Embarrassingly Parallel with Joblib

Simple parallel computing in Python and some other tools for performance.

Jason Neal

IA Programmers Club
27 May 2016

instituto de astrofísica e ciências do espaço

U.PORTO

python™

Purpose:

- Discuss what parallel processing is
- Embarrassingly parallel with Joblib
- Other Joblib features
- Goal:

  Be able to preform parts of our research faster.

Context:

- Needed to fit/optimise/minimise a function that contained a convolution.
- 1 Convolution would take ~20min to run
- Needed to make it faster

Parallel processing

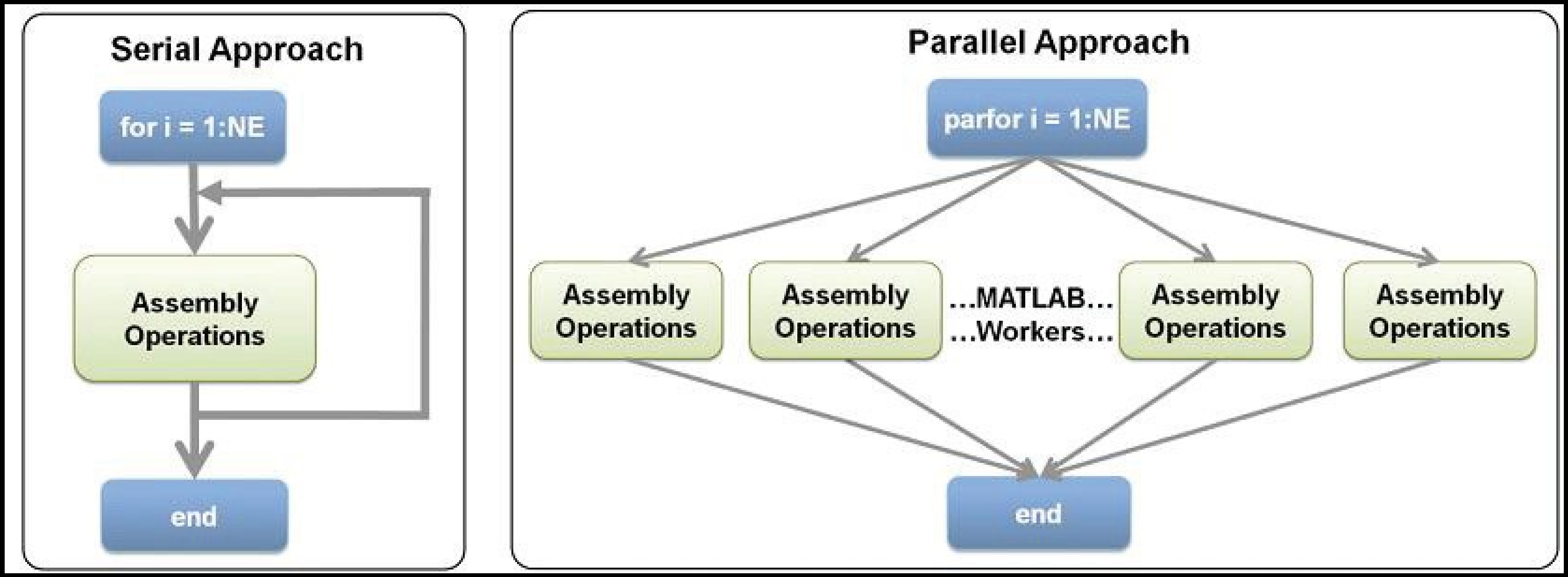
- Most (if not all) modern computers are Multicore computers
  - e.g. Dual Core (2), Quad Core (4)
  - 70% idle

- Splitting a computation into parts which run in different processes/CPU cores/Threads.

2 types of processing

- Serial – sequential, one after another

- Parallel – simultaneous, all or many at once.



Embarrassingly Parallel (pleasingly parallel)

- Where little or no effort is needed to separate the problem into a number of parallel tasks.
- little or no
  - dependency,
  - need for communication between those parallel tasks
  - or for results between them
- Each iteration is independent.

Joblib:

- Wrapper for multiprocessing library
- Embarrassingly parallel for loops
  - https://pythonhosted.org/joblib/parallel.html
- Other main features:
  - logging and tracing of the execution
  - transparent disk-caching of the output values and lazy re-evaluation (memoize pattern)
    - Caches the inputs and outputs of a function
    - If it is called with the same inputs then it just returns the result without recalculating it.
    - Suitable for large arrays

Installing Joblib

- Documentation says use :
  - easy_install joblib
- I would suggest trying these instead
  - conda install joblib
  - pip install joblib

https://pythonhosted.org/joblib/installing.html

Parallel for loops
Syntax Example

```
>>> from math import sqrt
>>> [sqrt(i ** 2) for i in range(10)]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

can be spread over 2 CPUs using the following:

```
>>> from math import sqrt
>>> from joblib import Parallel, delayed
>>> Parallel(n_jobs=2)(delayed(sqrt)(i ** 2) for i in range(10))
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

* Not recommended to actually implement this already fast calculation due to overheads.

