Parallelizing DIPY model fits with Ray

Ariel Rokem　　Asa Gilmore

1 Under construction

2 Abstract

3 Introduction

Dipy is a popular open-source Python library used for the analysis of diffusion imaging data. It provides tools for preprocessing, reconstruction, and analysis of MRI data. Here we focused on three different reconstruction (XXX need to finish testing of sfm model) models included in Dipy, the constrained spherical deconvolution, free water diffusion tensor and sparse facile models. These reconstruction models, along with several others not tested here are good candidates for parallel computing, as they are independent on the voxel level. While in theory parallelizing these workloads should be a fairly simple task, due to pythons GIL (global interpreter lock), it can prove more difficult in practice. To work around pythons GIL we utilized the library Ray, which is a great system for parallelization of python (https://arxiv.org/abs/1712.05889). In preliminary testing, we looked at three different libraries to accomplish this task, Joblib, Dask, and Ray, but ray quickly proved to be both the most performant, as well as user-friendly and reliable option of the three. Ray's approach to serialization, the process of converting Python objects into a format that can be easily stored and transmitted (XXX improve definition of serialization), also proved to be the least prone to errors for our use case.

(XXX this was written as a word dump, some of this might need to be moved to discussion or methods) Ray is a great system for parallelization (https://arxiv.org/abs/1712.05889).

