View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by UPCommons. Portal del coneixement obert de la UPC

Accelerating Hyperparameter Optimisation with PyCOMPSs

Albert Njoroge Kahira
Barcelona Supercomputing Center
Barcelona, Spain
albert.kahira@bsc.es

Leonardo Bautista Gomez
Barcelona Supercomputing Center
Barcelona, Spain
leonardo.bautista@bsc.es

Javier Conejero
Barcelona Supercomputing Center
Barcelona, Spain
javier.conejero@bsc.es

Rosa M Badia
Barcelona Supercomputing Center
Barcelona, Spain
rosa.m.badia@bsc.es

ABSTRACT

Machine Learning applications now span across multiple domains due to the increase in computational power of modern systems. There has been a recent surge in Machine Learning applications in High Performance Computing (HPC) in an attempt to speed up training. However, besides training, hyperparameters optimisation(HPO) is one of the most time consuming and resource intensive parts in a Machine Learning Workflow. Numerous algorithms and tools exist to accelerate the process of finding the right parameters for a model. Most of these tools do not utilize the parallelism provided by modern systems and are serial or limited to a single node. The few ones that are offer distributed execution require a serious amount of programming effort.

There is, therefore, a need for a tool/scheme that can scale and leverage HPC infrastructures such as supercomputers, with minimum programmers effort and little or no overhead in performance. We present a HPO scheme built on top of PyCOMPSs, a programming model and runtime which aims to ease the development of parallel applications for distributed infrastructures. We show that PyCOMPSs is a powerful framework that can accelerate the process of Hyperparameter Optimisation across multiple devices and computing units. We also show that PyCOMPSs provides easy programmability, seamless distribution and scalability, key features missing in existing tools. Furthermore, we perform a detailed performance analysis showing different configurations to demonstrate the effectiveness our approach.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**; **Machine learning**.

KEYWORDS

Machine Learning, Hyperparameter Optimisation, Distributed Computing, HPC, PyCOMPSs

1 INTRODUCTION

In the last decade or so, Machine Learning(ML), especially Deep Learning (DL), has provided impressive results in different applications and tasks such as image classification [11], speech recognition [20] and autonomous driving [6]. This can be attributed to extensive research in algorithms, model architectures and advancements in hardware which provides massive parallelism such as Graphic Processing Units (GPUs) and dedicated hardware for neural networks such as Tensor Processing Units (TPUs), that allow us to train very large and complex models in a reasonable amount of time. As a result, Deep Learning [12] has now become the go to method for applications with big datasets. Interestingly, the spectrum of these applications continues to grow.

To facilitate and accelerate both research and application of ML, numerous tools and libraries have been developed. ML frameworks such as Tensorflow [2], PyTorch [17] and Caffe [8] enable developers to train and deploy complex models. Most of these advanced frameworks focus on training and deployment. However, before training, two important things have to be decided. The architecture of the model and the configuration of the model. Arriving at the right model for a specific dataset is a complex process, that does not only require skilled engineers but is also time consuming and compute intensive. Interestingly though, the time consumed to train the model as well as the effectiveness (accuracy) of the model heavily depends on a set of parameters selected prior the training procedure. This set of parameters is called hyperparameters.

The process of finding the correct combination of parameters for a certain model is called Hyperparameter Optimisation (HPO), sometimes refereed to as Hyperparameter Tuning (HPT). It is one of the key parts in a machine learning workflow. The most common hyperparameters include number of epochs, batch size, learning rate, optimiser, and sometimes specific model parameters such as number of layers. Finding the correct combination of these parameters is non trivial and manual tuning is both difficult and sometimes impossible as the best solution is not always obvious. Furthermore, in most cases HPO takes the longest time as it is not only computationally intensive but also involves multiple trainings. As such, there has been significant research interest in HPO. Research in this domain has generally taken two paths, 1) Algorithms for HPO 2) Tools to implement and execute these algorithms. Further details of both are discussed in section 2, but our focus is on the latter.

Numerous tools for HPO have been developed as is evident in section 2. An in-depth look into most of these tools reveals 3 major

The final publication is available at ACM via http://dx.doi.org/10.1145/3339186.3339200

issues that this paper will address. First, most existing HPO tools are sequential. Those that are parallel constrain the user to a single node and those that span multiple nodes involve complex cluster configuration. Considering that most, if not all, HPO algorithms are embarrassingly parallel, these processes can be significantly accelerated by exploiting both parallelism and distributed execution. Second, recent trends in DL show an increase in the size of models. This not only translates to an increase in the number of hyperparameters(to magnitudes of hundreds) but also requires skillful partitioning and usage of available computing resources as one model can span across several devices and take training time in magnitude of days. Long execution times also raises the important question of fault tolerance.

