A performance comparison between centralized versus distributed machine learning algorithms

1st Anton Sakarias Rørbæk Sihm
Department of engineering (Aarhus IHA)
Aarhus, Denmark
au535993@uni.au.dk, StudentID.: 201504954

2nd Peter Marcus Hoveling
Department of engineering (Aarhus IHA)
Aarhus, Denmark
au536878@uni.au.dk, StudentID.: 201508876

***Abstract*—DASK is python framework that enables the use distributed computing. This report compares the centralized machine learning algorithm k-nearest neighbor with an experimental distributed variation. The report analysis critical points in the algorithm from which there is possible gain, and experiments on different approaches to optimize the algorithm through DASK workers orchestrated with Kubernetes. The best yielded result took advantage from a distributed system of 4 DASK workers, mounted storage and CUDA in order to achieve a result 117.9 times better than the initial centralized method.**

I. Introduction

The focus of the this project is to design, implement, test, conduct and document experiments to compare run-time differences between centralized and distributed implementations of a selected machine learning algorithm. The selected algorithm is the k-nearest neighbor algorithm, which focuses on classification and chosen on the basis of its simplicity and comprehensibility. There exits a lot of different ways to solve a classification problem, with some of the newer being deep neural networks and deep convectional neural networks. The focus of this report will be on how to further improve the efficiency of a well know and documented algorithm in terms of previously measured computation time. This report will further investigate different approaches of making k-nearest neighbor run faster using both DASK, CUDA and orchestrating the entire setup using Kubernetes to simulate a distributed manner.

II. Methods

A. K nearest neighbor

K-nearest-neighbor is a classification algorithm, based on the thought that k data-samples numerical value would be close to each other, if they were from the same class [3]. The k is the number of neighbors used in electing the predicted class of a given test sample.

B. DASK

DASK which natively scales Python, promises advanced parallelism enabling performance at scale [2]. In data science the programming language Python has grown to become the dominant language, both in analytics and general programming. With many computational libraries like Pandas, Numpy and Scikit-Learn, DASK tries to scale beyond a single machine. DASK offers not only single and distributed solutions for parallel computations of well known libraries, but provides real-time and responsive dashboards for showing progressions, communication costs, memory-use and more.

By use of DASK, data is divided into either; Dask- Arrays, DataFrames, Bags, Delayed or Futures. From these collections, DASK splits the workload, by delegating tasks to worker nodes and monitoring the entire process. From this a task graph can be created, for better insight in how data is divided, what functions are executed on workers and more.

C. Kubernetes

For simulating distributed servers DASK workers were made to be containerized workloads. For managing these containers, Kubernetes was chosen for its managing capabilities and facilities for declarative configuration and automation [5]. Making the choice of using Kubernetes for the virtualized deployment, allowed for better control of container utilization and the assurance of all worker nodes being equal in computation power.

Hosted in a virtual machine, were both the containerized worker nodes and master. On top of this was Kubernetes orchestrating. As the entire setup was situated on a single virtual machine, only one GPU could be passed through. Kubernetes further allowed the use of all containers sharing one single GPU, by matching labels and selectors to be allowed access to a percentage of GPU power.

Lastly Kubernetes aided in the use of persistent storage and mounting of shared data.

D. CUDA

CUDA which stands for Compute Unified Device Architecture is a parallel computing platform and API, which allows software to load desired computations to the GPU [1]. Giving access to the GPU's virtual instruction set and parallel computational elements. Compared to having programs executed by the CPU, GPU's address the demand for real-time, high resolution 3D graphics and compute-intensive tasks. Allowing for very fast and efficient execution of chosen algorithms.

CUDA was used in extension with Kubernetes. The virtual machine hosting the entire setup had CUDA drivers installed, allowing access to the shared NVIDIA `GTX 2080 SUPER` GPU.

E. RAPIDS

RAPIDS is a collection of open source software libraries and APIs, giving the ability to execute end-to-end data science projects and analytics pipelines entirely on GPUs [4]. RAPIDS utilizes NVIDIA CUDA for low-level compute optimization. Offering GPU parallelism and high-bandwidth memory speed through user friendly Python interfaces. RAPIDS also actively contributes with DASK to allow for GPU-accelerated machine learning.

For the project, RAPIDS offers a variety of multi-node, multi-GPU algorithms, from which the k-nearest-neighbor classifier can be found.

III. Experiments

A. Baseline

Acquiring a baseline on how the default K-nearest-neighbor algorithm performs, the well known dataset MNIST has been used in conjunction with Scikit-Learn. Note that for all experiments the hyper parameter k for K-nearest neighbor is selected to be $k = 8$. As the libraries from Scikit-learn made use of available cores, a further measure of capping Kubernetes container had to be made. A fair comparison from running the algorithm on non-worker nodes, should be equal in processing power. From this, all containerized nodes were allowed at maximum two cores.

Analysing the computation time distribution, as seen in figure 1, training is a lot faster than predictions of the MNIST dataset. Taking up to 99% of the total computation time.

