Final Project Report

Nawat Ngerncham

July 2022

1 How the project works

1.1 Testing the Project

To get started, run the following commands:

```
python3 -m venv venv
source ./venv/bin/activate
pip install numpy maturin
```

To compile and install the library, run the following command after ensuring that the virtual environment is already activated:

```
maturin develop --release
```

Once installed, run the test script given under `test.py`. To use in Python, simply import the library, `numpar`.

1.2 Implementation

Since the choice of which numeric algorithm to use for each operation is not the main focus of this project, we will ignore that for the most part.

The project is implemented in Rust and uses PyO3 to bind Rust code to Python. For parallelism, Rayon is used. This library contains heavy use of Rayon's parallel iterators. Honestly, probably too much.

2 Findings

I would like to point out that some of the findings are not used or implemented in the code base but were things that I've found out from experiments while trying to improve the speed (though a lot has not helped). Thus, I would like to split my findings into direct and indirect findings.

2.1 Direct

Turns out that coding up numerical computations in linear algebra proves to be very difficult, let alone parallelizing it and making it faster than a *well-optimized single-core (by default) library.*

In general, functions that deal with vectors and vectors only can get some speed up (up to 2x) from vanilla NumPy. I speculate that this is just due to simplicity of vector operations which are simply iterating a 1D Vector in parallel.

However, anything that has to do with a matrix is significantly slower (avg: 0.3-0.6x slower but down to 0.01x in certain cases) with parallelization. I speculate that this is due to the cache-unfriendliness of multi-threaded memory access.

2.2 Indirect

Interesting findings I've found from my (somewhat useless) experiments in no particular order:

- Simply slapping on a parallel iterator on a simple *ikj* matrix multiplication can give a significant speed boost.

- Matrix operations on a 2D Vector is generally faster than operating a row-major 1D Vector representation of the same matrix.
- While operations on already 2D Vector is generally faster, it has a significantly higher overhead when it comes to generating/converting a row-major a 2D Vector.
- Using atomic floats in row-major form is slower than 2D Vector of floats, but faster than using parallel chunking on row-major of floats.
- Best way to generate empty row-major matrix with `AtomicF64(0.0)` is to use range then parallel map them with *slightly* better performance.

3 What I'd like you and the world to know

I can't really say that I am happy with how this project turned out.

The target performance may have been too ambitious or slightly unrealistic for me to achieve but I feel like I can at least do a bit better if I spend enough (read: way longer than a couple weeks) time on it.

I plan to continue to spend some more time on it during the break unless I feel super burnt out (which is also likely the case) so I wanna say that *this isn't its final form yet* but it might as well be so please also grade it as is. Thank you!

PS. *Go use NumPy for any numerical computation. Anything professional-baked is most likely to be better than anything home-baked.*