Third, there has been a proliferation of Machine Learning applications in High Performance Computing (HPC). However, not much has changed in ML workflows, especially for HPO. Since HPO is an integral part of the ML workflow, there is a need for a HPO scheme and tool that can leverage the full power of HPC such as very high inter-node communication, HPC file systems, heterogeneous computing and scalability. However such a tool should not come with a steep learning curve or increased overhead in programmability. A study of current trends and existing literature reveals that an ideal HPO tool should therefore have the following characteristics.

- Parallel : Intra-node task parallelization
- Distributed : Distribute tasks across multiple nodes
- Scalable : Speed up as the number of nodes increases
- Robust : Guarantee a certain degree of fault tolerance
- Framework agnostic: It should not be constrained to a specific framework
- It should also provide essential features such as early stopping and visualisation dashboards to enable researchers make sense of the output.

Taking into consideration the above mentioned, this paper presents a robust HPO scheme, built on top of PyCOMPSs, to accelerate HPO. PyCOMPSs [21] is the Python binding of COMPSs, a programming model and runtime which aims to ease the development of parallel applications for distributed infrastructures, such as Clusters and Clouds. A detailed description of PyCOMPSs is provided in section 3. The contributions of this paper can be listed as follows.

- We present a robust scheme for HPO in HPC clusters and grid with minimum changes to the code.
- We implement grid search and random search using PyCOMPSs to demonstrate the usage.
- We present an alternative tool for both HPO and other ML workloads that are embarrassingly parallel.

The remainder of this paper is divided as follows, section 2 gives a look into existing tools and previous work on HPO, section 3 introduces the PyCOMPSs framework, in section 4, we explain how to implement HPO using PyCOMPSs, then we provide details of the experiments performed in section 5 and discuss the results in section 6 . Finally section 7 gives a conclusion and future work.

2 BACKGROUND AND EXISTING TOOLS

In this section, we give a brief background of ML and HPO. We then review and discuss existing tools for HPO. The list covered is by no means exhaustive but every attempt has been made to cover the most popular tools.

2.1 Background

Even though the idea of a machine capable of learning and mimicking human intelligence was proposed in the early 1950s, its only recently that we have seen significant progress and commendable results. One factor for this is the invention or Artificial Neural Networks (ANNs) and back-propagation, the algorithm used to train these ANNs. The other factor is a major surge in the amount of data available. These combined with increased computing power have made Deep Learning a major research research topic in Computer Science. A subset of this research has been tools and algorithms for HPO.

In algorithms for HPO, the most popular ones are Exhaustive Grid Search and Random Search. Exhaustive Grid search involves trying out all possible combinations and comparing the result using a metric such as loss or accuracy. This approach is feasible when there is a small set of hyperparameters. However, it becomes impossible and unrealistic with a larger search space. Random search [5] was proposed by Bergestra et al and has become more common. Rather than search through the entire search space, combinations of parameters are picked randomly. Empirical results show that random research is more efficient than grid search and arrives at parameters that are good or better at a fraction of the time required by grid search.

Though random search is a superior algorithm in many cases, several other approaches have been proposed. Gaussian Process and Tree-structured Parzen Estimator were proposed by Bergstra et al [4] for Deep Belief Networks. Bayseian optimisation is another approach that essentially builds a surrogate model to approximate the ideal trained model by using different hyperparameters. It's practical usage and implementation is presented by Snoek et al [19] .The tools discussed below implement one or several of these algorithms.

2.2 Existing Tools

Madrigal et al [15] did a review of existing tools HPO using a computer vision application. They analysed and compared 4 tools for multiple object tracking applications: MCMC, SMAC, TPE and Spearmint. These tools were analysed in terms of stability, performance and usability with the goal of helping making informed decisions when choosing a tool and method for HPO. We discuss other tools not mentioned in that work.

Scikit-learn [18] is perhaps one of the most popular machine learning tools. It combines many of the state of the art algorithms in an easy to use way. Scikit-learn provides both exhaustive grid search and randomized parameter optimisation and uses cross validation to evaluate the best performing parameters. Furthermore scikit-learn computations can can also be run in parallel by setting the number of jobs. However, scikit-learn does not provide multi-node support and is not efficient for complex tasks such as deep learning.

Sherpa [7] is a hyper-parameter optimisation tool geared towards HPO for computationally expensive tasks such as deep learning. It includes several HPO algorithms such as random search, grid search, Bayesian optimisation and local search. Even though Sherpa is intended to run in a multi node environment, doing so requires scheduler and mongoDB. Besides the extra overhead introduced by MongoDB, scheduler configuration is a complex task in HPC.

Shadho [9] developed by Kinnison et al is a general purpose massively scalable hardware-aware distributed hyperparameter optimisation tool. It ranks models using two heuristics, complexity and priority. Models are then ranked and assigned resources accordingly.