Parallel reference documentation

class `joblib.`**`Parallel`**(*n_jobs=1*, *backend='multiprocessing'*, *verbose=0*, *pre_dispatch='2 * n_jobs'*, *batch_size='auto'*, *temp_folder=None*, *max_nbytes='1M'*, *mmap_mode='r'*)

Helper class for readable parallel mapping.

Parameters:

n_jobs: int, default: 1

The maximum number of concurrently running jobs, such as the number of Python worker processes when backend="multiprocessing" or the size of the thread-pool when backend="threading". If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

backend: str or None, default: 'multiprocessing'

Specify the parallelization backend implementation. Supported backends are:

- "multiprocessing" used by default, can induce some communication and memory overhead when exchanging input and output data with the with the worker Python processes.
- "threading" is a very low-overhead backend but it suffers from the Python Global Interpreter Lock if the called function relies a lot on Python objects. "threading" is mostly useful when the execution bottleneck is a compiled extension that explicitly releases the GIL (for instance a Cython loop wrapped in a "with nogil" block or an expensive call to a library such as NumPy).

verbose: int, optional

The verbosity level: if non zero, progress messages are printed. Above 50, the output is sent to stdout. The frequency of the messages increases with the verbosity level. If it more than 10, all iterations are reported.

pre_dispatch: {'all', integer, or expression, as in '3*n_jobs'}

The number of batches (of tasks) to be pre-dispatched. Default is '2*n_jobs'. When batch_size="auto" this is reasonable default and the multiprocessing workers shoud never starve.

batch_size: int or 'auto', default: 'auto'

The number of atomic tasks to dispatch at once to each worker. When individual evaluations are very fast, multiprocessing can be slower than sequential computation because of the overhead. Batching fast computations together can mitigate this. The `'auto'` strategy keeps track of the time it takes for a batch to complete, and dynamically adjusts the batch size to keep the time on the order of half a second, using a heuristic. The initial batch size is 1. `batch_size="auto"` with `backend="threading"` will dispatch batches of a single task at a time as the threading backend has very little overhead and using larger batch size has not proved to bring any gain in that case.

temp_folder: str, optional

Folder to be used by the pool for memmaping large arrays for sharing memory with worker processes. If None, this will try in order: - a folder pointed by the JOBLIB_TEMP_FOLDER environment variable, - /dev/shm if the folder exists and is writable: this is a RAMdisk

filesystem available by default on modern Linux distributions,

- the default system temporary folder that can be overridden with TMP, TMPDIR or TEMP environment variables, typically /tmp under Unix operating systems.

Only active when backend="multiprocessing".

max_nbytes int, str, or None, optional, 1M by default

Threshold on the size of arrays passed to the workers that triggers automated memory mapping in temp_folder. Can be an int in Bytes, or a human-readable string, e.g., '1M' for 1 megabyte. Use None to disable memmaping of large arrays. Only active when backend="multiprocessing".

Parallel Issues:

- Overheads
  - Time to initalize/organize parallel processes
- Risk of opening/destroying multiple workers

```
>>> with Parallel(n_jobs=2) as parallel:
...     accumulator = 0.
...     n_iter = 0
...     while accumulator < 1000:
...         results = parallel(delayed(sqrt)(accumulator + i ** 2)
...                            for i in range(5))
...         accumulator += sum(results)  # synchronization barrier
...         n_iter += 1
...
>>> (accumulator, n_iter)
(1136.596..., 14)
```

- Memory
  - The data given to Parallel is reproduced n_job times.

Memmapping

Shared memory location in a temporary file

- Automatic if the array exceeds max_nbytes parameter of Parallel
- Can use to manually free up memory space if you have large arrays.
-
- Use joblib.dump(), joblib.load()

Manual management of memmaped input data

For even finer tuning of the memory usage it is also possible to dump the array as an memmap directly from the parent process to free the memory before forking the worker processes. For instance let's allocate a large array in the memory of the parent process:

```
>>> large_array = np.ones(int(1e6))
```

Dump it to a local file for memmaping:

```
>>> import tempfile
>>> import os
>>> from joblib import load, dump

>>> temp_folder = tempfile.mkdtemp()
>>> filename = os.path.join(temp_folder, 'joblib_test.mmap')
>>> if os.path.exists(filename): os.unlink(filename)
>>> _ = dump(large_array, filename)
>>> large_memmap = load(filename, mmap_mode='r+')
```

The `large_memmap` variable is pointing to a `numpy.memmap` instance:

```
>>> large_memmap.__class__.__name__, large_array.nbytes, large_array.shape
('memmap', 8000000, (1000000,))

>>> np.allclose(large_array, large_memmap)
True
```

We can free the original array from the main process memory:

```
>>> del large_array
>>> import gc
>>> _ = gc.collect()
```

Go forth and be efficient

Links

- Embarrassingly parallel for loops
  - https://pythonhosted.org/joblib/parallel.htm
- Parallel processing python related libraries
  - https://wiki.python.org/moin/ParallelProcessing
- Youtube video on Joblib
  - https://www.youtube.com/watch?v=nEyYt-CHRZo