Programming in Ray: Tips for first-time users

ION STOICA / FEBRUARY 20, 2019 /

Ray (https://github.com/ray-project/ray) is a general-purpose framework for programming a cluster. Ray enables developers to easily parallelize their Python applications or build new ones, and run them at any scale, from a laptop to a large cluster. Ray provides a highly flexible, yet minimalist and easy to use API. Table 1 shows the core of this API.

In this blog, we describe several tips that can help first-time Ray users to avoid some common mistakes that can significantly hurt the performance of their programs.

API	Description	Example
ray.init()	Initialize Ray context.	

@ray.remote	Function or class decorator specifying that the function will be executed as a task or the class as an actor in a different process.	@ray.remote
		def fun(x):
		...
		
		@ray.remote
		class Actor(object):
		def method(y)
		...

.remote	Postfix to every remote function, remote class declaration, or invocation of a remote class method. Remote operations are *asynchronous*.	ret_id = fun.remote(x)
		a = Actor.remote()
		ret_id = a.method.remote(y)

ray.put()	Store object in object store, and return its ID. This ID can be used to pass object as an argument to any remote function or method call. This is a *synchronous* operation.	x_id = ray.put(x)

ray.get()	Return an object or list of objects from the object ID or list of object IDs. This is a *synchronous* (i.e., blocking) operation.	x = ray.get(x_id)
		…
		objects = ray.get(object_ids)

ray.wait()	From a list of object IDs returns (1) the list of IDs of the objects that are ready, and (2) the list of IDs of the objects that are not ready yet. By default it returns one ready object ID at a time.	ready_ids, not_ready_ids = ray.wait(object_ids)

Table 1: The core Ray API we use in this blog. The complete API is available here (https://ray.readthedocs.io/en/latest/api.html).

All the results reported in this blog were obtained on a 13-inches MacBook Pro with a 2.7 GHz Core i7 CPU and 16GB of RAM. While ray.init() automatically detects the number of cores when it runs on a single machine, to reduce the variability of the results you observe on your machine when running the code below, here we specify num_cpus = 4, i.e., a machine with 4 CPUs. Since each task requests by default one CPU, this setting allows us to execute up to four tasks in parallel. As a result, our Ray system consists of one driver executing the program, and up to four workers running remote tasks or actors,

Delay ray.get()

With Ray, the invocation of every remote operation (e.g., task, actor method) is asynchronous. This means that the operation returns *immediately* a promise/future, which is essentially an identifier (ID) of the operation's result. This is key to achieve parallelism, as it allows the driver program to launch multiple operations in parallel. To get the actual results, the programmer needs to call ray.get() on the IDs of the results. This call blocks until the results are available. As a side effect, this operation also blocks the driver program from invoking other operations, which can hurt parallelism.

Unfortunately, it is quite natural for a new Ray user to inadvertently use ray.get(). To illustrate this point, consider the following simple Python code which call the do_some_work() function four times, where each invocation takes around 1 sec:

```
import time

def do_some_work(x):
    time.sleep(1) # Replace this with work you need to do.
    return x

start = time.time()
results = [do_some_work(x) for x in range(4)]
print("duration =", time.time() - start, "\nresults = ", results)
```

The output of a program execution is below. As expected, the program takes around 4 ses:

```
duration = 4.0149290561676025
results =  [0, 1, 2, 3]
```

Now, let’s parallelize the above program with Ray. Some first-time users will do this by just making the function remote, i.e.,

```
import time
import ray

ray.init(num_cpus = 4) # Specify this system has 4 CPUs.

@ray.remote
def do_some_work(x):
    time.sleep(1) # Replace this is with work you need to do.
    return x

start = time.time()
results = [do_some_work.remote(x) for x in range(4)]
print("duration =", time.time() - start, "\nresults = ", results)
```

However, when executing the above program one gets:

```
duration = 0.0003619194030761719
results =  [ObjectID(0100000000bdf683fc3e45db42685232b19d2a61), ObjectID(01000000da69c40e1c2f43b391443ce23de46cda), Objec
tID(010000007fe0954ac2b3c0ab991538043e8f37e0), ObjectID(01000000cf47d5ecd1e26b42624454c795abe89b)]
```

When looking at this output, two things jump out. First, the program finishes immediately, i.e., in less than 1 ms. Second, instead of the expected results (i.e., [0, 1, 2, 3]) we get a bunch of identifiers. Of course, this should come as no surprise. Recall that remote operations are asynchronous and they return futures (i.e., object IDs) instead of the results themselves. This is exactly what we see here. We measure only the time it takes to invoke the tasks, not their running times, and we get the IDs pf the results corresponding to the four tasks.