Hyperopt [3] by Bergstra et al is another tool for serial and parallel HPO over awkward search spaces. It includes random search and Tree of Parzen Estimators (TPE) algorithms. Like Sherpa, HyperOpt also requires mongoDB for parallel execution.

Kopt and Tolos are HPO tools specifically built for Keras. Kopt is based on Hyperopt and requires mongoDB to parallelise on multiple workers. Both are constrained to specific frameworks.

Tune [14] is a unified framework for model selection that allows straightforward scaling in large clusters. Each training is referred to as a trial and an experiment is a collection of trials. Tune is built on top of Ray [16] framework

Google Cloud Machine Learning Engine is part of the larger family of Google products for machine learning. However, the product is heavily dependent on Google infrastructure and is neither open source nor free.

As we shall show in the following sections, PyCOMPSs not only enables the design of more complex workflows with little programming effort, it also handles job management, data transfers dependencies and reuse of memory objects from one task to the next if they use the same object. These key features are not only missing from existing tools, but implementing them in existing job schedulers such as slurm requires multiple reservations and a serious developers effort.

3 PYCOMPSS

PyCOMPSs [21] is a task based programming model that enables the parallel execution of existing Python sequential applications, with minimal impact on the development effort, in distributed environments. To do this it offers an interface for parallelizing that uses Python decorators to identify the methods to be considered as tasks, and a small API for synchronization. Formally, PyCOMPSs is the Python binding of COMPSs [1] (Figure 1), which relies on the COMPSs runtime and communicates with it whenever a task is detected or its execution requested.

In order to enable the parallelization, the runtime builds a data dependency graph of the tasks that make up the application at execution time. To this end, the task parameters and its direction are taken into account to determine the dependencies among tasks. The runtime is responsible of keeping track of the tasks and respect the dependencies in order to guarantee the validity of the execution, that is, to produce the same result as if is executed sequentially. Consequently, the runtime that can exploit the inherent parallelism of the application at task level and can execute the application in a distributed environment, such as grids, clusters, clouds, and container managed clusters. To achieve this, the runtime is able to schedule the tasks in the available computational resources, acting as an interface with the different computing resources, and

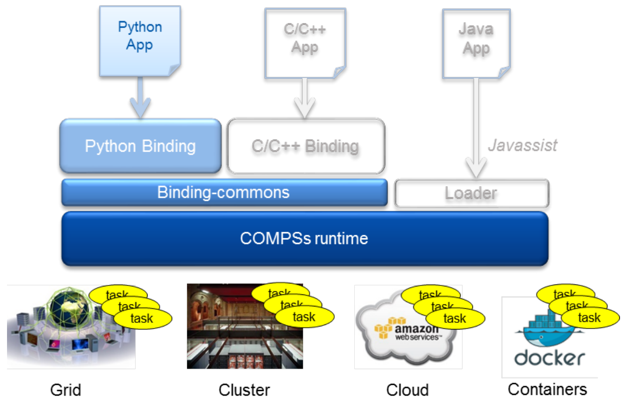


Figure 1: COMPSs Architecture

transferring the data when needed. COMPSs also supports Java and C++ applications.

The mechanism that PyCOMPSs provides to declare a method as a task is the *@task* decorator, which can be used over any function, instance method or class method. In this decorator, hints to specify characteristics of the function parameters or hints for the scheduler can be included. For example, the *returns* keyword into the *@task* decorator can be used to specify the type/number of return elements, the name of a parameter with its type (e.g., FILE) or their directionality (e.g., IN, OUT, INOUT), and *priority=True* for the scheduler so that it tries to schedule that task as soon as possible, among others.

Moreover, PyCOMPSs also provides a set of decorators which can be placed on top of *@task* in order to: define task constraints *@constraint*, define the task as an external binary, MPI or OmpSs executable *@binary, @mpi, @ompss respectively*, declare multiple implementations for the same task *@implement* (this decorator allows the runtime to choose the most appropriate task considering the resources), nesting tasks *@compss*, or even multi node tasks *@multinode*.

Listing 2 shows an example of the *experiment* task, which receives an IN parameter (*config* - since its direction is not explicitly defined, default is taken) and returns a single integer value. In addition, a constraint has been defined, declaring that the task requires one core and one GPU.In a nutshell, key strengths of PyCOMPSs that empower its usage for HPO, include the following:

Programmability: PyCOMPSs follows the natural Python way of programming and all a user has to do is add decorators to existing code. Furthermore, in the absence of PyCOMPSs, the program executes sequentially as it would and all PyCOMPSs directions are ignored. This is particularly important because new users can get up and running within no time hence encouraging adaptability.

