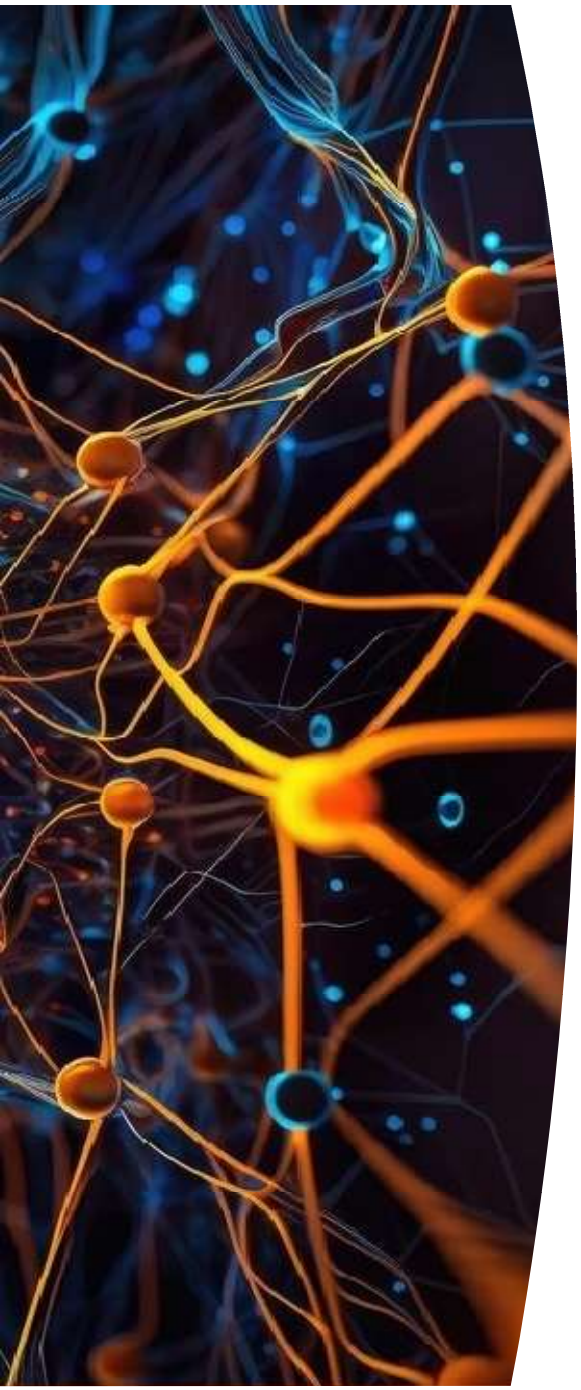

Lucid: A Non-intrusive, Scalable and Interpretable Scheduler for Deep Learning Training Jobs

Seminar hot topics on OS

Paper review

Paper by

- Qinghao Hu
- Meng Zhang
- Peng Sun
- Yonggang Wen
- Tianwei Zhang

Review by Maxime Geminiani Welcklen

Table of Contents

Table of figures

1. Context

There exists a striking similarity that one can observe when navigating through LUCID's cited papers, whether they are related to the analysis of the datasets used, the state of the field, or simply other proposed solutions: they always start by stating that deep learning is a cutting-edge revolution that has the potential to solve many problems around computer vision, natural language processing (NLP) with large language models (LLM), and other various problems. It is therefore an attractive field for companies and an interesting one for research. Based on this assumption, we can expect a general desire to make it more efficient. This section starts by laying out the foundations of the problems that LUCID attempts to solve.

1.1. Deep learning

In the context of this review, deep learning (DL) is a paradigm or a subset of machine learning / artificial intelligence. It is based on neural networks that can be traced all the way back to 1943. The general idea of neural networks originates from the "perceptron", a neuron-like concept that has an arbitrary number of inputs that are passed to an internal summation with an added bias. This result then goes to an internal activation function to form a single output. Such neurons are weak alone as they are mostly modeling a simple linear regression. Multiplying the number of neurons with the same input yields interesting results: Such a construct is often referred to a layer. A simple neural network can be composed of an input layer, a hidden layer of N neurons, and an output layer.

Deep learning expends on this structure with the following assumption that holds true under empirical observations: it is possible to chain hidden layers of neurons, and although having a few connected hidden layers does not seem that impactful, multiplying the number of hidden layers can yield good results. Multiple popular architectures are built on this assumption, such as generative adversarial networks (GANs), convolutional networks (CNNs), Resnets, or transformers.

