The Bifrost

Release 1.0

Andrew Garcia, Ph.D.

Apr 05, 2023

CONTENTS

1 A bridge to move your data structures across different programming languages **1**

2 Contents **3**
2.1 Usage 3
2.2 API Reference 8

Index **9**

CHAPTER

ONE

A BRIDGE TO MOVE YOUR DATA STRUCTURES ACROSS DIFFERENT PROGRAMMING LANGUAGES

This is an open-source suite of mini-libraries designed to connect array data between Python and C++ through the use of JSON files. Python and C++ modules use tojson and jsonload functions to convert [one- two- and three-dimensional] arrays to .json DOK (Dictionary of Keys) and viceversa, respectively

Check out the *Usage* section for further information, including how to *Installation* the project.

Note: This project is under active development.

CHAPTER

TWO

CONTENTS

2.1 Usage

2.1.1 Installation

It is recommended you use voxelmap through a virtual environment. You may follow the below simple protocol to create the virtual environment, run it, and install the package there:

```
$ virtualenv venv
$ source venv/bin/activate
(.venv) $ pip install tensorscout
```

To exit the virtual environment, simply type `deactivate`. To access it at any other time again, enter with the above `source` command.

2.1.2 Monte Carlo sampling with multiple processors

When performing Monte Carlo sampling at a high number, it can significantly impact computing power. To address this, we have developed the `@multicarlo` decorator, which allocates a specific number of iterations to a defined number of available processors or cores. In our case, since we have a computer with 4 cores, we have set the num_cores to 4. However, you can set it to as many cores as your computer or server may have available.

In this example, we compare the runtime performance of this multiprocessing decorator with the bare approach, which uses a single core. We begin by importing all the required modules and defining a function that is used in both approaches to avoid redundancy.

```
import tensorscout as scout
import numpy as np
import matplotlib.pyplot as plt
from timethis import timethis

def make_histograms(results, title):
   plt.figure()
   plt.title(title+'(N = 100,000)')
   plt.hist(data,alpha=0.5,label='data')
   plt.hist(results,alpha=0.5,label=title)
   plt.legend()
```

The operations we run on both methodologies are random sampling operations which take random numbers from a distribution. For both methods, we set the number of samples to 100,000, which is a considerable amount. In the following code block, we apply the `@multicarlo` decorator to our random sampling function `monte_carlo_function` and distribute the sampling iterations across four cores.

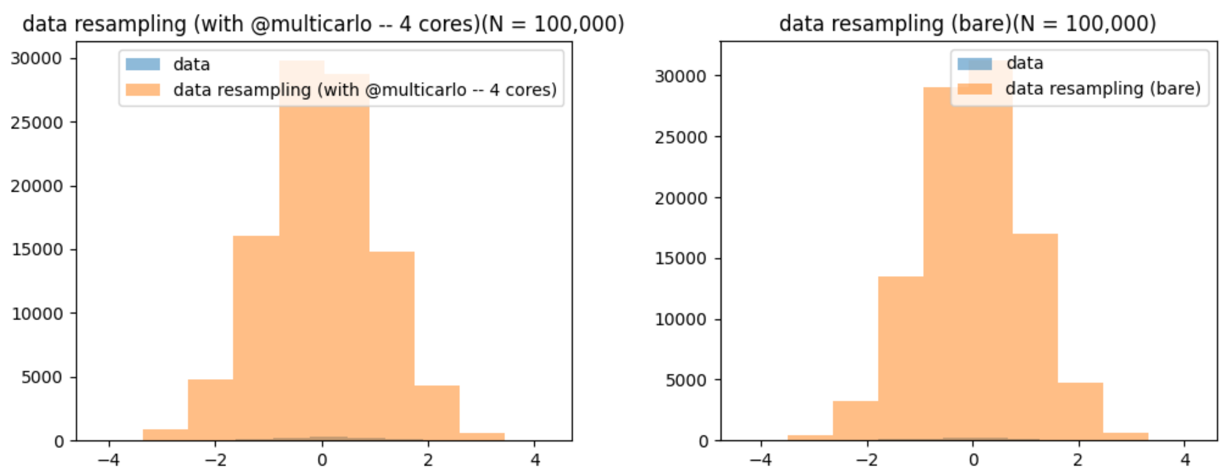
The `timethis()` function is used to record the run times of both methods and print them as a terminal output.

```
title = 'data resampling (with @multicarlo -- 4 cores)'
with timethis(title):

   @scout.multicarlo(num_iters=100000, num_cores=4)
   def monte_carlo_function(data, *args, **kwargs):
      simulated_data = np.random.normal(np.mean(data), np.std(data))
      return simulated_data

   data = np.random.normal(0, 1, 1000)
   results = monte_carlo_function(data)
   make_histograms(results,title)
```

