor runs a while loop
l.run.remote()
# This won't be processed forever because the actor thread is occupied by
the run method.
ray.get(l.other_task.remote())
```

2. Since we need to call ray.get within a loop, the loop is blocked until ray.get returns (it is because ray.get is a blocking API).

We can make this better if we use Ray's async APIs. Here is a documentation about ray's async APIs and async actors.

First, let's create an async actor.

```
class LongPollingActorAsync:
    def __init__(self, data_store_actor):
        self.data_store_actor = data_store_actor

    async def run(self):
        while True:
            # Coroutine will switch context when "await" is called.
            data = await data_store_actor.fetch.remote()
            self._process(data)

    def _process(self):
        pass

    async def other_task(self):
        return True
```

Now, it will work if you run the same code we used before.

```
l = LongPollingActorAsync.remote()
l.run.remote()
ray.get(l.other_task.remote())
```

Now, let’s learn why this works. When an actor contains async methods, the actor will be converted to async actors. This means all the ray’s tasks will run as a coroutine. That says, when it meets the `await` keyword, the actor will switch to a different coroutine, which is a coroutine that runs `other_task` method.

You can implement interesting actors using this pattern. Note that it is also possible to switch context easily if you use `await asyncio.sleep(0)` without any delay.

Advanced Antipatterns

Antipattern: Redefining task or actor in loop

TL;DR: Limit the number of times you re-define a remote function.

Decorating the same function or class multiple times using the ray.remote decorator leads to slow performance in Ray. This is because Ray has to export all the function and class definitions to all Ray workers.

Instead, define tasks and actors outside of the loop instead multiple times inside a loop.

Code Example
Antipattern:

```python
outputs = []
for i in range(10):
      @ray.remote
      def exp(i, j):
            return i**j
      step_i_out = ray.get([exp.remote(i, j) for j in range(10)])
      outputs.append(step_i_out)
```

Better approach:

```python
@ray.remote
def exp(i, j):
      return i**j

outputs = []
for i in range(10):
      step_i_out = ray.get([exp.remote(i, j) for j in range(10)])
      outputs.append(step_i_out)
```

Antipattern: Unnecessary stateful workers

TL;DR Make actors as stateless as possible if you care about fault tolerance.

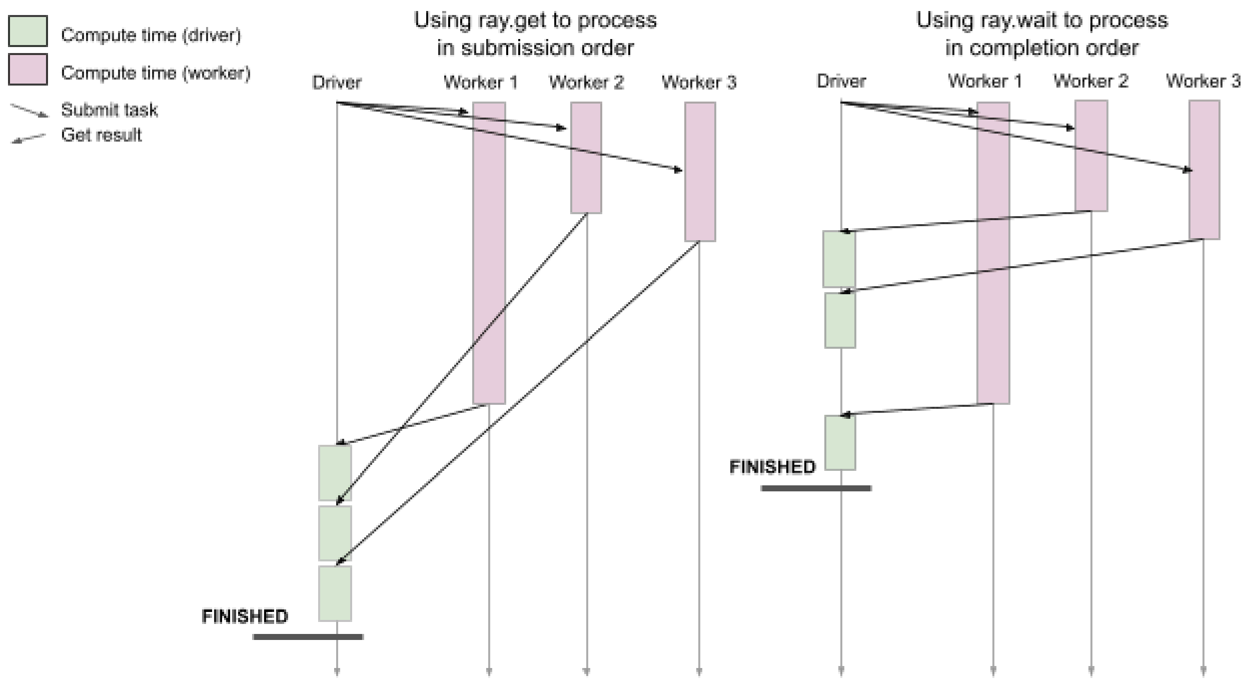
Stateful components can be a bottleneck to implement robust fault tolerant systems. It is because you need to think about how to retrieve the states when actors failed and restarted/recreated. When designing distributed applications using Ray, avoid having stateful actors when that's not absolutely necessary.

For example, when you are building an application that doesn't require high performance, you can just persist states to the database instead of actors' memory. It is because the application that is not performance-sensitive can tolerate performance degradation caused by database query (instead of accessing states directly from memory), but it will make it much easier to reason about failure scenarios.

Antipattern: Processing results in submission order using ray.get

TL;DR: avoid calling ray.get one by one in a loop if possible.

When processing in submission order, a remote function might delay processing of earlier finished remote function. When using ray.wait we can get finished tasks early, speeding up total time to completion.



A batch of tasks are submitted, and we need to process their results individually once they're done. We want to process the results as they finish, but use ray.get on the ObjectRefs in the order that they were submitted.

If each remote function takes a different amount of time to finish, we may waste time waiting for all of the slower (straggler) remote functions to finish while the other faster functions have already finished. Instead, we want to process the tasks in the order that they finish using ray.wait.

Code example