The learning is done by passing through the data under units called "batches". A forward propagation produces a result, the result is compared to a reference label, and the error is backpropagated through the layers. The processing of all batches of data is called an epoch. Training a DL network, particularly the modern ones can be a gargantuan task involving millions to billions of parameters (neurons) in their layers. This can thus require a significant amount of computation (number of data) and memory (size of the model).

The computation described above – given a model, a bunch of data, a set of hyperparameters and a number of epochs – can be regarded as a 'job': An atomic unit of work that has to be done. Several properties can be stated about DL jobs:

- The jobs themselves are mostly repetitive. This will be particularly important in following considerations as a repetitive task is one that can be estimated with satisfying accuracy given enough data.
- There is a high probability that the same kind of job is submitted in bulk. Either due to adjustment in the data where the model is re-trained 'fresh' such as for recommender systems at google or amazon, or for hyper-parameter tuning where the same model is trained multiple times with slightly different meta-properties to find the best one.
- As companies working on DL jobs are prone to development and testing, there is a high chance of jobs failing during execution.

A common scheme for DL entities (companies, research) is to regroup their needs in order to optimize their costs. This can be done in the form of a GPU cluster where the hardware is connected in nodes that can execute these DL jobs. However, some warnings have been raised regarding these clustered setups.

1.2. Cluster and Scheduling

A GPU cluster is an agglomeration of GPUs grouped under nodes that often have associated support hardware such as CPUs or network components. These nodes can have very high internal data bandwidth and act as a hardware-as-a-service with an interface allowing users to submit jobs that will be executed on the nodes.

A single model testing, tuning, or even training can be expressed as a number of very similar jobs to run on the cluster. With multiple tenants cohabitating, such as several teams working on different projects or entities sharing a cluster, the number of jobs to run tends to out scale the available resource of the cluster: the GPU computational power.

This problem of jobs of different lengths competing to obtain a limited and shared resource is analogous to one of the core aspects of operating systems (OS) that must deal with that situation for every program that runs on a CPU. The scheduling and the respective entity taking these decisions, the scheduler, is responsible of giving access to the resources in the best way possible. Two useful metrics to describe the 'best way possible' to assign resources in the scope of GPU clusters are the job completion time (JCT) that simply represents the total time taken to run through a set of jobs and the average queuing time representing the total time spent in the queue.

Although the situation between both environments is not directly comparable due to the nature of the jobs and the scale of their runtime, they share a common problem related to network and queuing theory: The head-of-line problem, or HOL. The HOL problem characterizes a situation that arises when a few large jobs are executed on all available machines while a group of smaller jobs wait in the queue. This creates a "clogged" situation that inflates the queuing time for the blocked jobs and can be reflected on the JCT. To avoid this HOL problem, schedulers tend to run the smallest jobs first.

The schedulers have become a critical part of the architecture of GPU clusters: as long as they are not perfectly scheduling jobs on the hardware, then it can not be utilized to the fullest of its capacities. Miss-management from the schedulers can lead a GPU cluster to require an over-dimensioning in order to resolve a group of tasks that it should be able to process. From the cited papers of LUCID, it can be shown that:

- Large parts of the clusters are under-utilized, although there is queuing time, meaning that an improvement is possible.
- Multiple iterations of schedulers have achieved better JCT and/or average queuing time, showing further improvement is possible.

1.3. Usual limitations of schedulers

Firstly, the time constraint is always a requirement for scheduling. Scheduling for DL jobs is performed on different scales compared to operating system's schedulers. Whereas the later have to take lightning quick decisions on limited information and can perform lot of pre-emption of jobs well below the sub-second level, the time of execution of DL jobs can vary between hours to days, letting more time for the scheduler

to take its decisions. Some fully mathematical approaches to scheduling are still too slow even for this setup, approaching minutes or hours for decisions.

Secondly, as a consequence of some of the mechanisms that are used in schedulers, the intrusiveness of the state-of-the-art algorithms is put in question. Pre-emption in the context of scheduling can be seen as the capacity of taking back a resource that was given to a job. In OS, pre-emption comes at a cost: a process has to be taken off and another one put back on the processor. In GPUs, pre-emption can be very expensive as the whole model has to be swapped. Furthermore, schedulers can do job packing as a way to better utilize the GPU by sending two jobs to run on the same hardware. This makes pre-emption more costly and difficult. Pre-emption can also be a way to adapt batch sizes mid-training. In order to do job pairing and pre-emption, the schedulers require to place checkpoint in models to safely stop-and-restart them in another machine. All of these involve a heavy intrusiveness of schedulers, meaning that the code must be adapted, that the deployment and maintenance is made harder, and that errors can arise from the specific frameworks introduced by the scheduler.