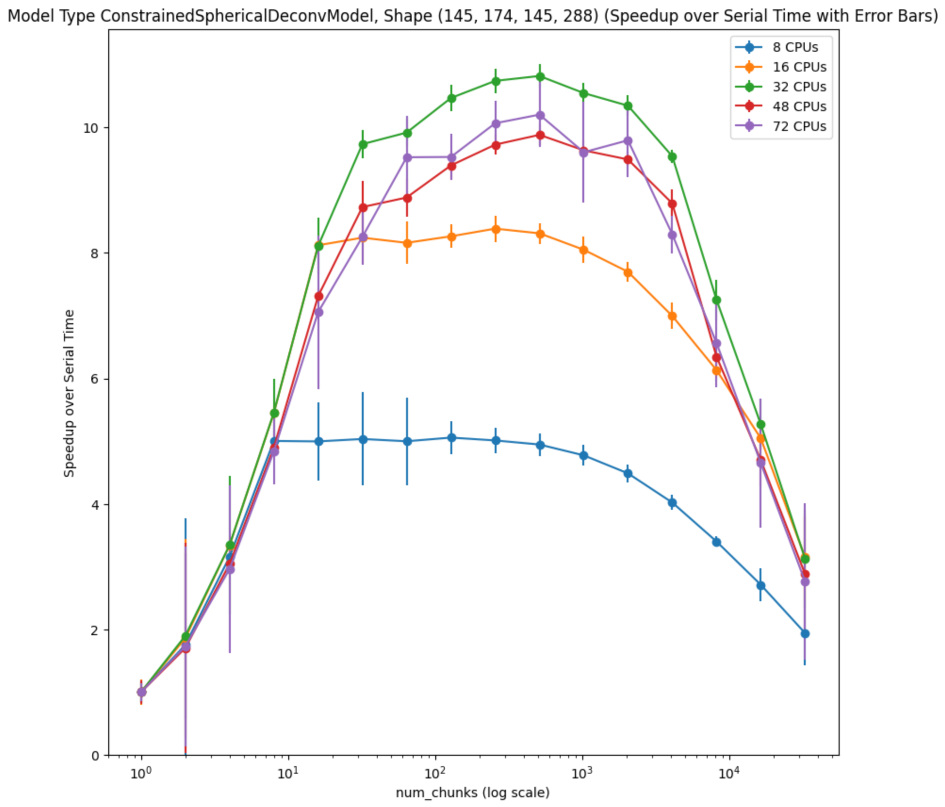
4 Methods

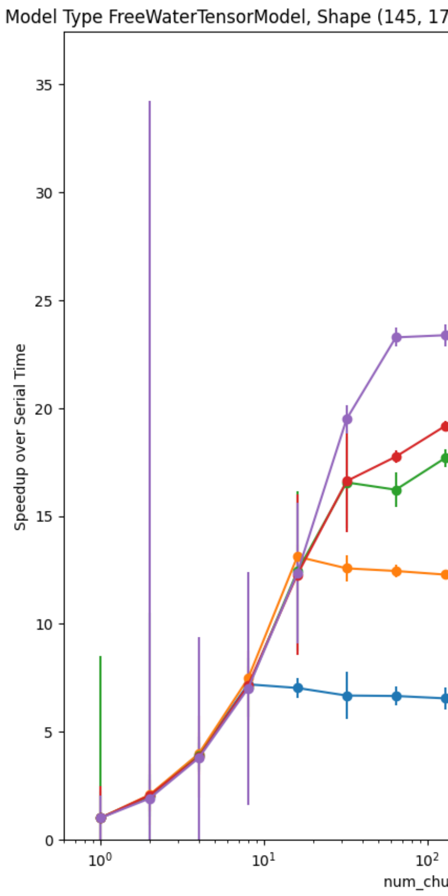
We ran both a constrained spherical deconvolution model and a free water diffusion tensor model through DIPY on a subject from the human connectome project (add more about hcp). We created a docker image to encapsulate the test and allow for easy reproducibility of the tests. The testing program computes each model 5 times for each set of unique parameters. We then iterate across chunk sizes exponentially, from 1-15, where the number of chunks is 2^x (XXX explain better). We ran the tests with the following arguments on docker instances with CPU counts, 8, 16, 32, 48, and 72:

```
--models csdm fwdtim --min_chunks 1 --max_chunks 15 --num_runs 5
```

5 Results

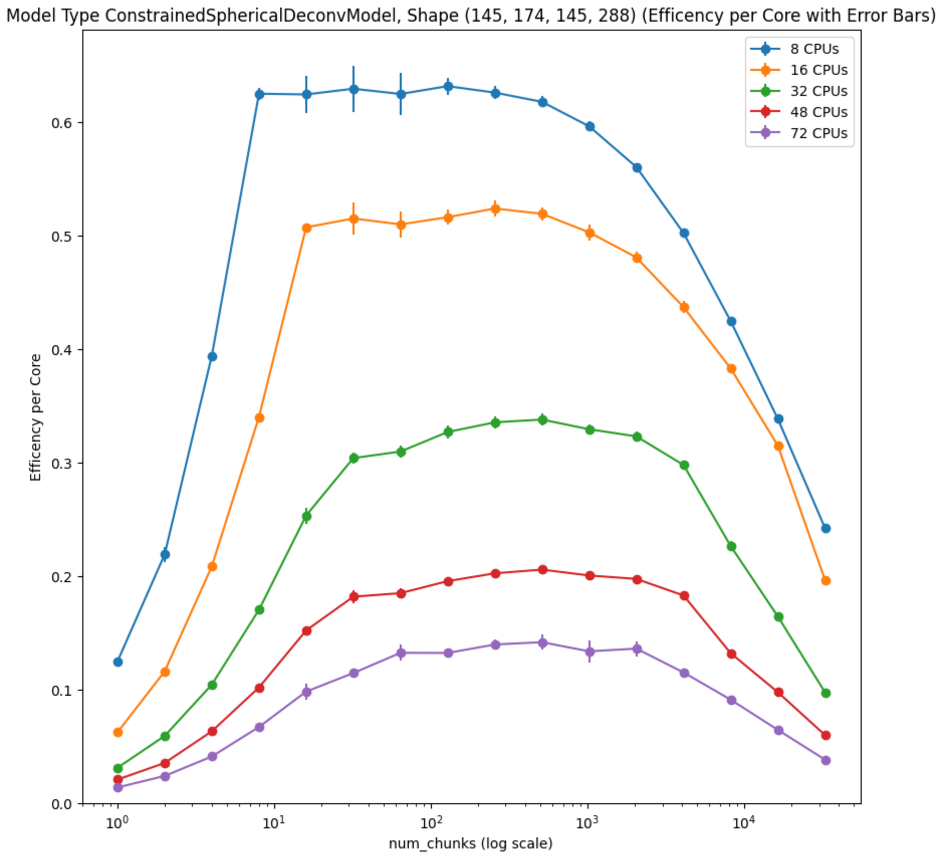
Parallelization with `ray` provided considerable speedups over serial execution for both constrained spherical deconvolution models and free water models. We saw a much greater speedup for the free water model, which is possibly explained by the fact that it is much more computationally expensive per voxel. This would mean that the overhead from parallelizing the model would have a smaller effect on the runtime. Interestingly 48 and 72 core instances performed slightly worse than the 32 core instances on the csdm model, which may indicate that there is some increased overhead for each core, separate from the overhead for each task sent to ray.

