Parallel Galactic Swarm Optimization

Project Description

Deep Learning is a family of machine learning algorithms which involve a deeper cascade of smaller learning models than traditional machine learning models have. These algorithms are used extensively for automate tasks which were previously thought to be only possible for humans to do. Today the go-to algorithm for optimizing the deep learning algorithms is Gradient Descent. Although, it works very well in practice. It gets stuck when there are multiple optimal solutions available for a given function. i.e A multimodal function. In practice a different version of it called Stochastic Gradient Descent introduces stochasticity to help navigate out of tough spots but the requirement of finding gradients limits its use case to just that.

Fortunately, We have a family of algorithms that are very good at dealing with the functions which are highly non-convex, the kind of ones for which we cannot find gradients to, called the metaheuristic algorithms. A research work was recently published by our university which proposed an algorithm called Galactic Swarm Optimization (GSO) which is currently the state-of-the-art. GSO doesn't need any gradient finding. So even for non-convex functions (the kind of ones which have the min/max) are also suitable. What it lacked was that it could not scale in with hardware.

Our primary contribution is that we have come up with a solution which modifies the GSO algorithm to scale in with increase in hardware. We have also created a library which is open-sourced on github along with provisions to rerun the benchmarks.

Related Work

Particle Swarm Optimization (PSO)

PSO is inspired from how birds search for food or how a school of fishes navigate. Where a bird is termed as a particle and the search for food/best global optima is guided by each particls position, velocity and intertia.Each particle's movement is influenced by its local best known position, but is also guided toward the best known positions in the search-space, which are updated as better positions are found by other particles. This is expected to move the swarm toward the best solutions. The equation for this is given by -

$$V^{i}_{(t+1)} = Currentmotion + ParticleMemoryInfluence + SwarmInfluence$$

or,

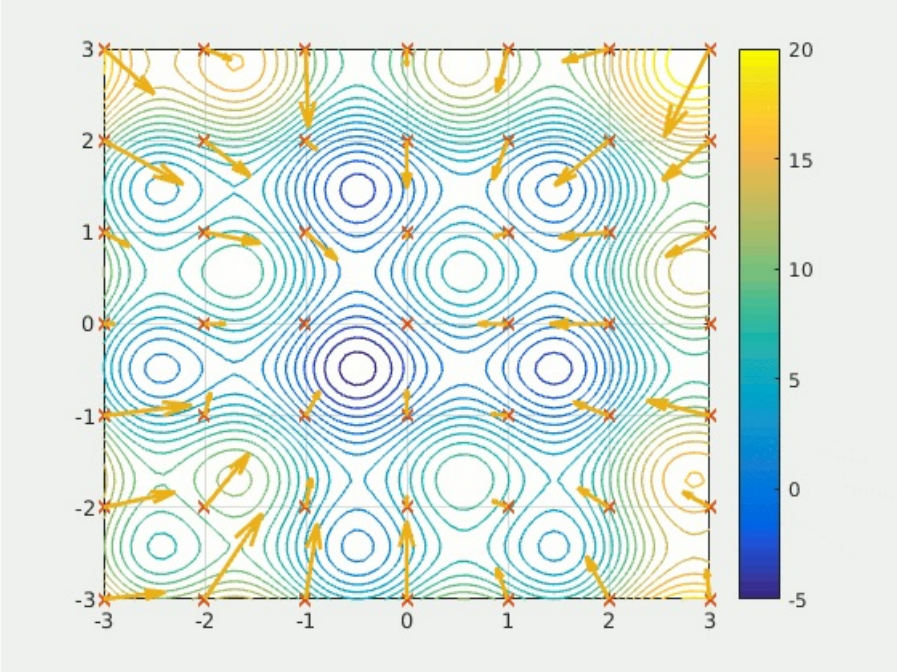
$$V^{i}_{(t+1)} = wv^{i}_{t} + C_1 r_1 (p^{i}_{t} - x^{i}_{(t)}) + C_2 r_2 (G - x^{i}_{(t)})$$

where C_1 and C_2 are cognition and social learning factors respectively and r_1, r_2 are randomly generated numbers in the range [0,1], G is the global best, p^{i}_{t} is the local best and v^{i}_{t} is the velocity of the particle at time t.