Finally, many research are trying to apply machine learning methods to solve DL scheduling. However, depending on the type of algorithms and the structure of the schedulers, the decision-making process can become unreadable for a human. But unlike computer vision or large language models where accuracy is key, the decision to schedule a job instead of another would benefit greatly from being explainable.

1.4. Case study

A few schedulers are presented to reinforce the understanding of the field and see examples of the limitations discussed above. They are presented in chronological order, Lucid being the latest, as innovation tends to follow an iterative process.

1.4.1. Tiresias

Tiresias is a scheduler proposed by the university of Michigan, Microsoft, Bytedance, the UNIST and Alibaba in 2019 [1]. It has a very similar drive as Lucid: There needs to be a good scheduler for DL jobs as low utilization of GPUs is inefficient and better JCT can be achieved with good policies. It has strong considerations for distributed learning where a single model is deployed over multiple GPUs in order to speed up the training. The prior existing scheduling mechanism taken into consideration is a simple resource manager giving resources to the first coming jobs based on Yarn.

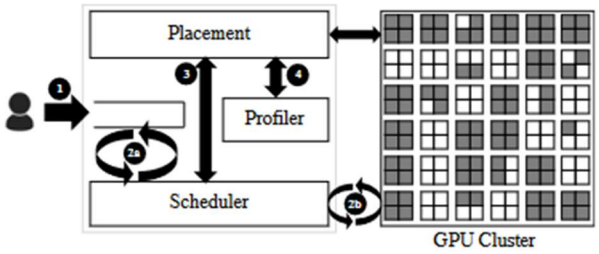

Figure 1 – Architecture overview of Tiresias

Tiresias's core principle is to prioritize least attained service, as in assign resources to the jobs that have the least amount of resources already allocated. This makes Tiresias a preemptive algorithm that sparsely gives-and-takes resources to cover the job pool. After some time, a profiling of the jobs based on their history can take place in order to create future estimation of their duration as seen in the Figure 1.

One very interesting remark by Tiresias is around consolidation. Consolidation is a method that aims to maximize GPU utilization by matching multiple jobs per server/GPU and in the case of distributed learning, trying to allocate enough resources for a given job's distributed requirements. However, Tiresias states that some models – especially big ones with high tensor size distribution – are highly reactive to consolidation, meanwhile most of the models run fine. These problematic models are classed as "skewed". It also states that trying to consolidate at all costs could create situations where jobs are waiting in the queues while a number of available GPUs are left idle because they are not matching the perfect setup for said job.

Another interesting note around preemption can found in the paper: *"We observe that preemptive scheduling is necessary to satisfy these objectives. One must employ preemption to avoid head-of-line (HOL) blocking of smaller/shorter jobs by the larger/longer ones – HOL blocking is a known problem of FIFO scheduling currently used in production"*. This is particularly interesting as Lucid is proposing a preemption-free solution avoiding HoL problems via prior profiling of the jobs.

Tiresias uses time and space-based metrics into consideration when during its decision process. This paper already shows that using both metrics in its policy significantly improves the results in term of JCT.

Overall, Tiresias can be seen as a precursor that Lucid bases itself on in many aspects: It learns that profiling job is generally a good idea to estimate their duration and act with that information, it learns from its analysis that using space and time metrics yields good results, and there is also the first glances of job pairing with the consolidation mechanism, as well as the concept of interference between some jobs that might be less compatible than others. However, Lucid improves on many of these points and achieves similar results while being preemption-free, staying out of the dreaded intrusive algorithms.

1.4.2. Gavel

Gavel is a scheduler proposed by Microsoft research and Standford in 2020. Its goal is to make a scheduler that can manage a meta-objective such a "achieve the lowest JCT" while taking heterogeneity in account [2]. It works by treating scheduling as an optimization problem between a set of policies such as FIFO, minimum makespan, shortest job first, hierarchical policies, etc… And a set of running states that the jobs can be in.

