# ... continued

Continues the discussion on pools and describes the map API.

In the previous section we discussed the simplest API offered by the `Pool` class. However, it offers a much more powerful `map()` API. The name may ring a bell with readers who have background in big data or have worked with the famous **map-reduce** programming paradigm.

## map reduce

Very briefly, the idea behind map-reduce is dividing a very large problem into smaller chunks that can be worked upon independently from each other. The chunks are distributed among different processing units and as the results for each chunk become ready, they are sorted and passed through a user-provided *reducer* function which defines the logic for combining all the results. Many real-world problems can be modeled on the lines of the map-reduce programming paradigm and be solved much faster.

## map ( )

The `Pool` class's map API breaks up the problem size into smaller chunks and distributes them off to various processes but it doesn't get as sophisticated as the original map-reduce model, rather it simply aggregates all the results in the same order as it received the input and presents them to a user-defined callback. Let's continue with our example of computing squares and this time we'll compute the square of several numbers instead of just one. The program to do so is given

several numbers instead of just one. The program to do so is given below:

```
from multiprocessing import Pool
import os

def square(arg):
    return arg * arg


def init(main_id):
    print("pool process with id {0} received a task from mai
n process with id {1}".format(os.getpid(), main_id),
          flush=True)


def on_success(new_result):
    print("\nReceived result as: " + str(new_result))


def on_error(err):
    print("Error : " + str(err))


if __name__ == '__main__':
    pool = Pool(processes=5, initializer=init,
                initargs=(os.getpid(),),
                maxtasksperchild=50)

    res = pool.map_async(square,
                         (1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
                         chunksize=2,
                         callback=on_success,
                         error_callback=on_error)


    pool.close()
    pool.join()
```

```
from multiprocessing import Pool
import os

def square(arg):
    return arg * arg


def init(main_id):
    print("pool process with id {0} received a task from main process with id {1}".
          flush=True)


def on_success(new_result):
    print("\nReceived result as: " + str(new_result))


def on_error(err):
    print("Error : " + str(err))


if __name__ == '__main__':
    pool = Pool(processes=5, initializer=init,
                initargs=(os.getpid(),),
                maxtasksperchild=50)

    res = pool.map_async(square,
                         (1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
                         chunksize=2,
                         callback=on_success,
                         error_callback=on_error)


    pool.close()
    pool.join()
```

We have used the **`map_async()`** but a corresponding blocking and much simpler **`map()`** also exists. The above program with the blocking map call appears below:

```
from multiprocessing import Pool
import os

def square(arg):
    return arg * arg


def init(main_id):
    print("pool process with id {0} received a task from main process with id {1}".
          flush=True)


if __name__ == '__main__':
    pool = Pool(processes=5, initializer=init,
                initargs=(os.getpid(),),
                maxtasksperchild=50)

    res = pool.map(square,
                        (1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
                        chunksize=2)

    print(res)

    pool.close()
    pool.join()
```

The most important argument to the map calls is the **chunksize**, which we have defined as 2. This means that the input consisting of 10 integers is divided into chunks of two integers and each process gets 2 integers to work on. We have five processes in the pool so each process gets exactly one chunk to work on. If we reduce the processes to less than 5 then some processes may get more chunks to work on.

There are other variants of map, for instance **`imap()`** and **`imap_unordered()`** which are blocking calls and return iterators for results without the need for specifying callbacks. The unordered version returns the results which may not match the order of the inputs. The

square program using the unordered version is given below:

## Using imap_unordered

```
from multiprocessing import Pool
import os


def square(arg):
    return arg * arg


def init(main_id):
    print("pool process with id {0} received a task from mai
n process with id {1}".format(os.getpid(), main_id),
          flush=True)


if __name__ == '__main__':
    pool = Pool(processes=5, initializer=init,
                initargs=(os.getpid(),),
                maxtasksperchild=50)

    res = pool.imap_unordered(square, (1, 2, 3, 4, 5, 6, 7, 8,
 9, 10),
                      chunksize=2)

    for sq in res:
        print(sq)
```

```
from multiprocessing import Pool
import os


def square(arg):
    return arg * arg


def init(main_id):
    print("pool process with id {0} received a task from main process with id {1}".
          flush=True)


if __name__ == '__main__':
    pool = Pool(processes=5, initializer=init,
                initargs=(os.getpid(),),
                maxtasksperchild=50)

    res = pool.imap_unordered(square, (1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
                      chunksize=2)

    for sq in res:
        print(sq)
```