The next position of the particle is determines as follows -

$$X^{i}_{(t+1)} = x^{i}_{(t)} + v^{i}_{(t+1)}$$

Simulation of PSO



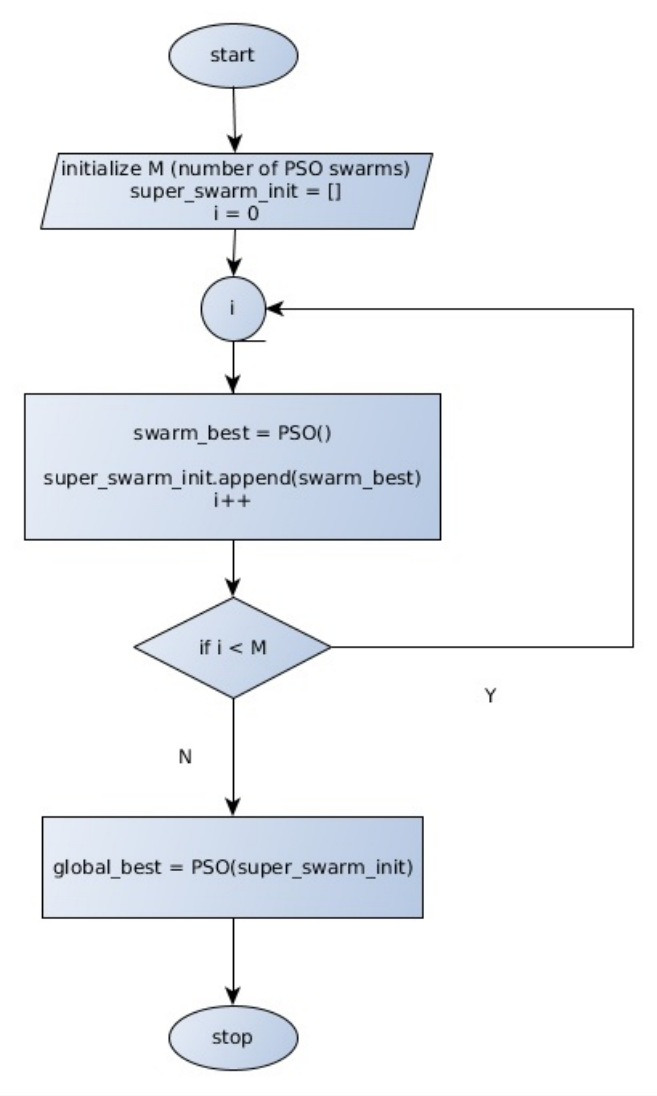
Credits - Ephramac

Implementation details

The sequential GSO algorithm

The GSO algorithm is a modification of the PSO algorithm which eliminates the pain points of the PSO algorithm. Most variants of PSO first have a full exploration phase which gradually becomes a full exploitation by using the learning rate decay to strike the balance. GSO has multiple cycles of exploration and exploitation by dividing search in terms of epochs. This allows us to explore the global minima more accurately. Consider each galaxy as a subswarm which have a centre of mass. These galaxies are part of a larger supercluster. Where they look like point masses revolving inside. Now using PSO we find the best solution a.k.a the centre of mass of galaxy which represents the galaxy in the supercluster. Now these representative points are used to find centre of mass of this large supercluster. This heirarchy can go on even more but we currently restrict it to 2 levels. We use PSO to find the centre of mass of a galaxy/supercluster. The algorithm looks as follows-