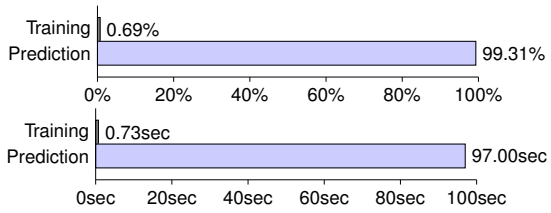

Fig. 1. Heavily weighted one-sided computation distribution

By this it can be concluded that the focus of optimization should be directed at the prediction phase of the algorithm, rather than the training.

B. Distributed using DASK

Using DASK with Kubernetes, three workers are spun up along the baseline worker to form a distributed simulated cluster of four workers. Since concluded from the baseline, that the focus was to not optimize on the K-nearest neighbor training segment, increasing the speed of the prediction segment will be the focal point. This will be done by distributing a chunk of work to each of the four DASK workers, allowing for parallelism.

DASK offers the build of a computation graph, gaining a visual representation of how the distributed computation is going to look. Splitting out the MNIST dataset into four pieces and running the predict segment in parallel, then aggregating the scores into a mean, the following computation graph on figure 2 is produced.

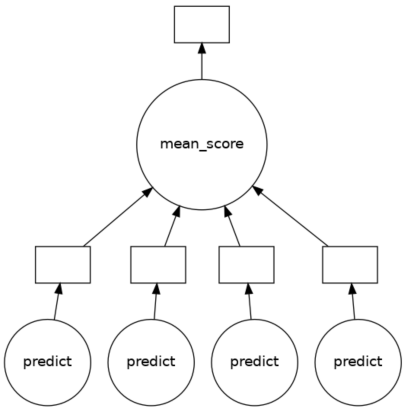

Fig. 2. DASK predict segment computation graph

For each DASK worker the trained model and a 1/4 distinct chunk of the MNIST dataset is needed, in order to execute the prediction. Sending this data back and forth will result in some overhead, as the time is spent on the bandwidth rather than doing computations.

Looking at the performance graph in figure 3 generated by DASK metrics, the task-stream has four different nodes running the prediction segment in parallel.

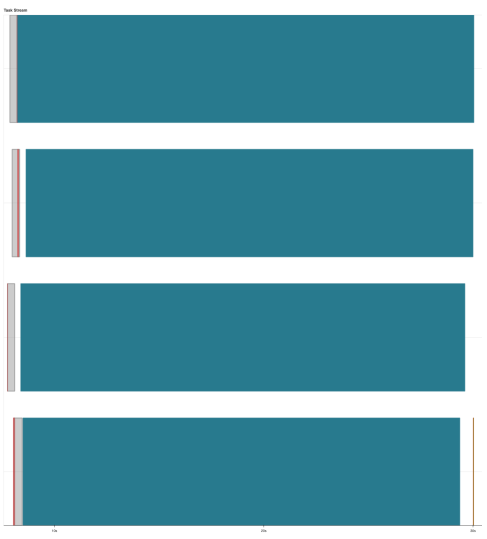

Fig. 3. Dask task-stream showing four nodes working in parallel

Analysing the bandwidth overhead and small overhead from serializing and de-serializing the python functions, figure 4 illustrates the very notable overhead of sending the MNIST data chunks and trained model to each worker.

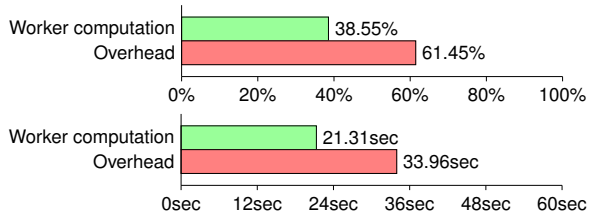

Fig. 4. Computation distribution using DASK

C. Distributed using DASK and Storage

Having a computing cluster of multiple machines, a common strategy to share data is to have physical storage accessible by attachment. Kubernetes offers this by mounting a slice of a physical stored disk into each DASK worker container, allowing the sharing of data between workers.

Repeating the same experiment, but removing data transfer time by storing the trained model and the MNIST data, the work distribution now looks considerably better as seen in figure 5. This is because the overhead now only includes disk read/write speed and the small overhead of serializing and de-serializing the python functions.

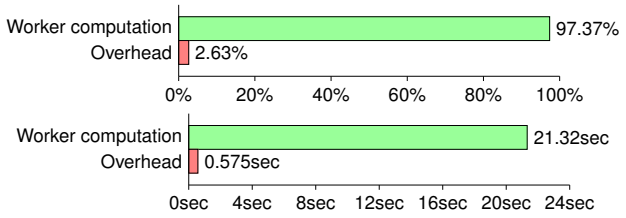

Fig. 5. Computation distribution with persistent storage

D. Distributed using DASK, Storage and GPU

Instead of having each node compute the prediction segment on pure CPU cores, the workers can make use of CUDA and RAPIDS AI library to offload work to the shared GPU.

Looking at the worker computation distribution in figure 6, it might seem like a larger overhead still is at play. However this is not the case, but in fact because the prediction segment is so fast that the small overhead of read/write speed to the GPU and the serializing and de-serializing of python functions, takes up most of the computation time.