GPU clusters can be homogeneous or heterogeneous. The former defines a setup where every GPU is the same whereas the later describes nodes composed of a number of different versions / families of processing units. The "running states" that the jobs can be in are the time allocated for the job on a given resource. Of course, there is an affinity to take into account between hardware accelerators and the jobs that run on them.

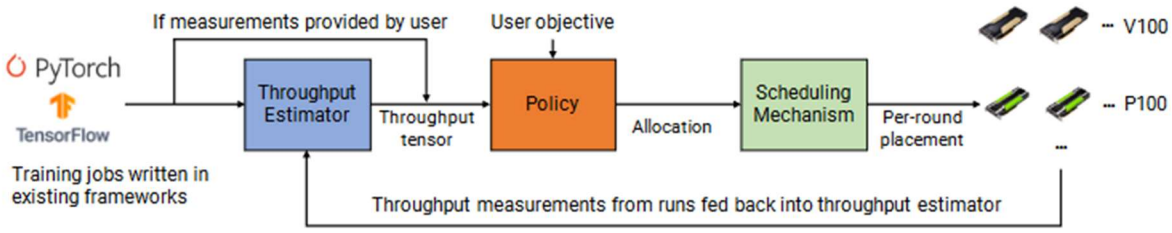

Figure 2 – Architecture overview of Gavel's round-based concept

Gavel works in a round-based iterative process showcased in the Figure 2 and therefore relies on checkpointing to start and stop the underlying training of the models. The scheduler runs the algorithms on their allocated GPUs for a given time, estimate the throughputs on the given timeframe, and uses the estimated affinity of the jobs with the hardware to compute the most efficient allocation for the next round. Some resources can be shared between multiple jobs as a collocation mechanism.

Gavel's results are presented with a round time of 6 minutes. This is a useful metric to keep in mind as it can be compared to the profiling of Lucid. Furthermore, although the overhead is observed at around ~1-3% for a 25-jobs steady cluster, the scalability in more realistic conditions is prohibitive. As Gavel uses round-based optimization taking every job and every allocator into account, the complexity of the computation becomes a problem on bigger clusters. Lucid's analysis states that Gavel fails on a 2048 jobs cluster as it takes thousands of seconds to compute the next round's allocations.

Gavel is not a direct competitor to Lucid: It is closer to a meta-algorithm that uses already established policies and a score function and is focused on scheduling the jobs on the best hardware available given a heterogeneous set. This problem is completely avoided by Lucid and the prohibitive computation cost of Gavel comes in part because of this additional dimension.

1.4.3. Horus

Horus is a scheduler proposed as part of a study published in in the IEEE transactions on parallel and distributed systems in 2022 [3].

The first part of the paper consists in an analysis of a massive amount of DL jobs in the scale of 3 million. Horus leverages this significant dataset to draw conclusions about the job's behaviors and the effectiveness of adding additional information into the profiling. To be precise, it proposes to take three point-of-views into account when drafting the metrics by which a job profile is defined: The job, the cluster, and the user that submitted the job.

The point of view of the cluster consists in taking into account the utilization trend of the overall system varying through time. For example, a clear daily pattern can be extracted from the data. The point of view of the job consists in the regular job execution time, the very first metric used by simple scheduling algorithms such as shortest job first. However, space could play a role as important as time data in the job's perspective. Finally, the point of view of the user consists in analyzing the history of job scheduled by users. For example, the statistic of 5% of users occupying 60% of the GPU time overall means that users are significant factors that can help profiling jobs.

With that in mind, the paper defines a two-part solution for efficient scheduling. These two components are the Model update engine and the resource orchestrator. The principle is simple: The resource orchestrator is composed of a number of services. A service has an internal policy, such as quasi shortest job first (QSSF). Every service is coupled to a model that uses historical data (from the three points of view) in order to profile the jobs in the cluster. The policy uses the results of the models as a priority to schedule jobs on the cluster. The model update engine is self-explanatory and regularly updates the models with fresh data. QSSF is non-preemptive and relies on the profiling done by the models to correctly plan the execution of the jobs.

The models used in Horus are gradient boosting decision tree (GBDT). GBDT is an ensemble model that creates a strong machine learning model composed of many weak learners. In this case, it is composed of a chain of decision trees reinforcing each other. Because of their architecture and although some visualization methods are proposed, GBDT are widely regarded as black box / unexplainable machine learning algorithms. This means that Horus's profiling mechanism can not be fully transparent to a user.

