README

What the artifact does

PyPar is a tool for automatically discovering parallelization possibilities in Python programs. It leverages data dependence analysis and graph-theoretic methods to find parallelisms, and uses dynamic analysis to select useful parallelisms.

Where it can be obtained

We have uploaded the PyPar tool to GitHub. See https://github.com/PyParTool/PyPar

How to install and use it

See `readme.md` in the GitHub repository. The contents are repeated here.

Installation

requirements:

```
python==3.8.10
astunparse==1.6.3
```

If you want to run `usage.py`:

```
scikit-image==0.19.3
```

install:

```
python3 setup.py sdist
python3 setup.py install --prefix ~/.local/
```

Usage

Prepare input

Suppose we want to find parallelisms in function `skimage.filters.ridges.frangi`:

```
from skimage.filters.ridges import frangi
```

Prepare input for this function:

```
import numpy as np
N = 1000
np.random.seed(1)
image = np.random.uniform(size=(N, N), low=0.0, high=1.0)
```

The code to run the target function:

```
code = 'frangi(image)'
```

Discover parallelism

Use `DynamicParallelizer` to find parallelisms:

```
from pypar import DynamicParallelizer
parallelizer = DynamicParallelizer(
    code=code,
    glbs=globals(),
    lcls=locals())
```

The detected parallelisms are recorded in `DynamicParallelizer` object.

Print parallelism report

```
from pypar import print_parallelizables
print_parallelizables(parallelizer)
```

The parallelism report is like this:

```
==================================================
('/path_to_skimage_package/skimage/filters/ridges.py', 'ridges', 'frangi')
Loop
--------------------------------------------------
Code:
for (i, sigma) in enumerate(sigmas):
    (lambda1, *lambdas) = compute_hessian_eigenvalues(image, sigma,
sorting='abs', mode=mode, cval=cval)
    r_a = (np.inf if (ndim == 2) else (_divide_nonzero(*lambdas) ** 2))
    filtered_raw = (np.abs(np.multiply.reduce(lambdas)) ** (1 / len(lambdas)))
    r_b = (_divide_nonzero(lambda1, filtered_raw) ** 2)
    r_g = sum(([(lambda1 ** 2)] + [(lambdai ** 2) for lambdai in lambdas]))
    filtered_array[i] = (((1 - np.exp(((- r_a) / alpha_sq))) * np.exp(((- r_b) /
beta_sq))) * (1 - np.exp(((- r_g) / gamma_sq))))
    lambdas_array[i] = np.max(lambdas, axis=0)
--------------------------------------------------
compute_hessian_eigenvalues(image, sigma, sorting='abs', mode=mode, cval=cval)
--------------------------------------------------
N_loop:  5
parallel_degree:  5
expected parallel time:  0.4581724658450882
```

`('/path_to_skimage_package/skimage/filters/ridges.py', 'ridges', 'frangi')` gives the parallelizable function.

`Loop` indicates the parallelism is within a loop.

`Code: ...` gives the code piece that has parallelization possibilities.

`compute_hessian_eigenvalues(image, sigma, sorting='abs', mode=mode, cval=cval)` gives the parallelizable task.

`expected parallel time:  0.4581724658450882` gives the expected running time after parallelization.

Print parallelized code

```
from pypar import print_rewrite
print_rewrite(parallelizer, 0)
```

This prints the parallelized code (using Ray).

More

For more information, see `docs/` directory in the GitHub repository.

A small example

See `usage.py` in the GitHub repository.

How to reproduce the results presented in the paper

See `reproducing_pipeline/readme.md` in the GitHub repository. The contents are repeated here.

The directory `reproducing_pipeline` contains scripts for reproducing the experiment results in the published work.

These scripts provide following functionalities:

- collect function calls from `pytest` scripts and example programs of target package
- generate input of suitable size for these functions
- discover parallelisms and rewrite
- measure acceleration

Requirements

```
pytest==7.2.1
ray==1.11.0
```

In the published work, we conduct experiments on 6 packages, their versions are:

```
scikit-image==0.19.3
scipy==1.9.1
librosa==0.9.2
trimesh==3.16.3
scikit-learn==1.2.2
seaborn==0.11.2
```

Structure

`0.collect.1.py`, `0.collect.2.py`, `1.merge.py`, `2.get_scalable.py`, `3.parallelize.py`, `4.rewrite.py` and `5.eval.py` constitute a pipeline. One should run them serially to reproduce the experiment results.

`filter.py`, `generator.py` and `inputgenerate.py` provide utilities for input generation.

`stats.py` is used to store relevant information.

`2.log.txt` and `timing/*` are intermediate results (on package `Seaborn`).

Workflow

collect function calls and their arguments

```
python3 0.collect.1.py
```

This script collects calls to target package's functions and their arguments from `pytest` scripts, and store them to `0.funcArgs.pkl`.

To run the script, change `'/path_to_seaborn_package'` in the script to path to target package (usually it is `'~/.local/lib/python3.8/site-packages/package_name'`).

```
python3 0.collect.2.py
```

This script collects calls to target package's functions and their arguments from example programs and store them to `./pkls/` directory. The example programs are provided by many packages, for example, https://github.com/scikit-learn/scikit-learn/tree/main/examples, https://github.com/mwaskom/seaborn/tree/master/examples.

To run the script, change `'/path_to_seaborn_package'` in the script to path to target package (usually it is `'~/.local/lib/python3.8/site-packages/package_name'`). Change `'/path_to_pkls/'` in the script to absolute path of `./pkls`. Download the example programs, and change `'/path_to_example_programs'` in the script to the absolute path of the example programs.

```
python3 1.merge.py
```

This script merges the function calls and arguments in `0.funcArgs.pkl` and `./pkls/`, and store them to `1.funcArgs.pkl`.

The number of function calls collected should be assigned to `n_traced` in `stats.py`.

generate inputs

This step generate inputs of proper size for function calls collected by previous step.

The generated inputs will be used to run the target function.

```
python3 2.get_scalable.py
```

This script finds functions that can run for longer than 1 second by scaling input arguments, and store them to `2.log.txt`.

The generated inputs in `2.log.txt` should also be assigned to `funcN` in `stats.py`.

The manually generated inputs should be stored in `funcInput` in `stats.py`.

discover parallelisms and rewrite

```
python3 3.parallelize.py
```

This script uses `DynamicParallelizer` to find parallelizable functions, and store them to `3.parallelizable_funcs.pkl`.

```
python3 4.rewrite.py
```

This script rewrites the parallelizable functions into parallelized versions (using `concurrent.futures` and `ray`), and print them to `stdout`.

One should check the correctness of parallelisms. If the parallelism is not false positive, one can paste the parallelized version to target package.

measure acceleration

```
python3 5.eval.py
```

This script runs the parallelizable functions and their parallelized versions for 100 times, and dump the timing results to `./timing/` directory.