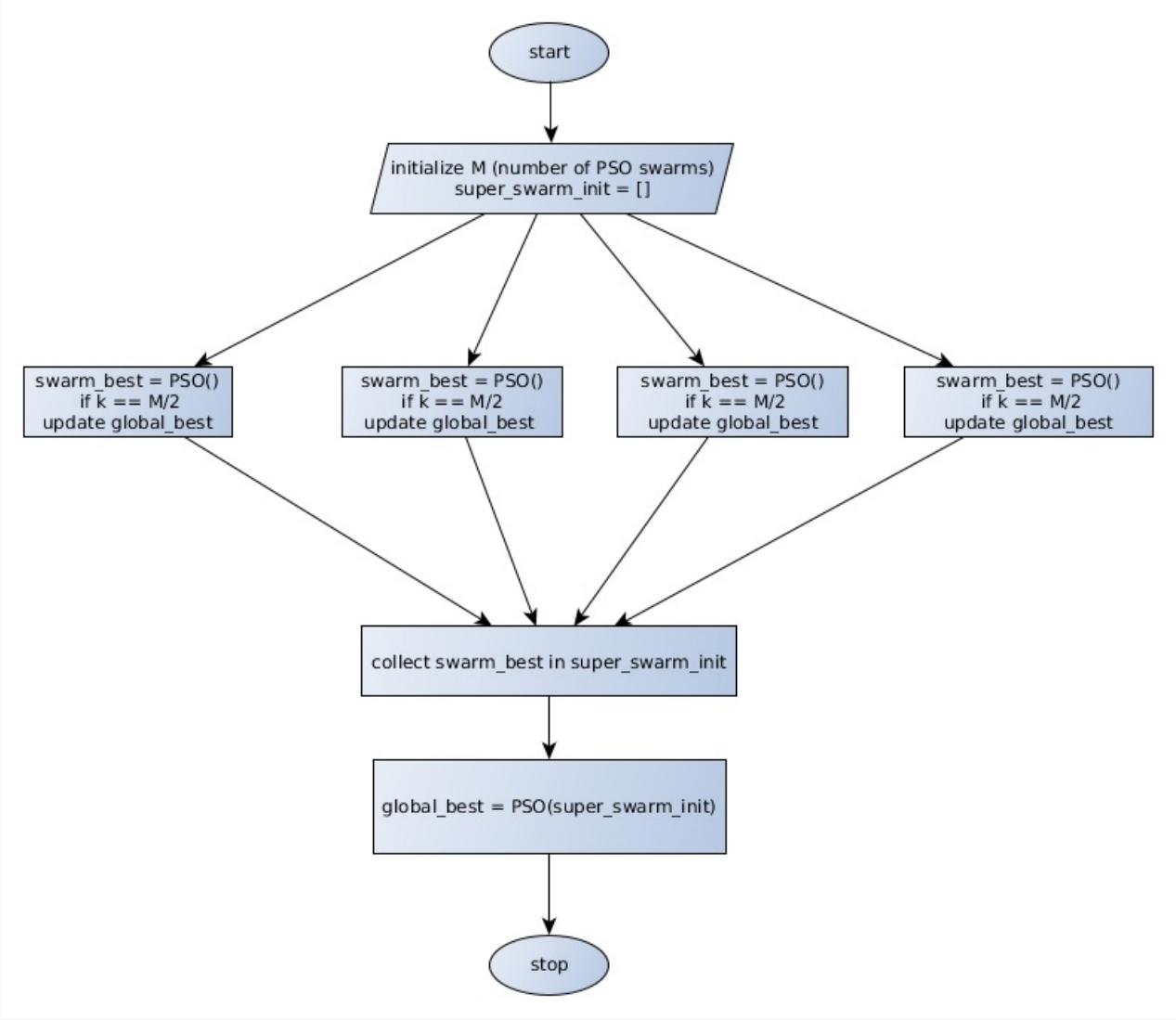
```
def GSO(M, bounds, num_particles, max_iter):
    subswarm_bests = []
    dims = len(bounds)
    lb, ub = bounds

    for i in range(M):
        swarm_init = list of randomly initialized num_particles in the range (lb, ub)
        subswarm_best,_ = PSO(error,bounds,maxiter, swarm_init=swarm_init)
        subswarm_bests.append(subswarm_best)
    best_position, best_error = PSO(error, bounds, maxiter, swarm_init=subswarm_bests)
    return best_position, best_error
```