2. Analyses of Lucid

This section focuses on Lucid: it summarizes its implementation and key features, links them to the related work, and finally suggests a few limitations and/or axis of reflection about enhancements to the scheduler.

2.1. LUCID: A symphony in three notes

Lucid is a scheduler proposed by the Nanyang Technological University and the Shanghai AI Laboratory in 2023 [4]. It is driven by a set of goals: solving a list of limitations present in other scheduler algorithms. These five limitations can be summarized as Inflexibility, High integration / maintenance cost, model quality degradation, limited scalability, and opaque decision making.

The model quality degradation can be present when schedulers adapt the hyper-parameters, particularly the batch size, during the training in order to optimize resource utilization. The inflexibility and high integration & maintenance are due to schedulers that force the utilization of a framework, library, or add-on, typically to either observe the code or checkpoint it. The limited scalability is reached if a scheduler's decision process is too long to keep up with the arrival of DL jobs. Finally, the opaque decision making is an undesirable feature for a scheduler, but many solutions involving machine learning models are built as black boxes.

On the three examples given in the previous section, it can be noted that those limitations appear:

- Tiresias is entirely based on preemption as a core of the least attained policy. This means that an ad-hoc implementation of a DL model in Pytorch will likely not be compatible with a deployment with Tiresias, as the model will not be able to be stopped. This in turns escalate to problems one and two: inflexibility and high integration costs.
- Gravel relies on preemption and thus suffers from the three limitations named above. But beyond that, it has proved incapable of keeping up with higher job load, hence also falling for limitation 4.

- Finally, Horus does not implement preemption as the QSSF relies on strong profiling in order to have a good prior estimation of the job's lengths. However, the profiling itself is done using GBDT and make it fail limitation n°5.

Lucid heavily takes from Tiresias and Horus to establish its structure. It is therefore logical that both of these algorithms are used as comparatives on its final results.

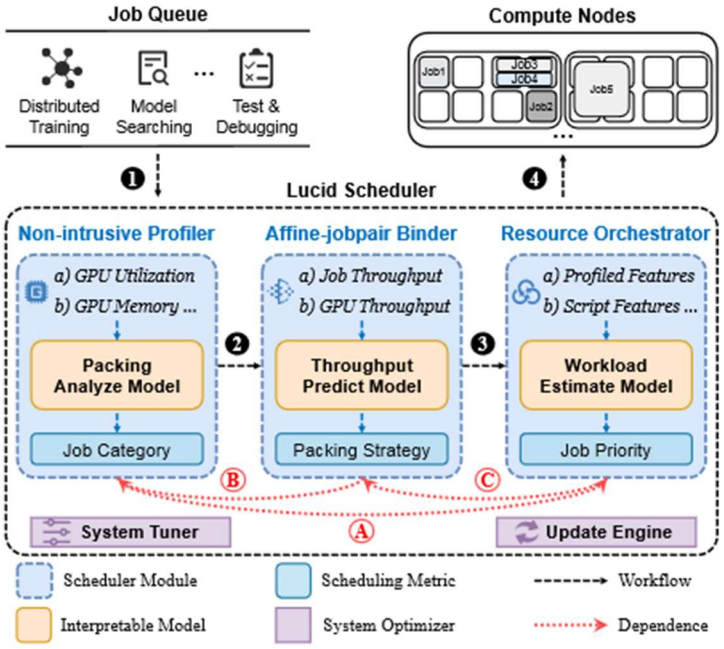

Figure 3 – Architecture overview of Lucid

Lucid is built around three components and two helpers organized as in the Figure 3. The job pool is analyzed through an initial profiler where they are categorized. The jobs then go through an affine-jobpair binder where they are matched by affinity. Finally, these pairs go through a resource orchestrator to be assigned to hardware. Each of these components has exactly one machine learning model. These models are kept fresh with an update engine and a system tuner.

The first step of the chain, the profiling, takes some inspiration from Tiresias. The profiling consists in running the jobs for a given and very short period of time in order to obtain running metrics. The profiling is two-dimensional: As Tiresias concluded a few years before Lucid, taking time and spatial utilization is beneficial to the overall efficiency of the scheduler. The profiling done by Lucid can also be compared to the round-based iterations of Gravel: Where Gravel continually runs-and-evaluates its jobs, Lucid does one pass at the start and base its estimation on it. It can be noted though that Lucid assumes a homogenous cluster, meaning that this single pass is representative of the actual training job, whereas Gravel is made for heterogeneous ones and uses the rounds to continually estimate the affinity of jobs on their accelerators. The model used in the profiler is a simple decision tree that takes the results of the profiling run and categorize jobs into small, medium, and jumbo.