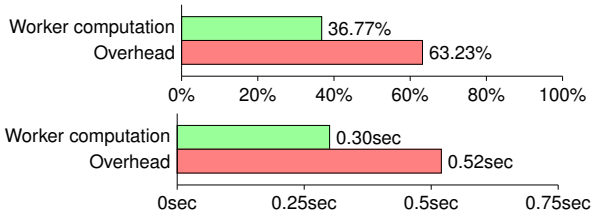

Fig. 6. Computation distribution using GPU

IV. Results

The total computation time including the various overheads for the prediction segment of K-nearest-neighbor can be seen in figure 7 and 8. The baseline yielded a result of 97 seconds with a local computation of 2 CPU's. Moving from the centralized version to a distributed computation using DASK, we achieved a time that was $1,75$ times superior to the baseline. Even though the same strategy had a total of 34 seconds overhead due to sending the model and the chunked MNIST dataset, as seen i figure 8, the distributed algorithm still outperformed the baseline.

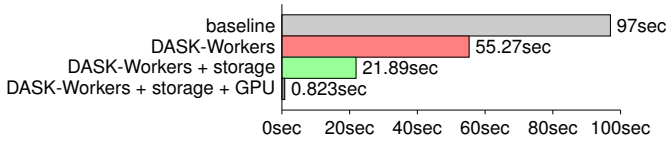

Fig. 7. Computation time comparison

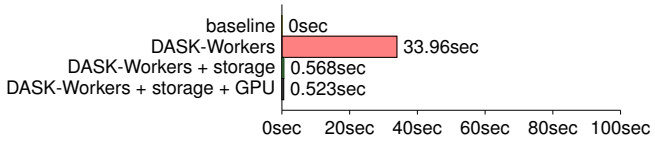

Fig. 8. Overhead comparison

Combating this overhead, shared disk space was introduced. From this the distributed algorithm was able to outperform the baseline by factor of $4,43$. Since the overhead only was 2.63%, the algorithm was able to spend $97,37\%$ of its time on pure computation.

Moving from CPU computations to GPU computations with the help of `RAPIDS AI` and `CUDA`, the k-nearest-neighbor algorithm ended up performing $\underline{117.9}$ times better than the initial baseline.

V. Discussion

Examining the results, the different distributed experiments outperformed the baseline by a large margin, but required understanding of the various overhead problems.

In the case of algorithm selection, the K-nearest-neighbor was parallelizable in the prediction segment, and not nearly as much to gain in the training segment. Applying other different machine learning algorithms might have different computational time distributions, which requires an entirely new rerun of analysis. Some algorithms can also be designed in a way where the possibility for palatalization is not an option.

Since the overhead of sending data over bandwidth hinders optimal performance gains, it is a must to a have a physical attached storage shared between the DASK workers.

The best achieved result was the setup of 4 DASK workers, storage and GPU. The total computation time was 0.823 seconds with a overhead of 0.523 seconds, which entailed an actual GPU time of 0.3 seconds. With this, only $36,45\%$ total computation time was used on GPU work, whilst the rest was overhead from GPU memory read/write and DASK serializing and de-serializing of python functions from the `RAPID AI`.

Distributing machine learning computations is a viable solution, though having the cons of requiring a dedicated Kubernetes cluster running DASK, CUDA and dedicated shared storage. For users valuing a faster machine learning algorithm, the experimented setup is a viable solution. However the setup alone requires time and money investments, which may not always be a feasible solution.

VI. Conclusion

From the results it can be concluded that distributing machine learning algorithms with the aid of CUDA, RAPIDS AI and dask can greatly optimize centralized algorithms with a speed factor of up to 117.9. Sending a lot of data can bottleneck the potential gain of running distributed algorithms, and it is therefor seen as a 'best practise' to make use of mounted shared storage. In the case of selecting the machine learning algorithm K-nearest-neighbor, the focus was set on optimizing the prediction segment, but this however may vary from other algorithms.

VII. Future work

As mentioned the Kubernetes cluster only had 1 GPU which the 4 different workers had to share. This was due to being constrained in hardware resources.

Areas that might be worth investigating in the future:

- Look into cloud solutions for more hardware accelerated learning (Better CPU's and more GPU's instead of GPU share)
- The use of training optimized GPU's such as TPU's
- What is the threshold of optimized splits of work and workers
- Optimizing serialization and de-serialization using compression
- Look into other machine learning algorithms.
- Training of deep neural networks using Dask and CUDA + Kubernetes

References

[1] Wikipedia on CUDA, what is CUDA?, (accessed on 18/12/2021), https://en.wikipedia.org/wiki/CUDA

[2] Dask website, DASK, (accessed on 18/12/2021), https://dask.org/.

[3] Wikipedia on machine learning algorithms, k-nearest-neighbor, (accessed on 18/12/2021), https://da.wikipedia.org/wiki/K-n%C3%A6rmeste_naboer

[4] Information about RAPIDS, RAPIDS, (accessed on 18/12/2021), https://rapids.ai/about.html

[5] Kubernetes Dashboard, What is Kubernetes?, (accessed on 19/12/2021), https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/