Bottleneck identification:

We can see that there is a for-loop above where we are calling PSO function M times and collecting the output sequentially can be identified as a clear case where we can apply **SIMD (Single Instruction Multiple Data)** based parallelism. So we fork out M threads which have their own stack. We tried not to use any shared datastructure for exchanging information in this fork procedure since we do not want any latency introduced due to synchronization. But in practice we have found that doing information exchange midway on where is the best solution greatly speeds up our exploitation phase, the global_best variable is present in the shared_list internally present inside a heap visible to all threads. We have introduced a lock region which gets activate midway once, where each PSO thread updates on where is the global best. This tradeoff is done in order to encourage exploitation which ultimately helps us reach the goal faster. Below is a diagram representing the proposed and implemented fork-join model -

The parallel GSO diagram



Software specifications

Libraries

- **Numba** - Used for speeding up math-intensive computations in python and maximizing CPU utilization
- **multiprocessing** - Used for spawning threads with PSO function calls
- **numpy** - Used some standard factory functions from numpy which have C like performance and are implicitly highly parallel
- **matplotlib, seaborn** - plotting graphs of CPU utilization and functions

Profiling tools

- **line_profiler** - For getting line by line execution time and number of hits
- **timeit** - For timing the whole function and taking the best average among top N executions
- **psutil** - For checking individual CPU utilization when our algorithm runs (reading taken at 1ms interval).

Code Optimizations

Generally whenever we prototype an algorithm the general strategies that were followed by famous libraries like scikit-learn, numpy are to profile the expensive parts and write them in C and then use the Cython API to call the C code into your python program. But, that is no longer the case. Why? - Because we have Numba. Numba lets you write C like performant Python Code. All you have to do is to know where and how to gain maximum performance. Our implementation is **fully vectorized and multi-threaded**. Changes we have made in our code to gain performance are as follows -

1. Defined a custom numpy datatype for creating particle objects instead of creating class Particle. The benefit we got from this was that Numba recognizes numpy datatype because it then knows what size it can take and therefore the intermediate bytecode generated by the LLVM compiler can assign type easily to the numpy object instead of it being a standard python object with no type definitiion. In short, **this helped us with save Memory** and also made it easier for Numba to recognize and generate efficient intermediate bytecode.
2. Inherently, numpy code is faster than Numba code (by a small factor) and therfore we have used Numpy methods in our code where possible because, numpy is heavyily optimized and scales in smoothly with increase in number of cores.
3. Used Numba's just in time compilation for each method making sure code written in each function is easily recognized by

numba (see examples on how we do it for a sample function) which **helped gain maximum performance**.

4. After a one time run, code is automatically cached and compilation is not needed again even for the JIT compiler.
5. Vectorized IO - Input/Output for all functions are N-dimensional numpy arrays. All transformations performed on them do not have any excess overhead and helped us with speed gains.
6. MultiThreading - Spawning PSO functions as threads using the multiprocessing library. Threading allows multiple PSO functions to run parallely.