The following code block executes the same tasks as the previous block, but using a bare approach, meaning that it uses a single core to perform all 100,000 random samples.

```
title='data resampling (bare)'
with timethis(title):

   def monte_carlo_function_bare(data, *args, **kwargs):
      simulated_data = np.random.normal(np.mean(data), np.std(data))
      return simulated_data

   data = np.random.normal(0, 1, 1000)
   results = [monte_carlo_function_bare(data) for i in range(100000)]
   make_histograms(results,title)

plt.show()
```

The output for the previous three code blocks is displayed below.




```
>>> [OUT]
monte carlo resampling (with @multicarlo -- 4 cores): 2.812 seconds
```

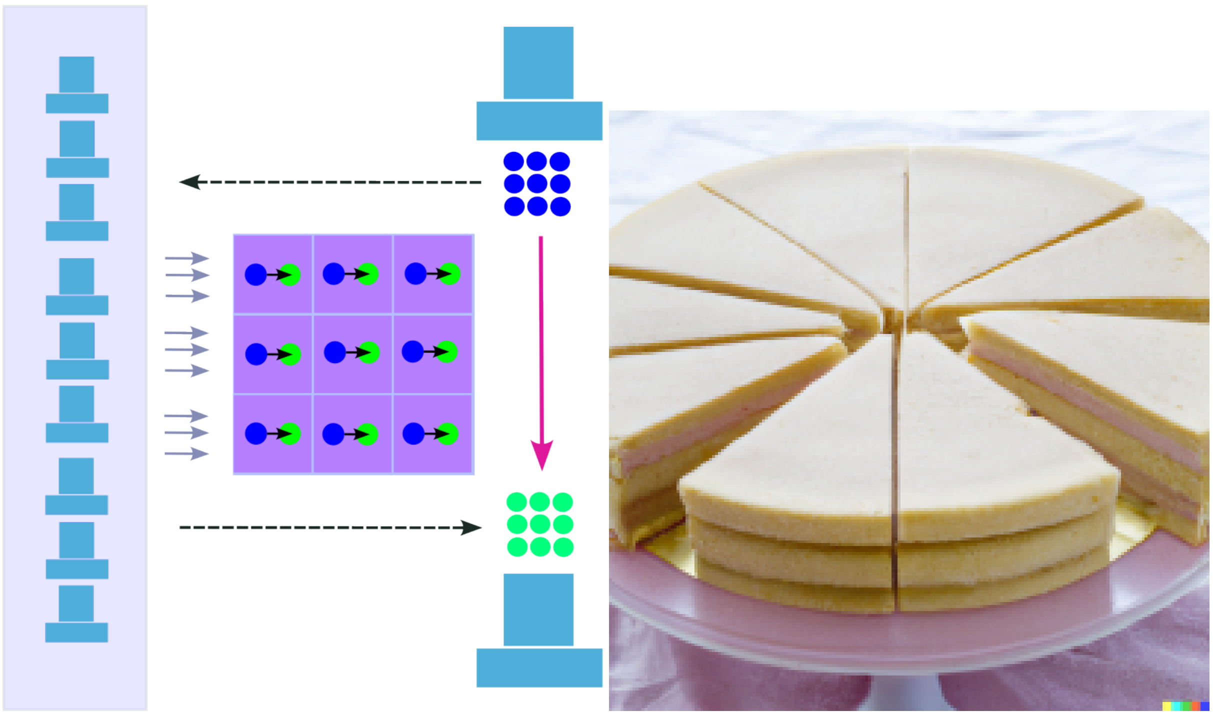
(continues on next page)

