Multiprocessing in Python

Making Embarassingly Parallel Tasks Embarassingly Fast

Vítek Novotný
witiko@mail.muni.cz

Faculty of Informatics, Masaryk University

May 12, 2021

Introduction

In Python, you can achieve several kinds of parallelism:

Single Instruction, Single Data (SISD) Plain Python. Ok for logic, slow for computation.

Single Instruction, Multiple Data (SIMD) Vector instructions for CPUs. Use `numpy`.

Single Program, Multiple Data (SPMD) Many processes of one program work in parallel.

- Shared Memory One machine. Use `multiprocessing`.
- Distributed Memory Many machines. Use GNU Parallel.

In this tutorial, I will show you how you can transform SISD code into SIMD and SPMD in your second term project with examples. This will *significantly speed up your code.* ;-)

SISD to SIMD I

Say you have the following code for computing the cosine similarity of vectors X and Y:

$$\frac{\sum_{i=1}^{n} X_i \cdot Y_i}{\sqrt{\sum_{i=1}^{n} X_i^2} \cdot \sqrt{\sum_{i=1}^{n} Y_i^2}}$$

```
>>> from math import sqrt
>>> X = [1.0, 0.0, 0.0, 1.0, 0.0, 1.0] * 100
>>> Y = [0.0, 1.0, 1.0, 1.0, 0.0, 1.0] * 100
>>> similarity, norm_X, norm_Y = 0.0, 0.0, 0.0
>>> for x, y in zip(X, Y):
...     similarity += x * y
...     norm_X += x**2
...     norm_Y += y**2
>>> norm = sqrt(norm_X) * sqrt(norm_Y)
>>> similarity /= norm if norm > 0.0 else 1.0
>>> print(similarity)
0.5773502691896258
$ python -m timeit -s 'from ex1 import similarity' 'similarity()'
10000 loops, best of 3: 69.7 usec per loop
```

SISD to SIMD II

Modern processors support *vector instructions*, which can process many elements at once. By using the numpy library, we can perform the same operation much more efficiently:

```
>>> import numpy as np
>>> X = np.array([1.0, 0.0, 0.0, 1.0, 0.0, 1.0] * 100)
>>> Y = np.array([0.0, 1.0, 1.0, 1.0, 0.0, 1.0] * 100)
>>> similarity = np.dot(X, Y)
>>> norm = np.sqrt(np.sum(X)) * np.sqrt(np.sum(Y))
>>> similarity /= norm if norm > 0.0 else 1.0
>>> print(similarity)
0.5773502691896258
$ python -m timeit -s 'from ex2 import similarity' 'similarity()'
100000 loops, best of 3: 11.8 usec per loop
```

We have received a 7× speed-up. If we compute similarity between many vectors, the speed-up is even bigger, since numpy uses optimized BLAS matrix operations.

SIMD to SPMD I

Say you have the following code for preprocessing the TREC collection documents:

```
>>> from pv211_utils.trec.loader import load_documents
>>> from gensim.utils import simple_preprocess
>>> from tqdm import tqdm
>>>
>>> documents = load_documents(Document)
>>> tokenized_documents = []
>>> for document in tqdm(documents.values()):
...     tokenized_document = simple_preprocess(document.body)
...     tokenized_documents.append(tokenized_document)

  2%||                | 8689/527890 [00:12<11:24, 758.54it/s]
```

This is pretty to read, but also *pretty inefficient*, since we are using only a single CPU! :(

SIMD to SPMD II

On your notebook, you have up to 8 CPUs. On the `aura.fi.muni.cz` server, you have 64 CPUs. You can split documents into *chunks* and have CPUs process them in parallel:

```
>>> from pv211_utils.trec.loader import load_documents
>>> from gensim.utils import simple_preprocess
>>> from tqdm import tqdm
>>> from multiprocessing import Pool
>>>
>>> documents = load_documents(Document)
>>> with Pool(None) as pool:  # None means we use all CPUs!
...     document_bodies = (document.body for document in documents.values())
...     document_bodies = tqdm(document_bodies, total=len(documents))
...     tokenized_documents = pool.map(simple_preprocess, document_bodies)
 18%|||||||                          | 96961/527890 [00:06<01:22, 5205.96it/s]
```

On 64 CPUs, we have received a 7× speed-up. You can see that the scaling is not linear and there is overhead, since Python has to *pickle* and send `Document` objects to the individual CPUs and then pickle and retrieve the results. This is the price of Python's *GIL*. We get 50× speed-up by exchanging bigger chunks: `pool.map(..., chunksize=10000)`.

SIMD to SPMD III

The previous design pattern is not unique to Python. You can parallelize any *any program* that processes one file one at a time. Suppose you have a script called `tokenize.py`, which takes an input text file and produces a tokenized output text file:

```
$ for INPUT in input/*.txt
> do
>     OUTPUT=output/"${INPUT#input/}"
>     python tokenize.py "$INPUT" "$OUTPUT"
> done
```

You can use the GNU Parallel tool to run the script on all your CPUs in parallel:

```
$ parallel python tokenize.py {} output/{/} ::: input/*.txt
```

If your disk can catch up, we will receive a full 64× speed-up on 64 CPUs! ^_^

We can go even further and run our script on many different machines:

```
$ parallel --sshlogin user@machine1 --sshlogin user@machine2 ...
```

Unix admins beware!

Conclusion

Python is a nice language for writing logic, but slow for computation.
In order not to wither while waiting for results, you should obey the following rules:

- For vector and matrix computation, replace `for` loops with `numpy`.
- For processing many independent elements with single-CPU transformations, replace `for` loops with `multiprocessing` (or GNU Parallel).

I wish you the best of luck with your projects!

MUNI
FACULTY
OF INFORMATICS