Seamlessly Distributed: Cluster management and distributed computing can be a complex task that adds a significant overhead to machine learning researchers. PyCOMPSs takes the burden of cluster configuration from the researcher. For instance, if the user wants to use multiple nodes for HPO, they only need to set the

[1]compss.bsc.es

number of nodes for the entire job and PyCOMPSs seamlessly manages this resources and allocates tasks based on the requirements of each task.

Resource Management: PyCOMPSs manages all available resources accordingly. The user can exclusively determine the type and number of computing resources for a particular task. Furthermore and very important for machine learning, PyCOMPSs supports heterogeneous resources. As such, for compute intensive deep learning applications, each task can will be assigned a number of CPUs and a GPU. If further, a task has built-in parallelism, PyCOMPSs will not interfere with this. PyCOMPSs also enforces CPU and GPU affinity and therefore prevents tasks from competing for same resources.

Fault Tolerance: A sequential application has a single point of failure. Depending on where the failure happens, this could be a wastage of both time and resources. For long running applications such as HPO, its important to ensure continuity in case of failure. Fault tolerance in PyCOMPSs is supported in two ways. If a task fails for whatever reason, an attempt is made to start the task again. Secondly if a computing unit fails or becomes unavailable for whatever reason, PyCOMPSs restarts this task in another computing unit. This is especially important in machine learning where some tasks are bound to fail during execution due to long execution times.

4 APPROACH

In this section present our approach to implement HPO using PyCOMPSs. First we explain how to structure the application, then give programming details and finally we explain what PyCOMPSs does behind the scenes to distribute the application.

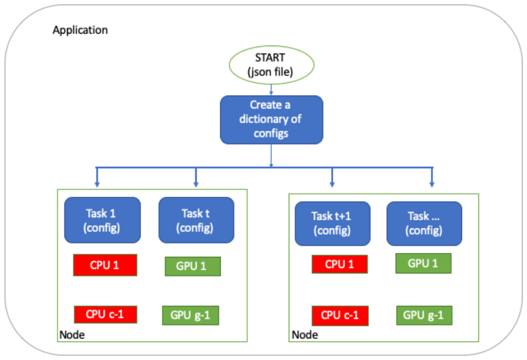


Figure 2: Application Structure

As mentioned earlier, HPO is essentially running multiple trainings with different configurations to determine the one that generalizes best. Training doesn't have to run all the way to the end as one can tell how a model is training after several epochs/iterations. This multiple runs are generally independent of each other. Traditionally, one would just launch one training after the other and make observations. If they have several computers available, one could launch multiple trainings on different computers. We build our HPO tool based on these principles.

On the general structure, at the very top we have an ***application***, which is the entire HPO process. A JSON file containing all the hyperparameters and their values is passed to this application at start. Training and observing a model is an experiment and can be defined as a ***task*** in PyCOMPSs terms. The application will therefore be made up of multiple tasks that will be executed either on the same node or across multiple nodes. Each task requires a unique set of hyperparameters, we call this ***config***, that is passed to the task as a parameter. This configs are generated, depending on the algorithm selected, from the list of hyperparameters contained in the JSON file that was passed to the application. A sample config file is shown in Listing 1. A high level overview of the structure and flow is shown in figure 2.

```
{
     "optimizer": ["Adam" ,"SGD", "RMSprop"]
    ,
     "num_epochs": [20, 50, 100],
     "batch_size": [32, 64, 128]
 }
```

Listing 1: A simple config file

From the structure show in Figure 2 programming entails very few changes to existing sequential code. To make each experiment a task, executable in parallel with other tasks, we simply add the ***@task*** decorator from the PyCOMPSs API before the method/function. We also use the ***@constraint*** decorator to assign the type and number of computing resources to each task, e.g CPU or GPU. We then launch the tasks in a loop passing a different config to each task. The code in Listing 2 summarises how to implement HPO in PyCOMPSs.

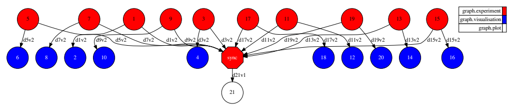


Figure 3: Tasks graph

When we launch this application using using PyCOMPSs (to launch we use ***runcompss application.py json_file***), a dynamic graph is created and all dependencies are established. A sample graph for one of the experiments is shown in Figure 3. PyCOMPSs then assigns computational resources based on the requirements for each task. When not using a Parallel File System (PFS) such as IBM's General Parallel File System (IBM GPFS) then the data required by the task is copied to the specif node that the task will be executed. Otherwise all tasks can read and write to the PFS. Its important to note that most HPC clusters are equipped with PFS.If no further resources are available, tasks wait for the resources.

