Implementing MapReduce

The `Pool` class can be used to create a simple single-server MapReduce implementation. Although it does not give the full benefits of distributed processing, it does illustrate how easy it is to break some problems down into distributable units of work.

In a MapReduce-based system, input data is broken down into chunks for processing by different worker instances. Each chunk of input data is *mapped* to an intermediate state using a simple transformation. The intermediate data is then collected together and partitioned based on a key value so that all of the related values are together. Finally, the partitioned data is *reduced* to a result set.

```
# multiprocessing_mapreduce.py

import collections
import itertools
import multiprocessing


class SimpleMapReduce:

    def __init__(self, map_func, reduce_func, num_workers=None):
        """
        map_func

          Function to map inputs to intermediate data. Takes as
          argument one input value and returns a tuple with the
          key and a value to be reduced.

        reduce_func

          Function to reduce partitioned version of intermediate
          data to final output. Takes as argument a key as
          produced by map_func and a sequence of the values
          associated with that key.

        num_workers

          The number of workers to create in the pool. Defaults
          to the number of CPUs available on the current host.
        """
        self.map_func = map_func
        self.reduce_func = reduce_func
        self.pool = multiprocessing.Pool(num_workers)

    def partition(self, mapped_values):
        """Organize the mapped values by their key.
        Returns an unsorted sequence of tuples with a key
        and a sequence of values.
        """
        partitioned_data = collections.defaultdict(list)
        for key, value in mapped_values:
            partitioned_data[key].append(value)
        return partitioned_data.items()

    def __call__(self, inputs, chunksize=1):
        """Process the inputs through the map and reduce functions
        given.

        inputs
          An iterable containing the input data to be processed.

        chunksize=1
          The portion of the input data to hand to each worker.
```

```
          This can be used to tune performance during the mapping
          phase.
        """
        map_responses = self.pool.map(
            self.map_func,
            inputs,
            chunksize=chunksize,
        )
        partitioned_data = self.partition(
            itertools.chain(*map_responses)
        )
        reduced_values = self.pool.map(
            self.reduce_func,
            partitioned_data,
        )
        return reduced_values
```

The following example script uses SimpleMapReduce to counts the “words” in the reStructuredText source for this article, ignoring some of the markup.

```
# multiprocessing_wordcount.py

import multiprocessing
import string

from multiprocessing_mapreduce import SimpleMapReduce


def file_to_words(filename):
    """Read a file and return a sequence of
    (word, occurences) values.
    """
    STOP_WORDS = set([
        'a', 'an', 'and', 'are', 'as', 'be', 'by', 'for', 'if',
        'in', 'is', 'it', 'of', 'or', 'py', 'rst', 'that', 'the',
        'to', 'with',
    ])
    TR = str.maketrans({
        p: ' '
        for p in string.punctuation
    })

    print('{} reading {}'.format(
        multiprocessing.current_process().name, filename))
    output = []

    with open(filename, 'rt') as f:
        for line in f:
            # Skip comment lines.
            if line.lstrip().startswith('..'):
                continue
            line = line.translate(TR)  # Strip punctuation
            for word in line.split():
                word = word.lower()
                if word.isalpha() and word not in STOP_WORDS:
                    output.append((word, 1))
    return output


def count_words(item):
    """Convert the partitioned data for a word to a
    tuple containing the word and the number of occurences.
    """
    word, occurences = item
    return (word, sum(occurences))


if __name__ == '__main__':
    import operator
    import glob
```

```
    input_files = glob.glob('*.rst')

    mapper = SimpleMapReduce(file_to_words, count_words)
    word_counts = mapper(input_files)
    word_counts.sort(key=operator.itemgetter(1))
    word_counts.reverse()

    print('\nTOP 20 WORDS BY FREQUENCY\n')
    top20 = word_counts[:20]
    longest = max(len(word) for word, count in top20)
    for word, count in top20:
        print('{word:<{len}}: {count:5}'.format(
            len=longest + 1,
            word=word,
            count=count)
        )
```

The `file_to_words()` function converts each input file to a sequence of tuples containing the word and the number 1 (representing a single occurrence). The data is divided up by `partition()` using the word as the key, so the resulting structure consists of a key and a sequence of 1 values representing each occurrence of the word. The partitioned data is converted to a set of tuples containing a word and the count for that word by `count_words()` during the reduction phase.

```
$ python3 -u multiprocessing_wordcount.py

ForkPoolWorker-1 reading basics.rst
ForkPoolWorker-2 reading communication.rst
ForkPoolWorker-3 reading index.rst
ForkPoolWorker-4 reading mapreduce.rst

TOP 20 WORDS BY FREQUENCY

process         :    83
running         :    45
multiprocessing :    44
worker          :    40
starting        :    37
now             :    35
after           :    34
processes       :    31
start           :    29
header          :    27
pymotw          :    27
caption         :    27
end             :    27
daemon          :    22
can             :    22
exiting         :    21
forkpoolworker  :    21
consumer        :    20
main            :    18
event           :    16
```

Passing Messages to Processes

asyncio — Asynchronous I/O, event loop, and concurrency tools

This page was last updated 2016-12-29.

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?