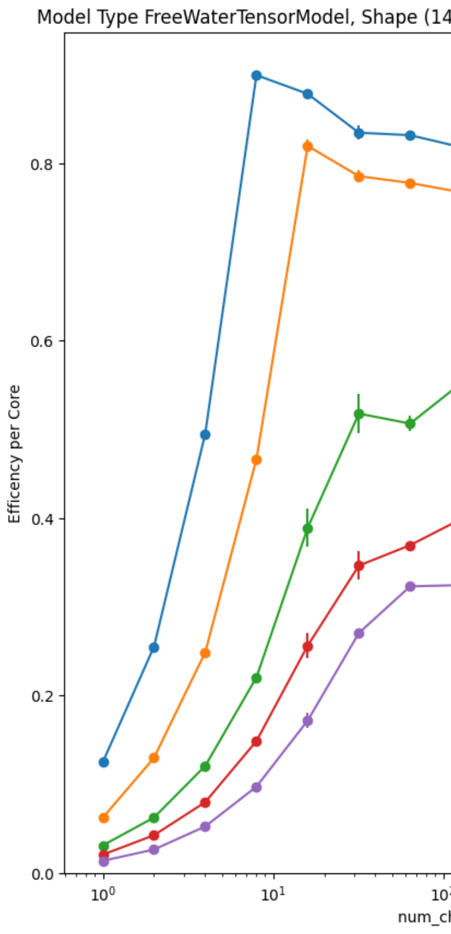




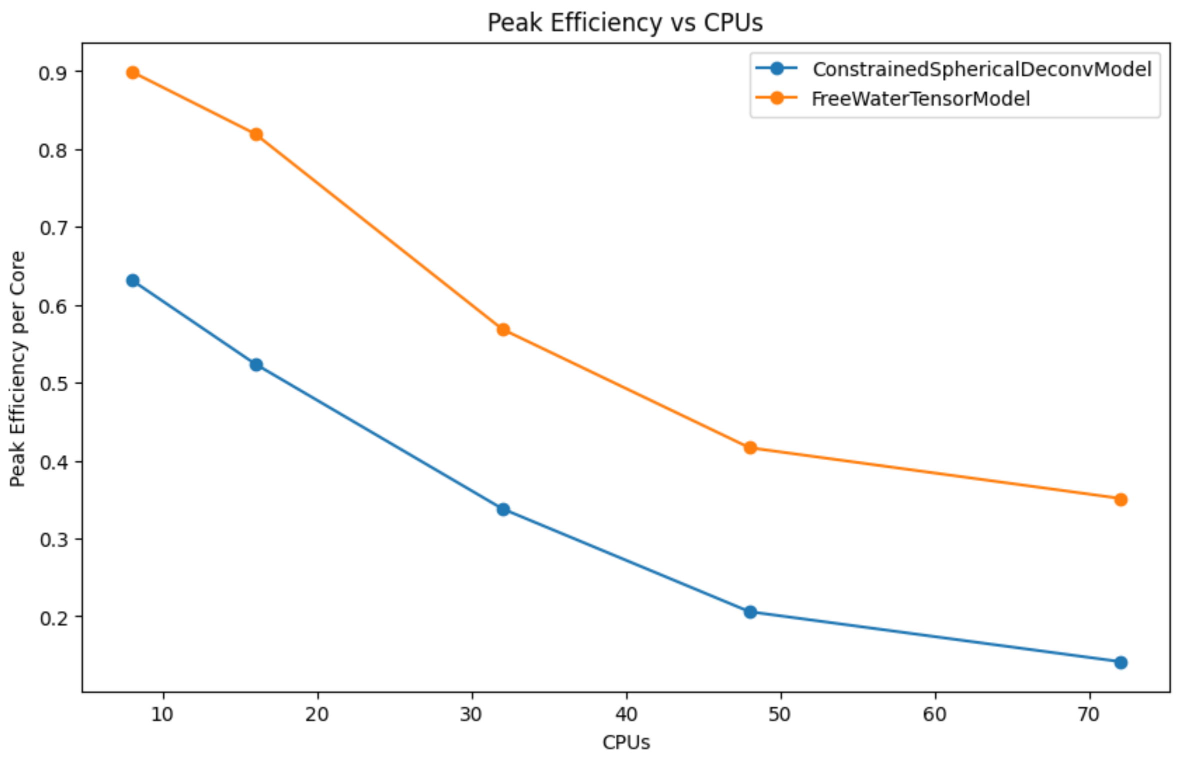
Efficiency decreases as a function of number of CPUs, but is still rather high in many configurations. Efficiency is also considerably higher for the free water tensor model, which is consistent with our expectations given that it is more computationally expensive per voxel and therefore ray overhead would have less effect. The high efficiency of 8 core machines suggests that the most cost-effective configuration for processing may be relatively cheap low core machines.