Tasks are then executed in workers independently and in parallel to completion. A task can utilise its own internal parallelism if its designed to do so, for instance, Tensorflow executes tensor operations in parallel.In case a task fails for whatever reason (such as node failure), the runtime tries to start the same task in the same node, if it fails again, its restarted in another node. This way, PyCOMPSs ensures fault tolerance. The failure of task does not affect the other tasks unless there are some dependencies.

```python

# PyCOMPSs moduleslsl
from pycompss.api.task import task
from pycompss.api.api import compss_wait_on
from pycompss.api.constraint import constraint

def create_model(config):
    # New model created every time with different parameters
    # Model parameters can be set here from the config file (i.e optimisers)
    ...
    ...
    return model

@constraint(processors=[{'ProcessorType':'CPU', 'ComputingUnits':1}, {'ProcessorType':'GPU', 'ComputingUnits':1}])
@task(returns=int)
def experiment(config):
    # Trainign parameters can be set here (ie No of Epochs)
    model = create_model(config)
    model.train(config)
    return val_acc

def main():
    args = get_args()
    configurations = process_config(args.config)

    for config in configurations:
        experiment_result = experiment(config)
        results.append(experiment_result)
     results = compss_wait_on(results)

if __name__ == '__main__':
    main()
```

Listing 2: Implementing HPO with PyCOMPSs

On completion, each task returns the result which can be a performance measure such as validation loss or accuracy and training history. For immediate and interactive action, the performance measure returned can be visualised using another task. When all tasks are completed, we plot the graphs showing the performance of each experiment. To do this, we use the ***comps_wait_on*** over the list of results of all the experiments that synchronizes and ensures all results are available for plotting.

5 EXPERIMENTS

To demonstrate the usage and effectiveness of our scheme, we designed and performed several experiments using popular machine learning benchmarks , MNIST [13] and CIFAR 10 [10]. The experiments are performed at the MareNostrum 4 supercomputer. Each node has two Intel Xeon Platinum chips, each with 24 processors, a total of 48 per node. For GPU implementations, we perform experiments on both MinoTauro cluster, which has 2 K80 NVIDIA GPU Cards and 2 Intel Xeon E52630 v3 (Haswell) 8-core processors and CTE IBM POWER9 cluster which has 2 x IBM Power9 8335-GTH @ 2.4GHz (3.0GHz on turbo, 20 cores and 4 threads/core, total 160 threads per node and 4 x GPU NVIDIA V100 (Volta) with 16GB HBM2.

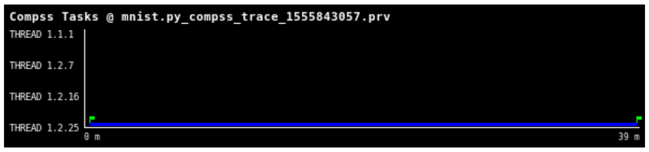


Figure 4: Running a single task on a single core

When tracing is set (this is done using a simple flag), PyCOMPSs generates a set of traces that help in application analysis. This is because PyCOMPSs is instrumented with Extrae, an instrumentation package that captures information during the program execution and generates paraver traces. Paraver [1] is a powerful tool that provides detailed quantitative analysis of program performance. We run the first set of experiments with tracing set. From the traces generated, X axis is the time while Y axis is the resource (i.e cores and nodes).

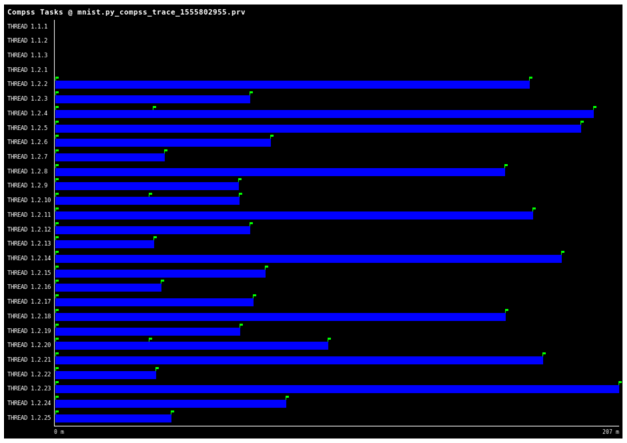
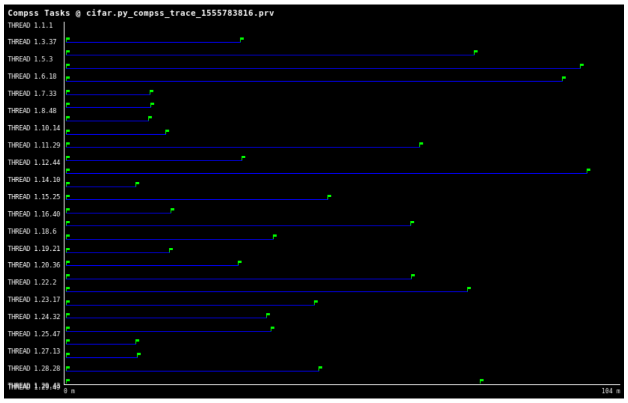