The second step of the chain, the affine-jobpair binder, aims to find the best pairs amongst all possible matching of jobs based on their category. This consideration of interference when forming pairs can be linked to the skewed classification of Horus: some jobs react badly to pairing, therefore, it is useful to have

this scan process to determine the best co-joined runners. The model of the binder uses the throughput estimation model that is based on the cluster's information, such as its temporal status. For example, the binding is indolent / lazy: If the cluster is not at full capacity, then no pairing needs to occur, and the resulting pair map will simply consist of single jobs.

Finally, the resource orchestrator uses the information of its two predecessors to determine a job priority. The overall focus of the orchestrator is to prioritize light and small jobs in order to avoid the HoL problem and have the best resulting queuing / job completion time. It does so by training a Generalized Additive Model (GA2M), a white box type model that is close to linear regression. It uses input metrics related to cluster, user, and job's point of views, as heavily encouraged by Horus. Its output is the priority by which the jobs will be ordered for the access to resources.

This system allows Lucid to efficiently profile every job of the job pool and determine a priority for each best matching pair, meaning it does not intrude in user code. Furthermore, tests indicate that the three models used are more than fast enough to compete with large DL workload: decision tree and regression (GA2M) models are heavier during training time and fast during inference. As the three models used are white boxes, every decision taken about job profiling, job pairing, and resulting priority can be traced back to the job pool. However, the paper has some aspects that can be scrutinized and would warrant a discussion.

2.2. Critical discussion

This section regroups a list of suggestion and discusses several weak points and/or possible limitations in the implementation of the paper.

2.2.1. LUCID: A complex symphony

Lucid's second concern is that schedulers can generate "*High integration and maintenance cost*" because of the necessity to adapt the model's code in order to allow intrusion from the scheduler. Although this concern is addressed properly by getting metrics from a profiling run and not having the need for preemption and checkpointing, an eyebrow can be raised concerning another source of maintenance cost. Lucid is implemented in 4700 lines of codes of python, which can be considered a medium-sized project. Furthermore, Lucid relies on several external libraries (GA2M, GBDT…) to work. The point can be made that Lucid itself is a significant software that has to be maintained by one or multiple engineers of the clusters. Furthermore, Lucid requires data to be trained with in order to be more efficient. Preparing and processing the data output by a generic GPU cluster in order to have a seamless deployment of Lucid can be considered an integration cost of the scheduler.

2.2.2. Suffering from success: fail fast

Lucid's particular 'profiling run' mechanic allows it to fail fast: As stated by multiple studies cited by the paper, a lot of the DL jobs submitted on companies are tests and will fail very fast. These usually cause a lot of troubles to schedulers that rely on an expectation of reliability as a crash instantly stops a job and creates a big discrepancy between the expected and the actual duration of the job. This is a very good advantage for lucid, however, this could also be a strong bias in favor of it when testing on big dataset that include these failing jobs such as the ones employed in the trace simulations compared to schedulers that have no protection against this phenomenon. This does not reduce the effectiveness of the solution, but it can be useful to note that some of the apparent performance upgrade over Tiresias and especially Horus

could be partially explained by these catches and not only by a better scheduling in the strict sense. This could be proven or disproven by doing a sort of ablation study where the failing jobs are removed from the comparative tests.

2.2.3. Fairness

In some settings, fairness can be very important in a GPU cluster. As long as a single company owns it, then the performance can be a fine single goal. But some clusters will be multi-tenants: research clusters can be shared across universities and cloud clusters will be shared by design (albeit with some level of separation). If a tenant of a virtual cluster is submitting very large jobs such as GPT-like training including billion of parameters, Lucid's resource orchestrator could deny it access to GPUs as long as a sufficient number of small jobs are submitted. There is no fairness mechanism enforcing the allocation of resources to the groups associated with the clusters. This is linked to the next discussion.

2.2.4. Starvation

Even in a single tenant setting, Lucid's resource orchestrator has no apparent anti-starvation mechanism. The resource orchestrator is described in section **3.4 - Resource Orchestrator** as such:

"Workload Estimate model predicts the duration of each job and then the prediction is multiplied by the number of GPUs as the job's priority value. […] Next, the job queue is sorted according to the priority values. Then it checks whether job packing is allowed at the current moment. If not, jobs are allocated in an exclusive manner. […] If yes, we pack jobs suitable for colocation, and eliminate jobs with little remaining runtime."