(continued from previous page)

```
monte carlo resampling (bare): 4.328 seconds
```

Both approaches produce a similar random sampling distribution outcome, but the `@multicarlo` decorated function that uses multiprocessing on 4 cores shows around 50% better runtime performance.

2.1.3 Parallel Computation on Sectorized Matrices using Multiprocessing



The question of whether it's faster to eat a cake alone or have 100 people cut a slice and eat their portions until it's gone highlights the main concept behind the cakerun decorator. Essentially, the decorator partitions an array into a specified number of equally-sized sectors and performs the same task on all sectors in parallel.

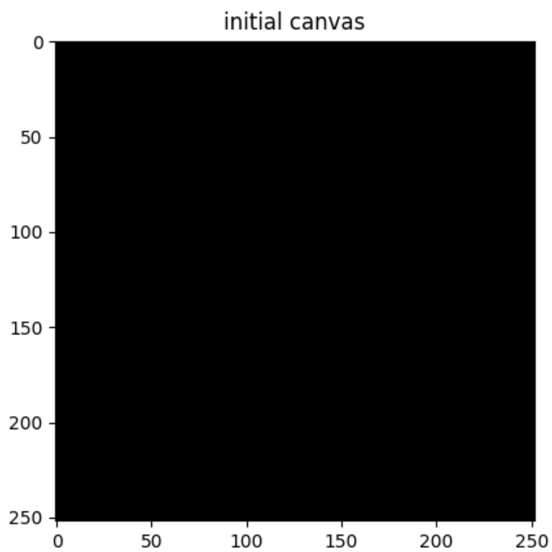
In this example, we set the number of cores to 4 and compare the performance of using multiprocessing versus using a single core. Before proceeding, we import all necessary modules and define the draw function which is used in both approaches to avoid redundancy. Additionally, we define the initial matrix, which is a 252 x 252 matrix of 1s, that will be operated on by both methodologies.

```
import tensorscout as scout
import numpy as np
import matplotlib.pyplot as plt
from timethis import timethis

num_iters = 40000

def draw(result):
   plt.figure()
   plt.title('{} -- $N_{{perforated}}$ = {}'.format(title, np.multiply(*result.shape) - np.count_nonzero(result)))
   plt.imshow(result,cmap='bone')
```

(continues on next page)

(continued from previous page)

```
matrix = np.ones((252,252))

plt.imshow(matrix,cmap='bone')
plt.title('initial canvas')
```



In this example, the initial matrix is composed entirely of 1s and will appear as a single color when drawn. The purpose of this code is to apply an operation called "perforation" to the matrix. At each iteration, a random x-y coordinate is selected and the value at that location is set to 0.