```python
import random
import time
import ray

@ray.remote
def f():
    time.sleep(random.random())

# Antipattern: process results in the order they were spawned.
```

```
refs = [f.remote(i) for i in range(100)]
for ref in refs:
    # Blocks until this ObjectRef is ready.
    result = ray.get(ref)
    # process result

# Better approach: process results in the order that they finish.
refs = [f.remote(i) for i in range(100)]
unfinished = refs
while unfinished:
    # Returns the first ObjectRef that is ready.
    finished, unfinished = ray.wait(unfinished, num_returns=1)
    result = ray.get(finished)
    # process result
```

Antipattern: Fetching too many results at once with ray.get

TL;DR: Avoid calling `ray.get` on many large objects since this will lead to object store OOM. Instead process one batch at a time.

If you have a large number of tasks that you want to run in parallel, trying to do ray.get() on all of them at once could lead to object store OOM (out of memory). Instead you should process the results a batch at a time. Once one of the batches is processed, Ray will evict those objects preventing object store from running out of memory.

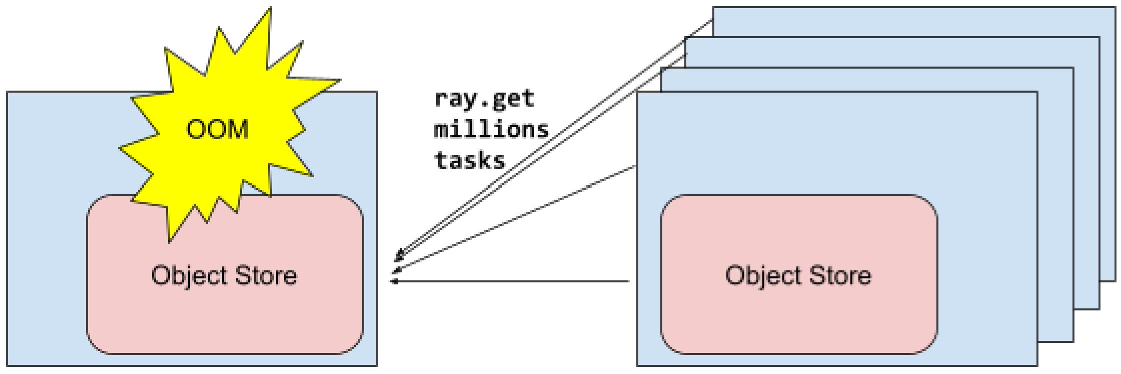
Code Example

Antipattern:

```
@ray.remote
def return_big_object():
      return np.zeros(1024*1024*10)

object_refs = [return_big_object.remote() for _ in range(1e6)]
# Calling ray.get will cause object store to OOM!
results = ray.get(object_refs)
write_to_file(results)
```

Better approach:

```
@ray.remote
def return_big_object():
      return np.zeros(1024*1024*10)

object_refs = [return_big_object.remote() for _ in range(1_000_000)]
for i in range(1_000):
      chunk = object_refs[:1_000]
      object_refs = object_refs[1_000:]
      results = ray.get(chunk)
      write_to_file(results)
```



More patterns (contributions welcome)

This is a list of patterns to add in the future; feel free to flesh them out into full pattern descriptions above, or suggest additions.

- **Using actor handles**: Handles to actors are serializable and can be passed to other tasks or actors.
- **Placement groups**: Ray placement groups allow bundles of resources spread across different nodes to be atomically reserved and scheduled into.
- **Using queues and actor pools**: Ray provides distributed queues and actor pools (task pools) in the util package.
- **AsyncIO actors as routers**: Since AsyncIO actors can handle multiple concurrent actor calls, they can be used to implement load balancing.
- **Running another framework inside Ray with actors (MPI, Torch, Horovod, etc):** Many frameworks can be conveniently launched / managed within Ray by starting "framework workers" from Ray actors.

- **Using Ray API from background threads**: The Ray API is thread safe and can be accessed from background threads in tasks and actors.
- **Passing objects by reference**: In some cases you may want to pass Ray objects by ObjectRef instead of their value. This delays the transfer of objects until they are retrieved with ray.get.
- **Returning objects by reference**: You can return a list (or other data structure) containing ObjectRefs from a task or actor method.
- **Custom resources for scheduling**: Ray allows custom resources to be declared on each node. This allows for fine-grained control over task placement (e.g., place on this particular machine that provides this custom resource).
- **Actor deadlock antipattern**: Calling ray.get(actor.remote()) to each other causes deadlock.
- **Named/Detached actors:** Creating actors that can be accessed by multiple processes by name and/or are long-living
- **Placing a worker on a specific node:** Using the ‘node:..’ resource, you can place a worker on a specific node with that ip address.