The jobs are taken from the job queue to be estimated and sorted, after which resources are allocated according to these values. The sorting is done on the priority value weighted by the number of GPU required by the task.

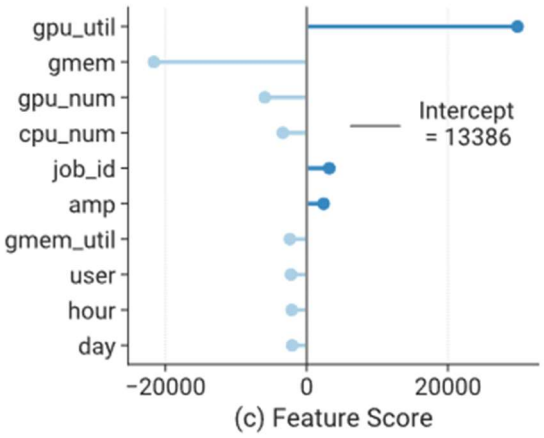

Figure 4 - Example of a job's priority computation used in the orchestrator of Lucid

A job's priority is computed by the workload estimator using prior results from job and cluster profiling. The computation of a job's priority can be seen in the Figure 4. As the inputs of the profiling and the affine job-pair binder are fixed and as the orchestrator is configured to avoid the HoL problem, we can imagine that a 'jumbo' job will have a heavily penalized priority value. If this holds true, then under a heavy utilization / overload of the cluster, this task could be delayed until all smaller tasks are all completed.

The only reference to that concern in the paper is: *"Lucid obviously outperforms Tiresias for both large and small jobs, which demonstrates large jobs will not experience starvation in Lucid scheduling"*. But running an experiment and outperforming a 3-year-old scheduler taken as a basis for Lucid can be considered as a very weak proof that there is no problem around starvation.

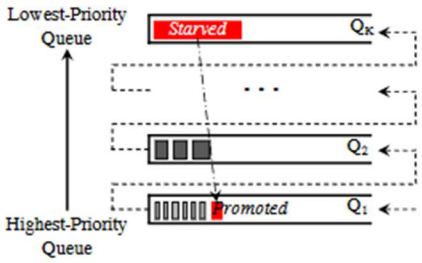

Figure 5 – Management of starvation in Tiresias via a looped multi-queue system

Some OS schedulers implement an aging mechanism to encourage the execution of a job after it has waited long enough in the queue. For example, the Figure 5 shows the multi-queue system of Tiresias where potentially starving jobs in the least prioritized queue are looped back to the most prioritized queue in order to be executed. However, Lucid never refers to a time-related metric when computing the priority value of a job-pair. In the meantime, depending on the trace used for the simulations and the general distribution and arrival of jobs in real conditions, it is possible that starvation is only a temporary concern: if the cluster does not end up regularly over-loaded for long periods of time, a starving jumbo job will eventually have resources allocated when the workload lightens up.

2.2.5. Heterogeneity

An obvious weakness of the current implementation of Lucid is its assumption of running on homogeneous clusters. This is addressed by the paper itself as a future work to be done. However, Lucid relies on this assumption to have this single-pass profiling upon which it bases its whole analyses and priority score. Without the guarantee of having a representative metric at the end of the profiling, the whole chain could be compromised. Furthermore, this seems like a complicated issue to fix: Will the profiling be done on each possible hardware present in the cluster, thus forcing a part of every unique accelerator to be assigned to profiling duty ? Would a heterogeneous cluster be broken down into homogeneous virtual cluster ? How would that impact performance ? The profiling of Lucid is one of its strong points and this issue could be undermining the foundations of the algorithm.

2.2.6. Data is key

Lucid can work in a cold-start setup, but the favorable results obtained and compared to Tiresias and Horus are taken with several month of cluster logged data in mind. Having this starting data prevents rough estimations made by the scheduler when data are missing and a user can expect degraded performance when starting cold. This can be a limiting factor of adoption for small clusters that would not have enough data to feed the scheduler. This would make Lucid more adapted to big established GPU clusters as opposed to relatively small clusters without massive backlog of history data.

2.2.7. Preemption-free absolute limitation

The preemption-free approach by LUCID means that a job given a resource will use it until completion. