06-ParallelComputation

August 10, 2020

1 Parallel Computation

1.1 Parallel computers

- Multiprocessor/multicore: several processors work on data stored in shared memory
- Cluster: several processor/memory units work together by exchanging data over a network
- Co-processor: a general-purpose processor delegates specific tasks to a special-purpose processor (GPU)

1.2 Parallel Programming

- Decomposition of the complete task into independent subtasks and the data flow between them.
- Distribution of the subtasks over the processors minimizing the total execution time.
- For clusters: distribution of the data over the nodes minimizing the communication time.
- For multiprocessors: optimization of the memory access patterns minimizing waiting times.
- Synchronization of the individual processes.

1.3 MapReduce

[1]:
```python
from time import sleep
def f(x):
    sleep(1)
    return x*x
L = list(range(8))
L
```

[1]:
```
[0, 1, 2, 3, 4, 5, 6, 7]
```

[2]:
```python
%time sum(f(x) for x in L)
```

```
CPU times: user 3.46 ms, sys: 446 µs, total: 3.91 ms
Wall time: 8.01 s
```

[2]:
```
140
```

[3]:
```
%time sum(map(f,L))
```

```
CPU times: user 3.45 ms, sys: 436 µs, total: 3.89 ms
Wall time: 8.01 s
```

[3]:
```
140
```

1.4 Multiprocessing

`multiprocessing` is a package that supports spawning processes.

We can use it to display how many concurrent processes you can launch on your computer.

[4]:
```
from multiprocessing import cpu_count

cpu_count()
```

[4]:
```
2
```

1.5 Futures

The `concurrent.futures` module provides a high-level interface for asynchronously executing callables.

The asynchronous execution can be performed with: - **threads**, using ThreadPoolExecutor, - separate **processes**, using ProcessPoolExecutor. Both implement the same interface, which is defined by the abstract Executor class.

`concurrent.futures` can't launch **processes** on windows. Windows users must install loky.

[5]:
```
%%file pmap.py
from concurrent.futures import ProcessPoolExecutor
from time import sleep, time

def f(x):
    sleep(1)
    return x*x

L = list(range(8))

if __name__ == '__main__':

    begin = time()
    with ProcessPoolExecutor() as pool:

        result = sum(pool.map(f, L))
    end = time()
```

```python
    print(f"result = {result} and time = {end-begin}")
```

```
Overwriting pmap.py
```

[6]:
```python
import sys
!{sys.executable} pmap.py
```

```
result = 140 and time = 4.014407396316528
```

- `ProcessPoolExecutor` launches one slave process per physical core on the computer.
- `pool.map` divides the input list into chunks and puts the tasks (function + chunk) on a queue.
- Each slave process takes a task (function + a chunk of data), runs map(function, chunk), and puts the result on a result list.
- `pool.map` on the master process waits until all tasks are handled and returns the concatenation of the result lists.

[7]:
```python
%%time
from concurrent.futures import ThreadPoolExecutor

with ThreadPoolExecutor() as pool:

    results = sum(pool.map(f, L))

print(results)
```

```
140
CPU times: user 6.72 ms, sys: 315 µs, total: 7.04 ms
Wall time: 2.01 s
```

1.6 Thread and Process: Differences

- A **process** is an instance of a running program.
- **Process** may contain one or more **threads**, but a **thread** cannot contain a **process**.
- **Process** has a self-contained execution environment. It has its own memory space.
- Application running on your computer may be a set of cooperating **processes**.
- **Process** don't share its memory, communication between **processes** implies data serialization.
- A **thread** is made of and exist within a **process**; every **process** has at least one **thread**.
- Multiple **threads** in a **process** share resources, which helps in efficient communication between **threads**.
- **Threads** can be concurrent on a multi-core system, with every core executing the separate **threads** simultaneously.

1.7 The Global Interpreter Lock (GIL)

- The Python interpreter is not thread safe.
- A few critical internal data structures may only be accessed by one thread at a time. Access to them is protected by the GIL.
- Attempts at removing the GIL from Python have failed until now. The main difficulty is maintaining the C API for extension modules.
- Multiprocessing avoids the GIL by having separate processes which each have an independent copy of the interpreter data structures.
- The price to pay: serialization of tasks, arguments, and results.

1.8 Weighted mean and Variance

1.8.1 Exercise 6.1

Use `ThreadPoolExecutor` to parallelized functions written in notebook 05

[8]:
```
X = [5, 1, 2, 3, 1, 2, 5, 4]
P = [0.05, 0.05, 0.15, 0.05, 0.15, 0.2, 0.1, 0.25]
```

[9]:
```
from operator import add, mul
from functools import reduce
from concurrent.futures import ThreadPoolExecutor as pool

def weighted_mean( X, P):

    with pool() as p:
        w1 = p.map(mul, X, P)

    return reduce(add,w1)

weighted_mean(X,P)
```

[9]: 2.8

[10]:
```
def variance(X, P):
    mu = weighted_mean(X,P)
    with pool() as p:
        w2 = p.map(lambda x,p:p*x*x, X, P)
    return reduce(add,w2) - mu**2

variance(X, P)
```

[10]: 1.9600000000000017

[11]:
```
import numpy as np
x = np.array(X)
```

```
p = np.array(P)
np.average( x, weights=p)
```

[11]: 2.8

[12]:
```
var =np.sum(p*x**2) - np.average( x, weights=p)**2
var
```

[12]: 1.9600000000000017

1.9 Wordcount

[13]:
```
from glob import glob
from collections import defaultdict
from operator import itemgetter
from itertools import chain
from concurrent.futures import ThreadPoolExecutor


def mapper(filename):
    " split text to list of key/value pairs (word,1)"
    with open(filename) as f:
        data = f.read()

    data = data.strip().replace(".","").lower().split()

    return sorted([(w,1) for w in data])


def partitioner(mapped_values):
    """ get lists from mapper and create a dict with
    (word,[1,1,1])"""

    res = defaultdict(list)
    for w, c in mapped_values:
        res[w].append(c)

    return res.items()


def reducer( item ):
    """ Compute words occurences from dict computed
    by partioner
    """
    w, v = item
    return (w,len(v))
```

1.10 Parallel map

- Let's improve the `mapper` function by print out inside the function the current process name.

Example

[14]:
```
import multiprocessing as mp
from concurrent.futures import ThreadPoolExecutor
#from loky import ProcessPoolExecutor
def process_name(n):
    " prints out the current process name "
    print(mp.current_process().name)

with ProcessPoolExecutor() as e:
    _ = e.map(process_name, range(mp.cpu_count()))
```

```
      ␣
↪---------------------------------------------------------------------------

    NameError                                 Traceback (most recent call␣
↪last)

    <ipython-input-14-2096cf4534b5> in <module>
      6     print(mp.current_process().name)
      7
----> 8 with ProcessPoolExecutor() as e:
      9     _ = e.map(process_name, range(mp.cpu_count()))


    NameError: name 'ProcessPoolExecutor' is not defined
```

1.10.1 Exercise 6.2

- Modify the mapper function by adding this print.

[15]:
```
def mapper(filename):
    " split text to list of key/value pairs (word,1)"

    print(f"{mp.current_process().name} : {filename}")
    with open(filename) as f:
        data = f.read()

    data = data.strip().replace(".","").lower().split()

    return sorted([(w,1) for w in data])
```

1.11 Parallel reduce

- For parallel reduce operation, data must be aligned in a container. We already created a `partitioner` function that returns this container.

1.11.1 Exercise 6.3

Write a parallel program that uses the three functions above using `ProcessPoolExecutor`. It reads all the "sample*.txt" files. Map and reduce steps are parallel.

[16]:
```
from concurrent.futures import ThreadPoolExecutor

def wordcount(files):

    with ThreadPoolExecutor() as e:

        mapped_values = e.map(mapper, files)
        partioned_values = partitioner(chain(*mapped_values))
        occurences = e.map(reducer, partioned_values)

    return sorted(occurences,
                  key=itemgetter(1),
                  reverse=True)

files = glob("sample*.txt")
wordcount(files)
```

[16]:
```
[]
```

1.12 Increase volume of data

Due to the proxy, code above is not runnable on workstations

1.12.1 Getting the data

- The Latin Library contains a huge collection of freely accessible Latin texts. We get links on the Latin Library's homepage ignoring some links that are not associated with a particular author.

[17]:
```
from bs4 import BeautifulSoup  # web scraping library
from urllib.request import *

base_url = "http://www.thelatinlibrary.com/"
home_content = urlopen(base_url)

soup = BeautifulSoup(home_content, "lxml")
```

```
author_page_links = soup.find_all("a")
author_pages = [ap["href"] for i, ap in enumerate(author_page_links) if i < 49]
```

1.12.2 Generate html links

- Create a list of all links pointing to Latin texts. The Latin Library uses a special format which makes it easy to find the corresponding links: All of these links contain the name of the text author.