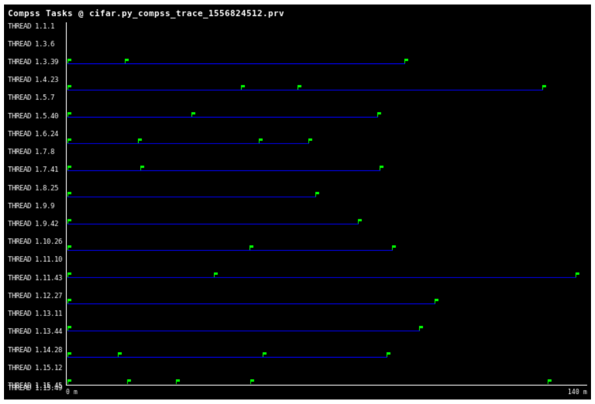
Figure 5: Multiple tasks on a single Node

The first experiment is to make sure and show that each task respects the resources given and CPU affinity is enforced. For this, we launch just one task and assign one core in a node with 48 cores. This is done using the MNIST data set as it is not a very compute intensive task. The traces are shown in the Figure 4. The task takes around 29 mins to run to completion and its constrained to a single core. Even though tensorflow's default behavior is to span across all available resources, PyCOMPSs is able to enforce CPU affinity and the application has access to only the resources allocated.

The next experiment is to see how tasks are distributed across cores in a single node. For this we launch the full MNIST HPO experiment using grid search. From the configuration file, 27 different experiments are created. The search space is number of epochs, batch size the optimiser, 3 parameters for each. Since the worker takes half of the cores in a node, 24 cores are left for the tasks. As such, not all tasks will run in parallel. However, the next task is assigned a computational unit as soon as one is available as shown by the event flags. The traces are shown in Figure 5.



(a) 28 Nodes



(b) 14 Nodes

Figure 6: Multiple tasks on multiple nodes

The final experiment is to demonstrate HPO across multiple nodes. For this we choose a much bigger dataset, CIFAR10. A total of 27 experiments are created to be distributed across 27 nodes. However, during job submission, we request an extra node for the worker to make sure that all the tasks run in parallel. We assign 48 cores to each task (the total number of cores in a node) and let Tensorflow take care of internal parallelism. We also repeat this experiment with half the number of nodes. The traces are shown in the Figure 6.

Both tracing and graph generation create a performance overhead. These two features can easily be turned off by a simple flag when launching the application. Ideally we do not need the traces in HPO, but they are important to show deeper details of the application. As such, we repeat the second and last experiment with traces turned off. We also execute the same experiment on the GPU cluster, as ideally, training is done on GPU. We also repeat the experiments with different GPU and CPU configurations. In these experiments we only measure the execution time. Results for those executions are presented in Section 6.

6 RESULTS AND DISCUSSION

In this section we provide detailed analysis of our tool and the results of the experiments. We first do a performance analysis and resource utilisation by looking at the traces generated. The objective is to show the effectiveness of PyCOMPSs in task parallelism and resource management. The first experiment tested that PyCompss can properly manage the hardware resources available in the supercomputer.

6.1 Application Analysis

From Figure 5, several observations can be made. First, the tasks take different times to complete with some taking almost half the time. This is due to the different number of epochs from the configuration file. Second, from the event flags, 24 tasks were started at the same time. The remaining tasks are started as soon as a new resource is available, in this case cores 4, 10 and 16 from node 2. The entire application takes 207 minutes. However, as will be shown later, this is not entirely necessary and the process can be stopped as soon as one task achieves a specified accuracy.

In Figure 6(a), each task runs on its own node as specified and all tasks run in parallel. The first node seems empty as it is used by the worker. Like in the previous case, some tasks finish earlier than others. This means that it is possible to run the same application with half the number of nodes for almost the same amount of time as the nodes remain idle for the tasks that complete. This is shown in Figure 6(b). Clearly, this is a better utilisation of resources. It is important to note that no code changes are required to run across multiple nodes, the user just has to request more nodes when submitting the job. Scaling from a single node to multiple nodes is seamless.

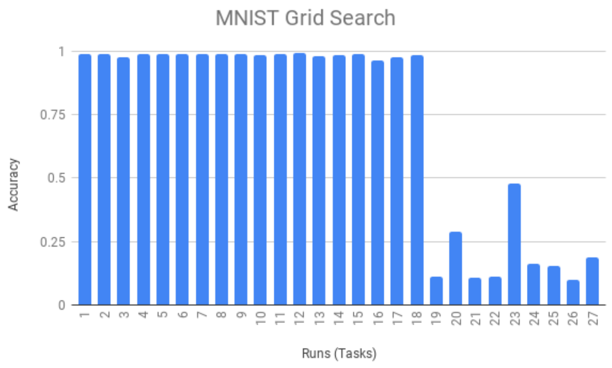


Figure 7: MNIST Hyperparameter optimisation using Grid Search

Figure 9 shows the time taken to complete the MNIST (CPU nodes) and CIFAR (GPU node) experiment using different configurations. By increasing the number of cores for each task, there is a continuous decrease in the time taken. However in the case of a single node, the time starts to increase after 4 cores. This is because assigning more cores than the total available means that some tasks will be waiting for resources. If the total number of cores available is the same or close to the number of cores requested by the application, the execution becomes sequential and therefore takes a longer time. One should therefore increase the number of nodes as they increase the number of cores per task to create a bigger pool of resources. This is evident when using two nodes as the time taken by the application continues to decrease.

There is a very significant time difference in the GPU Node. The GPU node has 4 GPUs and 160 cores. We assign each task a single GPU (therefore only 4 parallel tasks) and continuously increase the number of CPU cores. When using a single core, the time taken is even higher than that of CPU node. The explanation for this is that even though deep learning is significantly accelerated by GPUs, in our set up data preprocessing takes place in the CPU. Therefore a powerful GPU with just a single core is irrelevant as it will be idle more of the time. Increasing the number of cores brings down the time for the entire HPO process to less than an hour even though only 4 tasks run in parallel. Also important to note is that for the GPU node, we run the CIFAR10 dataset, which is much bigger in size. This is to create a noticable difference as the MNIST dataset is too small.

6.2 HPO Results

When all the tasks are done, we plot the results the same figure for easier comparison. Figure 7 shows the result of HPO for MNIST dataset after the entire application has completed. MNIST is a relatively simple application that generalises well after just a few epochs. Most of the combinations of hyperparameters are able to attain above 90% accuracy. For such task, early stopping is of paramount significance as it makes no sense to continue with other tasks after one has achieved the desired accuracy.

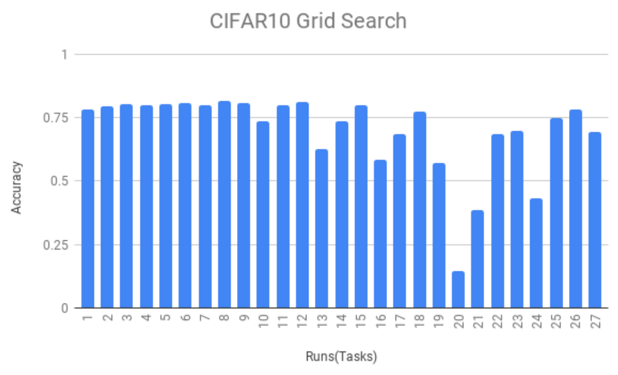


Figure 8: CIFAR10 Hyperparameter optimisation using Grid Search

CIFAR 10 is a slightly bigger and more complex benchmark in comparison with MNIST. Figure 8 shows the results of HPO for CIFAR 10 dataset. Most of the experiments perform well on the given hyperparameters. As mentioned earlier, random search would be a better alternative in this case as its possible to determine a good set of hyperparameters with just a few experiments.

6.3 Discussion

In this paper, our main focus was to provide a scheme and tool for HPO in HPC clusters. Though we have demonstrated the usage using primarily one algorithm, this scheme provides the user with the flexibility to choose and implement any HPO algorithm. Furthermore, even though all the experiments are implemented with Tensorflow, our scheme does not constrain the user to any framework. We focus on structuring the application rather than the inner details of the application.

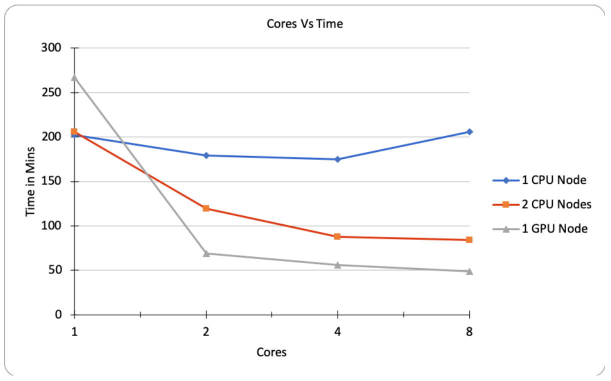


Figure 9: Time Vs Cores

Besides easy programmability as evident in listing 2, our scheme scales with an increase in the number of both cores and nodes. We tested scalability up to 27 nodes as shown in the traces. The time taken for the entire HPO even when using an algorithm such as grid is drastically brought down with an increase in the number of resources. Furthermore, our tool provides the much needed flexibility in resources management. By specifying the number of GPUs and CPUs for a task, one can come up with an optimal number and type of resources depending on the task.