To get the actual results, we need to use ray.get(), and here the first instinct is to just call ray.get() on the remote operation invocation, i.e., replace line "results = [do_some_work.remote(x) for x in range(4)]" with:

```
results = [ray.get(do_some_work.remote(x)) for x in range(4)]
```

(Note: You must run ray.init() only once. If you run it the second time in the same instance of a Python interpreter you will get an error.)

By re-running the program after this change we get:

```
duration = 4.018050909042358
results =  [0, 1, 2, 3]
```

So now the results are correct, but it still takes 4sec, so no speedup! What's going on? The observant reader will already have the answer: ray.get() is blocking, so calling it after each remote operation means that we wait for that operation to complete, which essentially means that we execute one operation at a time, hence no parallelism!

To enable parallelism, we need to call ray.get() *after* invoking all tasks. We can easily do so in our example by replacing line "results = [do_some_work.remote(x) for x in range(4)]" with:

```
results = ray.get([do_some_work.remote(x) for x in range(4)])
```

By re-running the program after this change we now get:

```
duration = 1.0064549446105957
results =  [0, 1, 2, 3]
```

So finally, success! Our Ray program now runs in just 1 sec which means that all invocations of do_some_work() are running in parallel.

In summary, always keep in mind that ray.get() is a blocking operation, and thus if called eagerly it can hurt the parallelism. Instead, you should try to write your program such that ray.get() is called as late as possible.

Tip 1: *Delay calling ray.get() as much as possible.*

Avoid tiny tasks

When a first-time developer wants to parallelize their code with Ray, the natural instinct is to make every function or class remote. Unfortunately, this can lead to undesirable consequences; if the tasks are very small, the Ray program can take *longer* than the equivalent Python program.

Let's consider again the above examples, but this time we make the tasks much sorter (i.e, each takes just 0.1ms), and dramatically increase the number of task invocations to 100,000.

```
import time

def tiny_work(x):
    time.sleep(0.0001) # Replace this with work you need to do.
    return x

start = time.time()
results = [tiny_work(x) for x in range(100000)]
print("duration =", time.time() - start)
```

By running this program we get:

```
duration = 13.36544418334961
```

This result should be expected since the lower bound of executing 100,000 tasks that take 0.1ms each is 10s, to which we need to add other overheads such as function calls, etc.

Let’s now parallelize this code using Ray, by making every invocation of do_some_work() remote:

```
import time
import ray

ray.init(num_cpus = 4)

@ray.remote
def tiny_work(x):
    time.sleep(0.0001) # Replace this is with work you need to do.
    return x

start = time.time()
result_ids = [tiny_work.remote(x) for x in range(100000)]
results = ray.get(result_ids)
print("duration =", time.time() - start)
```

The result of running this code is:

```
duration = 27.46447515487671
```

Surprisingly, not only Ray didn’t improve the execution time, but the Ray program is actually slower than the sequential program! What’s going on? Well, the issue here is that every task invocation has a non-trivial overhead (e.g., scheduling, inter-process communication, updating the system state) and this overhead dominates the actual time it takes to execute the task.

One way to speed up this program is to make the remote tasks larger in order to amortize the invocation overhead. Here is one possible solution where we aggregate 1000 tiny_work() function calls in a single bigger remote function, mega_work() :

```
import time
import ray

ray.init(num_cpus = 4)

def tiny_work(x):
    time.sleep(0.0001) # replace this is with work you need to do
    return x

@ray.remote
def mega_work(start, end):
    return [tiny_work(x) for x in range(start, end)]

start = time.time()
result_ids = []
[result_ids.append(mega_work.remote(x*1000, (x+1)*1000)) for x in range(100)]
results = ray.get(result_ids)
print("duration =", time.time() - start)
```

Now, if we run the above program we get:

```
duration = 3.2539820671081543
```

This is approximately one fourth of the sequential execution, in line with our expectations (recall, we can run four tasks in parallel). Of course, the natural question is how large is large enough for a task to amortize the remote invocation overhead. One way to find this is to run the following simple program to estimate the per-task invocation overhead:

```
@ray.remote
def no_work(x):
    return x

start = time.time()
num_calls = 1000
[ray.get(no_work.remote(x)) for x in range(num_calls)]
print("per task overhead (ms) =", (time.time() - start)*1000/num_calls)
```

Running the above program shows:

```
per task overhead (ms) = 0.4739549160003662
```

In other words, it takes almost half a millisecond to execute an empty task. This suggests that we will need to make sure a task takes at least a few milliseconds to amortize the invocation overhead. One caveat is that the per-task overhead will vary from machine to machine, and between tasks that run on the same machine versus remotely. This being said, making sure that tasks take at least a few milliseconds is a good rule of thumb when developing Ray programs.