[18]:
```
ap_content = list()
for ap in author_pages:
    ap_content.append(urlopen(base_url + ap))

book_links = list()
for path, content in zip(author_pages, ap_content):
    author_name = path.split(".")[0]
    ap_soup = BeautifulSoup(content, "lxml")
    book_links += ([link for link in ap_soup.find_all("a", {"href": True}) if␣
 ↪author_name in link["href"]])
```

1.12.3 Download webpages content

[19]:
```
from urllib.error import HTTPError

num_pages = 100

for i, bl in enumerate(book_links[:num_pages]):
    print("Getting content " + str(i + 1) + " of " + str(num_pages), end="\r",␣
 ↪flush=True)
    try:
        content = urlopen(base_url + bl["href"]).read()
        with open(f"book-{i:03d}.dat","wb") as f:
            f.write(content)
    except HTTPError as err:
        print("Unable to retrieve " + bl["href"] + ".")
        continue
```

```
Getting content 100 of 100
```

1.12.4 Extract data files

- I already put the content of pages in files named book-*.txt
- You can extract data from the archive by running the cell below

```
import os  # library to get directory and file paths
import tarfile # this module makes possible to read and write tar archives

def extract_data():
    datadir = os.path.join('data','latinbooks')
    if not os.path.exists(datadir):
       print("Extracting data...")
       tar_path = os.path.join('data', 'latinbooks.tgz')
       with tarfile.open(tar_path, mode='r:gz') as books:
          books.extractall('data')

extract_data() # this function call will extract text files in data/latinbooks
```

1.12.5 Read data files

[20]:
```
from glob import glob
files = glob('book*.dat')
texts = list()
for file in files:
    with open(file,'rb') as f:
        text = f.read()
    texts.append(text)
```

1.12.6 Extract the text from html and split the text at periods to convert it into sentences.

[21]:
```
%%time
from bs4 import BeautifulSoup

sentences = list()

for i, text in enumerate(texts):
    print("Document " + str(i + 1) + " of " + str(len(texts)), end="\r",␣
 ↪flush=True)
    textSoup = BeautifulSoup(text, "lxml")
    paragraphs = textSoup.find_all("p", attrs={"class":None})
    prepared = ("".join([p.text.strip().lower() for p in paragraphs[1:-1]]))
    for t in prepared.split("."):
        part = "".join([c for c in t if c.isalpha() or c.isspace()])
        sentences.append(part.strip())

# print first and last sentence to check the results
print(sentences[0])
print(sentences[-1])
```

```
sed nimirum nihil fortuna rennuente licet homini natu dexterum provenire nec
consilio prudenti vel remedio sagaci divinae providentiae fatalis dispositio
subuerti vel reformari potest

CPU times: user 1.58 s, sys: 42.4 ms, total: 1.62 s
Wall time: 1.56 s
```

1.12.7 Exercise 6.4

Parallelize this last process using `concurrent.futures`.

[22]:
```python
%%time
from bs4 import BeautifulSoup
from concurrent.futures import ThreadPoolExecutor as pool

def sentence_mapper(text):
    sentences = list()
    textSoup = BeautifulSoup(text, "lxml")
    paragraphs = textSoup.find_all("p", attrs={"class":None})
    prepared = ("".join([p.text.strip().lower() for p in paragraphs[1:-1]]))
    for t in prepared.split("."):
        part = "".join([c for c in t if c.isalpha() or c.isspace()])
        sentences.append(part.strip())
    return sentences

# parallel map
with pool(4) as p:

    mapped_sentences = p.map(sentence_mapper, texts)

# reduce
sentences = reduce(add, mapped_sentences )

# print first and last sentence to check the results
print(sentences[0])
print(sentences[-1])
```

```
sed nimirum nihil fortuna rennuente licet homini natu dexterum provenire nec
consilio prudenti vel remedio sagaci divinae providentiae fatalis dispositio
subuerti vel reformari potest

CPU times: user 2.68 s, sys: 971 ms, total: 3.65 s
Wall time: 2.74 s
```

1.13 References

- Using Conditional Random Fields and Python for Latin word segmentation