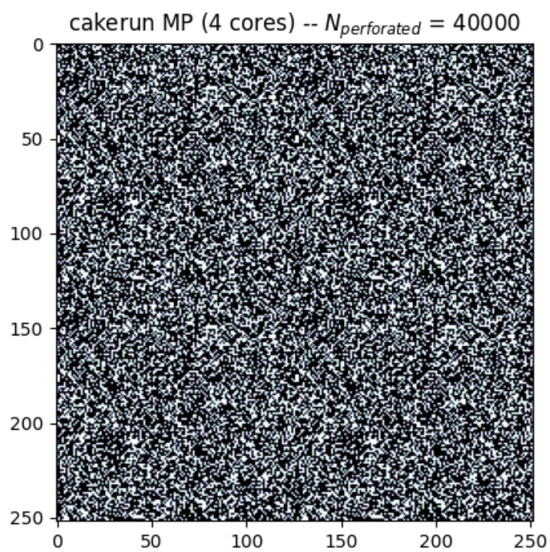
The first case demonstrates the use of the `@cakerun` decorator to split the matrix into sectors and apply the perforate function to each sector. The former code block specifies 40,000 perforating iterations, which for the case of this aprroach has them evenly distributed across the 4 sectors, resulting in 10,000 iterations per sector, ocurring simultaneously.

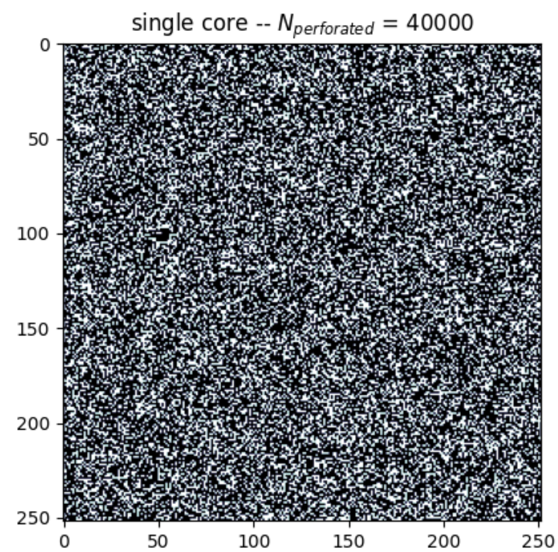
```
title = 'cakerun MP (4 cores)'
with timethis("{}".format(title)):

   cores = 4
   @scout.cakerun(num_cores=cores, L_sectors=2)
   def perforate(sector):

      for i in range(num_iters // cores):
            cds = np.argwhere(sector!=0)
            sector[tuple(cds[np.random.randint(cds.shape[0])])] = 0
      return sector

   result = perforate(matrix)
   draw(result)
```

In the next code block, the perforating operation is applied for 40,000 iterations using a bare approach with a single processor. Hence, there is no task split involved.

```
title = 'single core'
with timethis("{}".format(title)):

   def perforate_bare(sector):
      for i in range(num_iters):
            cds = np.argwhere(sector!=0)
            sector[tuple(cds[np.random.randint(cds.shape[0])])] = 0
      return sector


   result = perforate_bare(matrix)
   draw(result)


plt.show()
```

The following are graphical and runtime comparisons of both methods:






```
>>> [OUT]
cakerun MP (4 cores): 2.968 seconds
single core: 25.868 seconds
```

It is apparent that both approaches yield a similar outcome and have the same number of perforations. However, the `@cakerun` decorated function, which uses four cores simultaneously, has a runtime that is 8-9 times faster than the bare approach.

2.2 API Reference

2.2.1 Global Methods

At the time, tensorscout is a lean module composed of only 2 decorators.

`class tensorscout.multicarlo(num_iters, num_cores)`

This decorator performs a non-dynamic operation or task for a specified number of iterations num_iters and distributes the tasks across a requested number of available processors num_cores.

Parameters

num_cores: int
Number of processors to use

num_iters
[int] The number of iterations to perform for a specific model / Monte Carlo simulation.

`class tensorscout.cakerun(num_cores, L_sectors)`

This decorator partitions an array into sectors and applies a given function to each sector in parallel. The result of each computation is merged into a final output array.

Parameters

num_cores: int
Number of processors to use

L_sectors
[int] The length scale for the number of sectors [per column]. For non-square arrays, the number of sectors per row gets adjusted as a function of this value

INDEX

C

`cakerun` (*class in tensorscout*), 8

M

`multicarlo` (*class in tensorscout*), 8