Tip 2: *For exploiting Ray's parallelism, remote tasks should take at least several milliseconds.*

Avoid passing same object repeatedly to remote tasks

When we pass a large object as an argument to a remote function, Ray calls ray.put() under the hood to store that object in the local object store. This can significantly improve the performance of a remote task invocation when the remote task is executed locally, as all local tasks share the object store. However, there are cases when automatically calling ray.put() on a task invocation leads to performance issues. On example is passing the same large object as an argument *repeatedly*, as illustrated by the program below:

```
import time
import numpy as np
import ray

ray.init(num_cpus = 4)

@ray.remote
def no_work(a):
    return

start = time.time()
a = np.zeros((10000, 2000))
result_ids = [no_work.remote(a) for x in range(10)]
results = ray.get(result_ids)
print("duration =", time.time() - start)
```

This program outputs:

```
duration = 1.0699057579040527
```

(Note: If this program takes much more than 1 sec, it is probably because your machine does not have enough memory. If this is the case, stop this program and run the next one.)

This running time is quite large for a program that calls just 10 remote tasks that do nothing. The reason for this unexpected high running time is that each time we invoke no_work(a), Ray calls ray.put(a) which results in copying array a to the object store. Since array a has 20 million entries copying it takes a non-trivial time.To avoid copying array a every time no_work() is invoked, one simple solution is to explicitly call ray.put(a), and then pass a's ID to no_work(), as illustrated below:

```
import time
import numpy as np
import ray

ray.init(num_cpus = 4)

@ray.remote
def no_work(a):
    return

start = time.time()
a_id = ray.put(np.zeros((10000, 2000)))
result_ids = [no_work.remote(a_id) for x in range(10)]
results = ray.get(result_ids)
print("duration =", time.time() - start)
```

Running this program takes only:

```
duration = 0.12425804138183594
```

This is 8 times faster than the original program which is to be expected since the main overhead of invoking no_work(a) was copying the array a to the object store, which now happens only once.

Arguably a more important advantage of avoiding multiple copies of the same object to the object store is that it precludes the object store filling up prematurely and incur the cost of object eviction.

Tip 3: *When passing the same object repeatedly as an argument to a remote operation, use ray.put() to store it once in the object store and then pass its ID.*

Pipeline data processing

If we use ray.get() on the results of multiple tasks we will have to wait until the *last* one of these tasks finishes. This can be an issue if tasks take widely different amounts of time. To illustrate this issue, consider the following example where we run four do_some_work() tasks in parallel, with each task taking a time uniformly distributed between 0 and 4 sec. Next, assume the results of these tasks are processed by process_results(), which takes 1 sec per result. The expected running time is then (1) the time it takes to execute the slowest of the do_some_work() tasks, plus (2) 4 sec which is the time it takes to execute process_results().

```
import time
import random
import ray

ray.init(num_cpus = 4)

@ray.remote
def do_some_work(x):
    time.sleep(random.uniform(0, 4)) # Replace this with work you need to do.
    return x

def process_results(results):
    sum = 0
    for x in results:
        time.sleep(1) # Replace this with some processing code.
        sum += x
    return sum

start = time.time()
data_list = ray.get([do_some_work.remote(x) for x in range(4)])
sum = process_results(data_list)
print("duration =", time.time() - start, "\nresult = ", sum)
```

The output of the program shows that it takes close to 8 sec to run:

```
duration = 7.82636022567749
result = 6
```

Waiting for the last task to finish when the others tasks might have finished much earlier unnecessarily increases the program running time. A better solution would be to process the data *as soon it becomes available*.Fortunately, Ray allows you to do exactly this by calling ray.wait() on a list of object IDs. Without specifying any other parameters, this function returns as soon as an object in its argument list is ready. This call has two returns: (1) the ID of the ready object, and (2) the list containing the IDs of the objects not ready yet. The modified program is below. Note that one change we need to do is to replace process_results() with process_incremental() that processes one result at a time.