# starmap ( ) and starmap_aync ( )

The starmap versions of the map API allow us to specify more than one argument for our task method. So far in our example the **`square(x)`** was taking in a single argument. These APIs allow us to pass in more than one argument. The input to the starmap API is specified as an iterable of iterables e.g. a nested list. The nested list is then unpacked as arguments to be passed on to the task method. For example, say we now want the square method to accept an integer and a string and the return value is now the concatenation of the string and the squared value. With this change the starmap API can be invoked like so:

```
    res = pool.starmap_async(square,
                             ((1, 'chunk1'),
                              (3, 'chunk2'),
                              (5, 'chunk3'),
                              (7, 'chunk4'),
                              (9, 'chunk5')),
                             chunksize=2,
                             callback=on_success,
                             error_callback=on_error)
```

Note that the data being fed to the starmap api is a nested list and each list forms the input arguments to the **`square()`** method. The complete program appears below:

```
from multiprocessing import Pool
import time
import os


def square(arg1, arg2):
    return arg2 + '-' + str(arg1 * arg1)


def init(main_id):
    print("pool process with id {0} received a task from main process with id {1}".
          flush=True)


def on_success(new_result):
    for x in new_result:
        print("\n" + x)


def on_error(err):
    print("Error : " + str(err))


if __name__ == '__main__':
    pool = Pool(processes=5, initializer=init,
                initargs=(os.getpid(),),
                maxtasksperchild=50)

    res = pool.starmap_async(square,
                             ((1, 'chunk1'),
                              (3, 'chunk2'),
                              (5, 'chunk3'),
                              (7, 'chunk4'),
                              (9, 'chunk5')),
                             chunksize=2,
                             callback=on_success,
                             error_callback=on_error)

    pool.close()
    pool.join()
```

# Failures

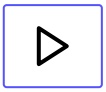
The traditional map-reduce is fault tolerant. If a failure occurs while processing one of the chunks, the processing is retried. However, the `map` APIs don't provide automatic handling of failures and rely on the user to handle failures via the **errorcallback**. Two possible approaches that can be taken for error handling are:

- Implementing retry logic in the task execution method.
- Using a queue to collect failed tasks and resubmit them.

The first one is obvious and we present the second one below. The program randomly fails a task with a 1 in 5 chance or twenty percent probability. The failed items are enqueued to a `Multiprocessing.Queue` and the main process resubmits the failed tasks to the pool till all tasks are completed. The order in the final results isn't preserved though. The example uses a blocking version of the map API but an asynchronous one can be made to work with proper synchronization.

```
from multiprocessing import Pool, Queue
import os
import random

random.seed()


def square(arg):
    i = random.randrange(0, 5)
    tmp = arg
    if i % 5 is 0:
        print("injecting failure", flush=True)
        tmp = None
    try:
        res = None
        res = tmp * tmp
    except:
        square.q.put(arg)

    return res


def init(main_id, q):
    print("pool process with id {0} received a task from main process with id {1}".
          flush=True)
    square.q = q


def queue_to_list(q):
    i = 0
    lst = list()
    while not q.empty():
        lst.append(q.get())
        i += 1

    return lst, i


if __name__ == '__main__':
    q = Queue()
    failures = 0
    done = False
    lst = list([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

    pool = Pool(processes=5, initializer=init,
                initargs=(os.getpid(), q),
                maxtasksperchild=50)

    final_results = list()
    while not done:
        res = pool.map(square, lst, chunksize=2)
```

```
        # print(res)
        final_results.append(res)


        if not q.empty():
            lst, i = queue_to_list(q)
            failures += i
        else:
            done = True

    print("failures: " + str(failures))
    print("final resultfinal_results", final_results)
```



← Back

Pool

Next →

Manager

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/-continued__multiprocessing__python-concurrency-for-senior-engineering-interviews)