7 CONCLUSION

In this paper we have presented a HPO scheme based on PyCOMPSs as an alternative tool for Hyperparameter Optimisation. We have shown that we can span multiple trainings across multiple nodes in a supercomputer and reduce the entire HPO process to days or hours instead of weeks. We have shown that our scheme is not only simple and easy to implement but provides all the features for a HPO tool discusses in section 1.

We hope to provide researchers with an alternative tool to accelerate HPO in complex infrastructures such as supercomputers and cloud. Furthermore, we provide a framework agnostic tool with a easy implementation. Even though we intend to support exhaustive grid search and random search, different algorithms can easily be implemented. We further present PyCOMPSs as a framework that enables seamless distributed computing to the machine learning community to facilitate further discussions and innovation around the subject. For future work, we are developing a library that puts together all key algorthms in HPO in an easy to use way. This library will enable the user to perform HPO over any search space by simply calling a function and specifying the algorithm.

ACKNOWLEDGMENTS

The project that gave rise to these results received the support of a fellowship from the "la Caixa" Foundation (ID **100010434**). The fellowship code is **LCF/BQ/DI17/11620059**.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie SkÅĆodowska-Curie grant agreement No. **713673**.

REFERENCES

[1] [n. d.]. Paraver: a flexible performance analysis tool. ([n. d.]). https://tools.bsc.es/paraver

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). https://www.tensorflow.org/ Software available from tensorflow.org.

[3] J. Bergstra, D. Yamins, and D. D. Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28 (ICML'13)*. JMLR.org, I–115–I–123. http://dl.acm.org/citation.cfm?id=3042817.3042832

[4] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for Hyper-Parameter Optimization. In *NIPS Proceedings.*

[5] James Bergstra JAMESBERGSTRA and Umontrealca Yoshua Bengio YOSHUABENGIO. 2012. Random Search for HyperParameter Optimization. *Journal of Machine Learning Research* (2012). https://doi.org/10.1162/153244303322533223 arXiv:1504.05070

[6] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. *End to End Learning for Self-Driving Cars*. Technical Report. arXiv:1604.07316v1 https://arxiv.org/pdf/1604.07316.pdf

[7] Lars Hertel, Julian Collado, Peter Sadowski, Julian Collado, and Pierre Baldi. 2018. Sherpa : Hyperparameter Optimization for Machine Learning Models. Nips (2018). https://www.semanticscholar.org/paper/Sherpa-{%}3A-Hyperparameter-Optimization-for-Machine-Hertel-Collado/342c5b941398c733659dd6fe9e0b3b4e3f210877

[8] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

[9] Jeffery Kinnison, Nathaniel Kremer-Herman, Douglas Thain, and Walter Scheirer. 2018. SHADHO: Massively scalable hardware-aware distributed hyperparameter optimization. In *Proceedings - 2018 IEEE Winter Conference on Applications of Computer Vision, WACV 2018.* https://doi.org/10.1109/WACV.2018.00086

[10] Alex Krizhevsky and Geoffrey Hinton. 2009. *Learning multiple layers of features from tiny images.* Technical Report. Citeseer.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. [n. d.]. *ImageNet Classification with Deep Convolutional Neural Networks.* Technical Report. http://code.google.com/p/cuda-convnet/

[12] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (may 2015), 436–444. https://doi.org/10.1038/nature14539

[13] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/. (2010). http://yann.lecun.com/exdb/mnist/

[14] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. 2018. Tune: A Research Platform for Distributed Model Selection and Training. 2012 (jul 2018). arXiv:1807.05118 http://arxiv.org/abs/1807.05118

[15] Francisco Madrigal, Camille Maurice, and Frédéric Lerasle. 2018. Hyperparameter optimization tools comparison for multiple object tracking applications. *Machine Vision and Applications* 30, 2 (mar 2018), 269–289. https://doi.org/10.1007/s00138-018-0984-1

[16] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2017. Ray: A Distributed Framework for Emerging AI Applications. (dec 2017). arXiv:1712.05889 http://arxiv.org/abs/1712.05889

[17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[19] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. [n. d.]. *Practical Bayesian Optimization of Machine Learning Algorithms.* Technical Report. https://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf

[20] William Song and Jim Cai. 2015. End-to-end deep neural network for automatic speech recognition. *Standford CS224D Reports* (2015).

[21] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M. Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. 2017. PyCOMPSs: Parallel computational workflows in Python. *International Journal of High Performance Computing Applications* (2017). https://doi.org/10.1177/1094342015594678