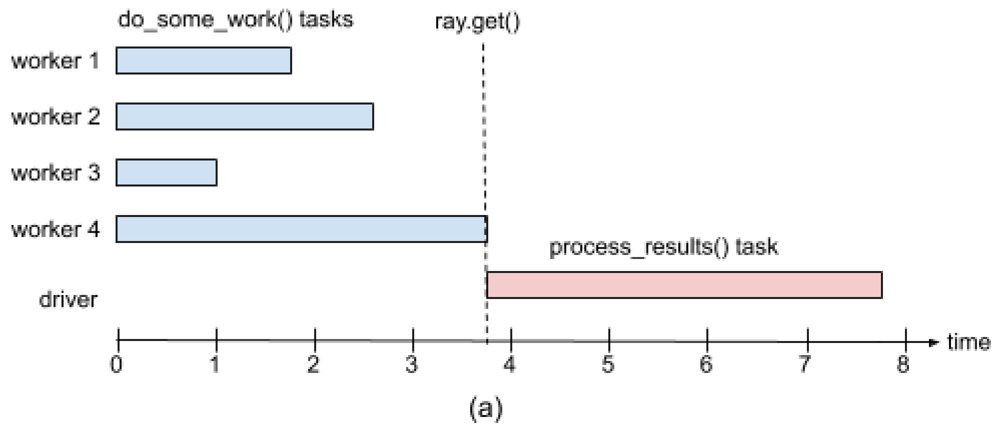
```
import time
import random
import ray

ray.init(num_cpus = 4)

@ray.remote
def do_some_work(x):
    time.sleep(random.uniform(0, 4)) # Replace this with work you need to do.
    return x

def process_incremental(sum, result):
    time.sleep(1) # Replace this with some processing code.
    return sum + result

start = time.time()
result_ids = [do_some_work.remote(x) for x in range(4)]
sum = 0
while len(result_ids):
    done_id, result_ids = ray.wait(result_ids)
    sum = process_incremental(sum, ray.get(done_id[0]))
print("duration =", time.time() - start, "\nresult = ", sum)
```

This program now takes just a bit over 4.8 sec a significant improvement:

```
duration = 4.852453231811523
result = 6
```

(Note: Different runs might take different times, but "duration" should still be significantly less than 8 sec.)

To aid the intuition, Figure 1 shows the execution timeline in both cases: when using ray.get() to wait for all results to become available before processing them, and using ray.wait() to start processing the results as soon as they become available.



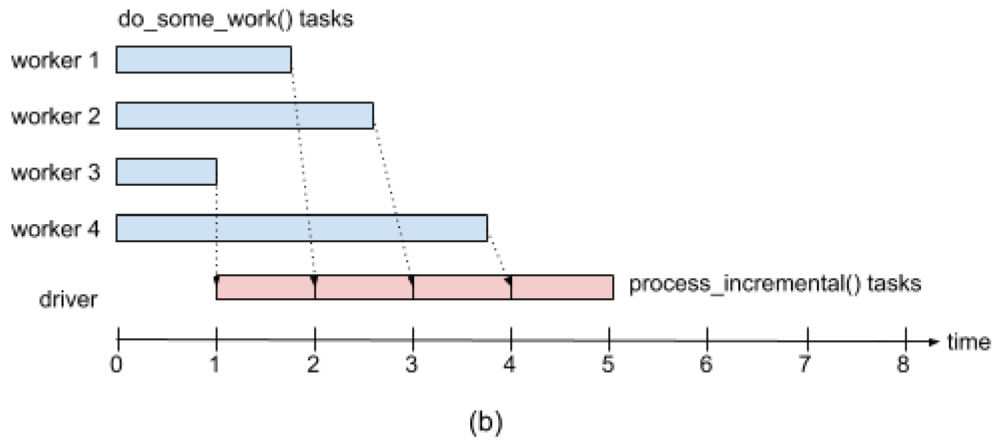


Figure 1: (a) Execution timeline when using ray.get() to wait for all results from do_some_work() tasks before calling process_results(). (b) Execution timeline when using ray.wait() to process results as soon as they become available.

Tip 4: *Use ray.wait() to process results as soon as they become available.*

Other resources

The Ray tutorial (https://github.com/ray-project/tutorial) is a great resource to learn more about programming in Ray.

Post Views: 12,246

HOME (HTTPS://RISE.CS.BERKELEY.EDU) PEOPLE (HTTPS://RISE.CS.BERKELEY.EDU/PEOPLE/)
PROJECTS (HTTPS://RISE.CS.BERKELEY.EDU/PROJECTS/) PUBLICATIONS (HTTPS://RISE.CS.BERKELEY.EDU/PUBLICATIONS/)
SPONSORS (HTTPS://RISE.CS.BERKELEY.EDU/SPONSORS/) ACADEMICS (HTTPS://RISE.CS.BERKELEY.EDU/ACADEMICS/)
NEWS (HTTPS://RISE.CS.BERKELEY.EDU/NEWS/) EVENTS (HTTPS://RISE.CS.BERKELEY.EDU/EVENTS/)
RISE CAMP (HTTP://RISECAMP.BERKELEY.EDU) BLOGS (HTTPS://RISE.CS.BERKELEY.EDU/BLOG/)
JENKINS (HTTPS://RISE.CS.BERKELEY.EDU/JENKINS/)