Ray tends to spill a large amount of data to disk and does not clean up afterward. This can quickly become problematic when running multiple consecutive models. Within just an hour or two of running ray could easily spill over 500gb to disk. We have implemented a fix for this within our model as follows:





We can also look at peak efficiency per core (efficiency at the optimal number of chunks for the given parameters), relative to the number of cores for both models. What's interesting is that we see a very similar relationship between both models, with the fwdti model being higher by almost the same amount for all core counts. This suggests that models such as fwdti that are more computationally expensive per voxel will see better speedups due to the overhead of parallelization being lower relative to the total cost. Interestingly increasing core counts doesn't further increase the benefit of parallelization relative to overhead, which suggests that ray overhead may be very linearly related to the number of cores.

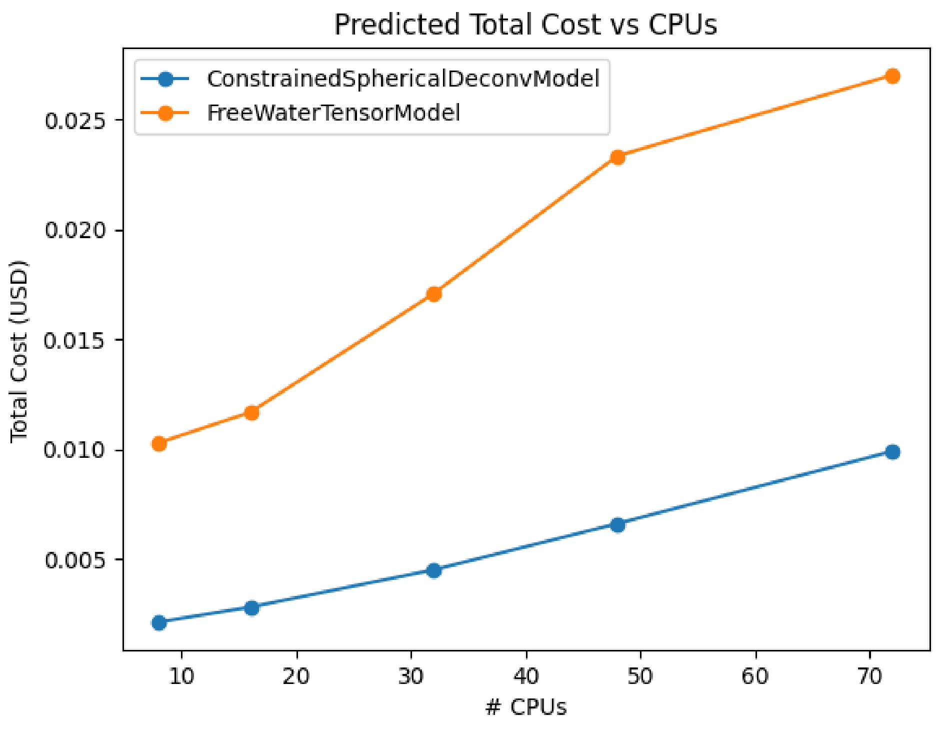


Ray tends to spill a large amount of data to the disk and does not clean up afterward. This can quickly become problematic when running multiple consecutive models. Within just an hour or two of running, Ray could easily spill over 500gb to disk. We have implemented a quick fix for this within our model as follows:

```
    if engine == "ray":
        if not has_ray:
            raise ray()

        if clean_spill:
            tmp_dir = tempfile.TemporaryDirectory()

            if not ray.is_initialized():
                ray.init(_system_config={
                    "object_spilling_config": json.dumps(
                        {"type": "filesystem", "params": {"directory_path":
                         tmp_dir.name}},
                    )
                },)

        func = ray.remote(func)
        results = ray.get([func.remote(ii, *func_args, **func_kwargs)
                          for ii in in_list])

        if clean_spill:
            shutil.rmtree(tmp_dir.name)
```

There seems to be an inverse relationship between the computational cost per voxel and the speedup that you get from parallelization. This is why CSD speedup is maximal for 32 cores.

We have also made a rough approximation of the total cost of computation relative to the number of CPUs. Because all tests were run on a ''c5.18xlarge" machine, and the docker container was simply limited in its access to cores, This approximation makes the following assumptions to estimate the cost of using smaller machines: It assumes that the only differentiating factor between aws c5 machines' performance is the number of CPUs, which may not be true for several reasons, such as total memory available, memory bandwidth, and single-core performance. With this approximation, we see that cost increases as a function of CPUs. This suggests that using the smallest machine that still computes in a reasonable amount of time is likely the best option.



6 Discussion

6.1 Acknowledgments

This work was funded through NIH grant EB027585 (PI: Eleftherios Garyfallidis) and a grant from the Chan Zuckerberg Initiative Essential Open Source Software program (PI: Serge Koudoro).