Simple Numba Demo to see which part of code numba speeds up

Function name: create_n_particles
in file: /home/souldiv/py_rep/parallelGSO/cuda_code/final/init_particles.py
with signature: (int64, int64, reflected list(array(float64, 1d, C))) -> pyobject

show numba IR

```
5:  @jit
6:  def create_n_particles(n, num_dimensions, swarm_init):
7:
8:      particle_dtypes = np.dtype({'names':['position_i', 'velocity_i', 'pos_best_i',
9:                                           'err_best_i', 'err_i', 'num_dimensions'],
10:                                 'formats':[(np.double,(1,(num_dimensions))),
11:                                            (np.double,(1,(num_dimensions))),
12:                                            (np.double,(1,(num_dimensions))),
13:                                            np.double, np.double, np.int32]
14:                                })
15:     particles = np.empty(n, dtype=particle_dtypes)
16:     for p,x0 in zip(particles, swarm_init):
17:         p['err_best_i'] = -1
18:         p['err_i'] = -1
19:         p['num_dimensions'] = num_dimensions
20:         for i in range(num_dimensions):
21:             p['velocity_i'][0][i] = np.random.uniform(-1,1)
22:             p['position_i'][0][i] = x0[i]
23:     return particles
```

The green part shows which part is sped up in the whole code.

Performance Numbers

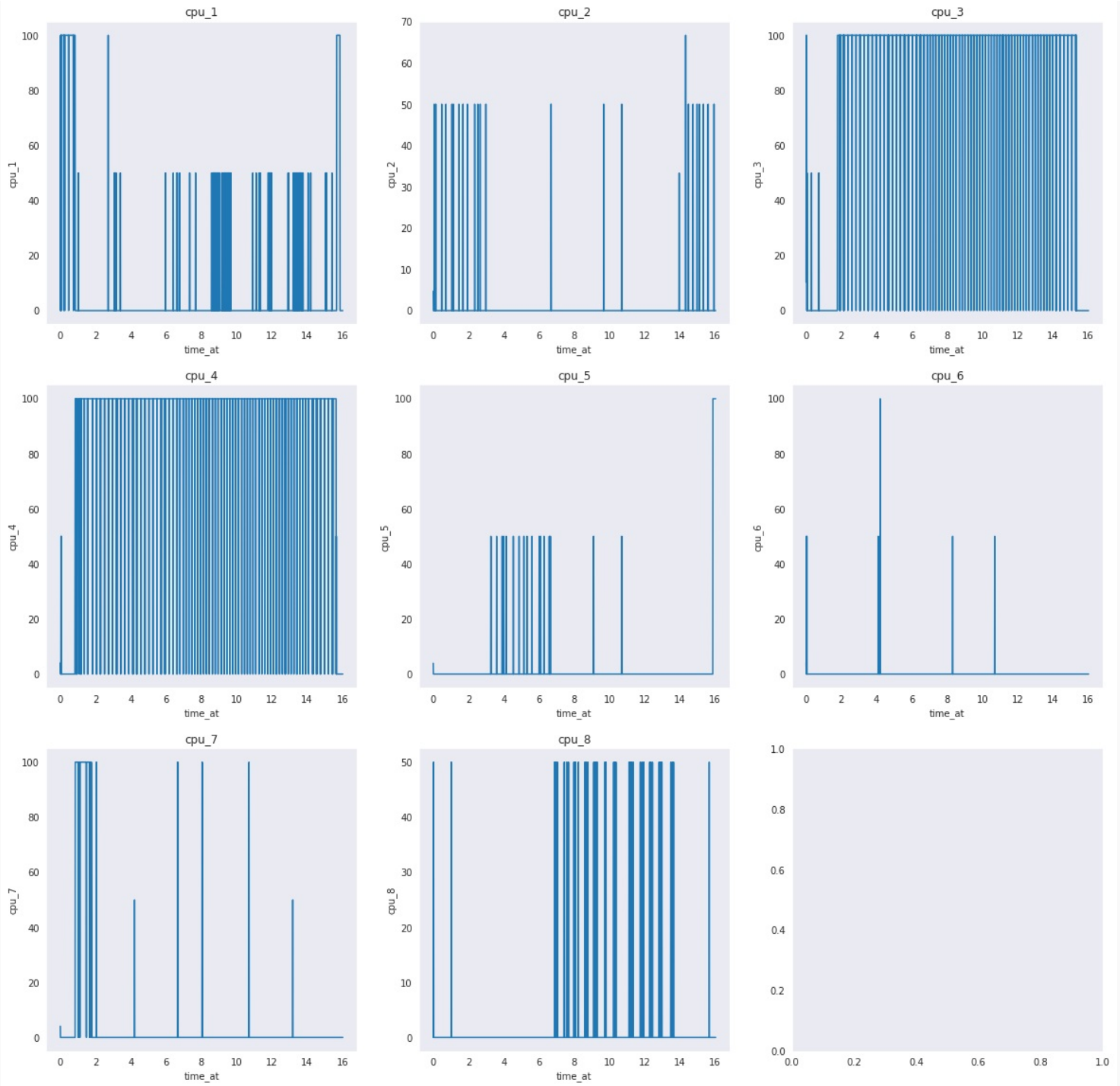
We measure our performance as follows -

We are not interested in how much time we get a solution rather in the same time if a we are able to explore the search space more agressively then we are at profit. So our parameters of algorithms worthiness is -

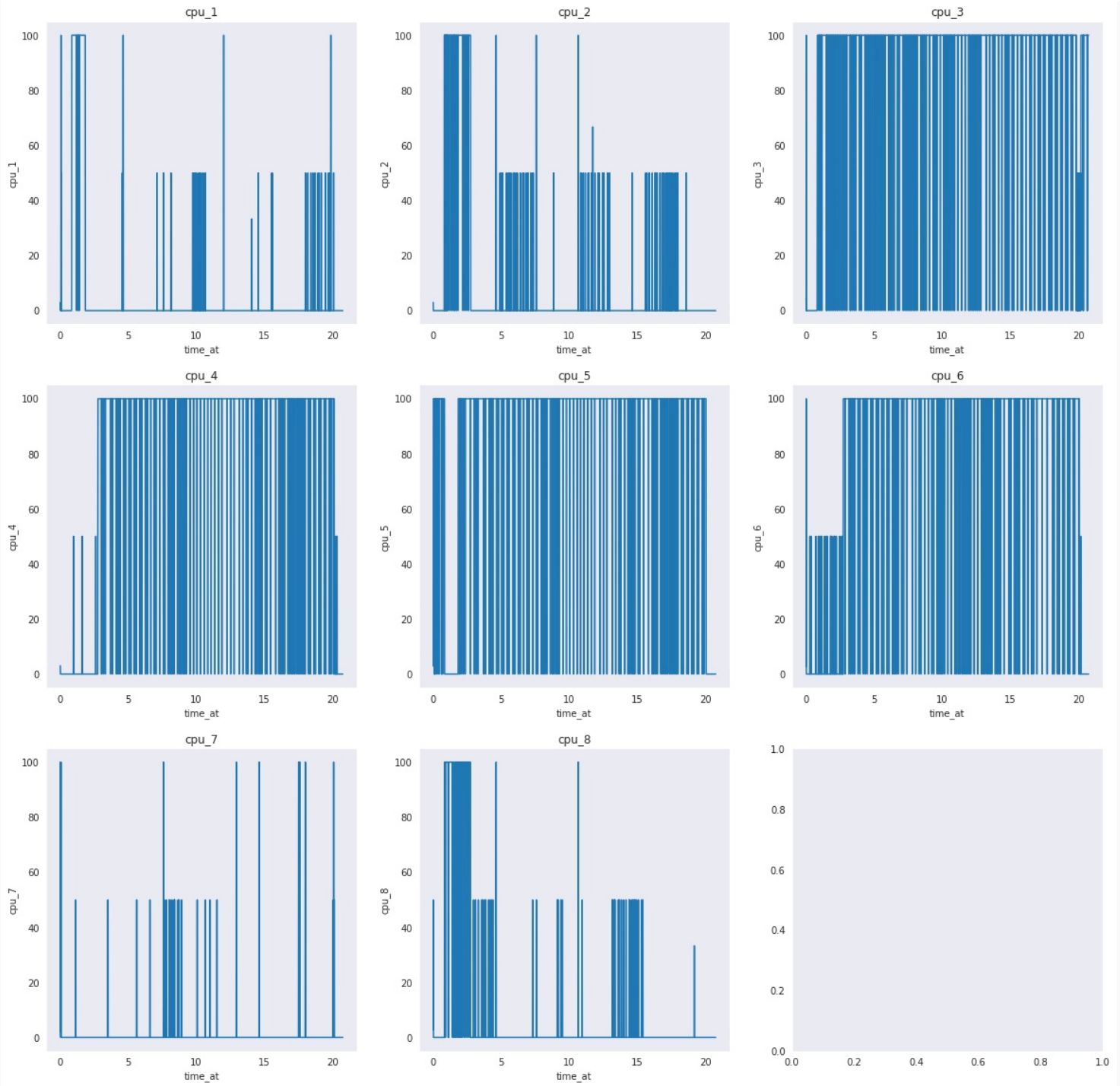
- Per CPU utilization
- Closeness of the output of GSO function to actual Global minima

Per CPU utilisation on ROSENS function

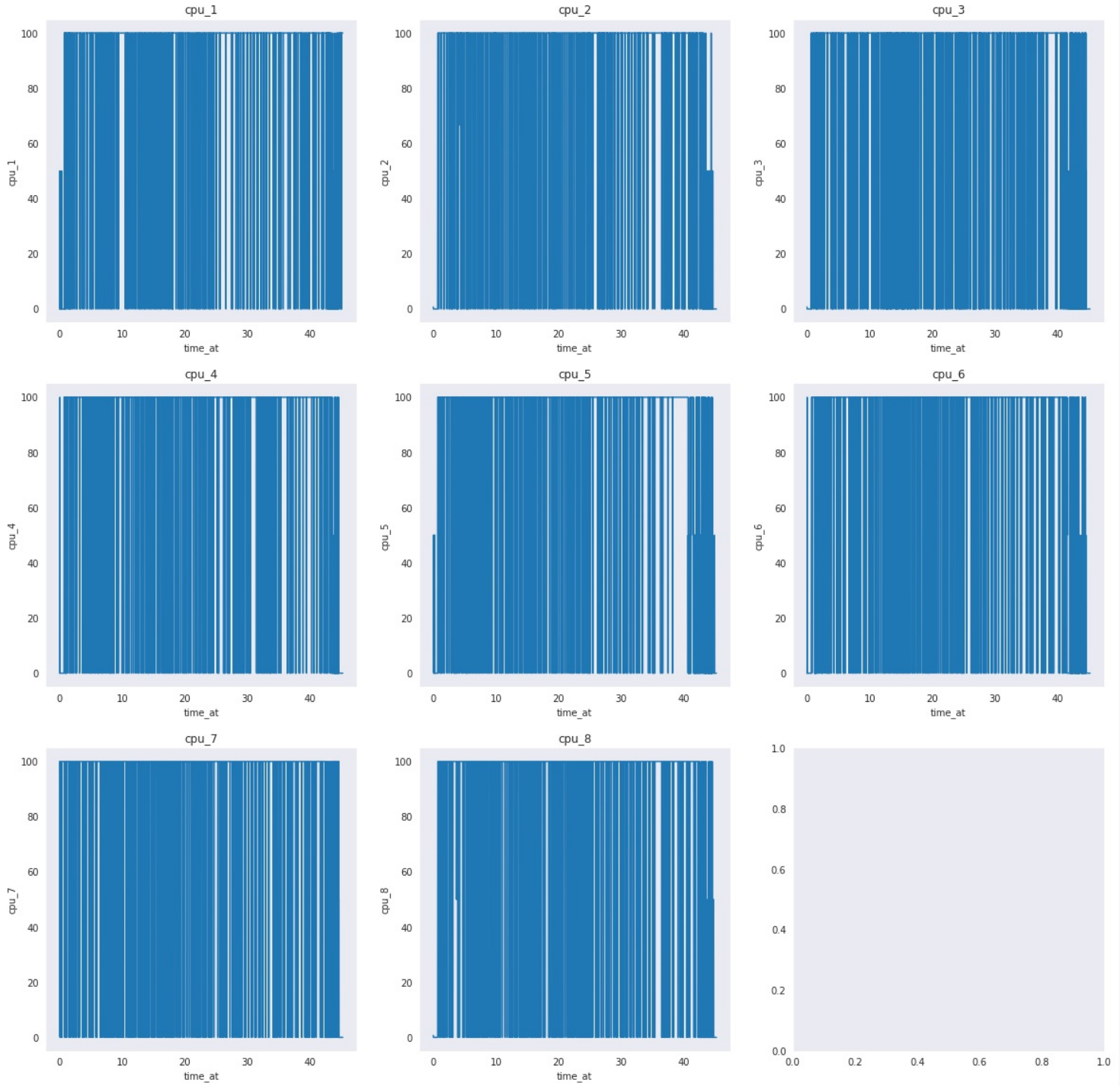
(a) 2 CPUs


cpu_1
cpu_1
time_at
cpu_2
cpu_2
time_at
cpu_3
cpu_3
time_at
cpu_4
cpu_4
time_at
cpu_5
cpu_5
time_at
cpu_6
cpu_6
time_at
cpu_7
cpu_7
time_at
cpu_8
cpu_8
time_at

(b) 4 CPUs



(b) 8 CPUs



Closeness to actual Global Minimas

Function	Actual minima	After SGSO	Error
sphere	[0,0]	[0.999963 0.99992281]	1.7521811060274705e-09
rosen	[1,1]	[1. 1.]	1.262177448353619e-29
rastrigin	[0,0]	[0.99999276 0.99998553]	5.2522347151565965e-11
griewank	[0,0]	[1.00000143 1.00000286]	2.0538605158670134e-12
zakharov	[0,0]	[0.99948174 0.99900123]	2.8253814587149963e-07
nonContinuousRastrigin	[0,0]	[0.99993307 0.99986582]	4.508960414993844e-09

We can clearly see all are errors are less than 10^-7 which shows the success of our algorithm. We can also clearly see that as we increase the number of processors the per-cpu utilization increases. The detailed reports on all the variants of the benchmarks are included in our notebooks also avaiable on github.

Benefits to the community

We all have seen the boon of genetic programming - from Neural Architecture Search [1] (https://arxiv.org/pdf/1704.00764) (AutoML) to automatically design better chassis for cars that humans could not possibly think of[2] (http://boxcar2d.com/) is truly amazing. The Genetic algorithms family comes under the meta-heuristic algorithms.

A comparison was made by (R.Hassan et. al 2004) (https://bee22.com/resources/Hassan%202004.pdf) where they found PSO to be performing way better than GAs in strong settings. Our approach greatly improves the PSO and makes the search more sufficient utilizing hardware at full capacity helping narrow down the search space quickly and therfore argue that our algorithm be put to use in the above areas. We tend to offer this algorithm to the community to bolster the following areas along with the ones mentioned above. As per survey done by (R Poli 2008) (https://www.hindawi.com/journals/jaea/2008/685175/) the areas where our algorithm will benefit are:

- Combinatorial Optimization
- Communication Networks
- Controllers
- Clustering and Classification
- Design
- Finance
- Faults
- Images and Videos

and much more...

We can't wait enough to see what can be done further on with our work and warmly welcome constructive feedback from both the developer and the research community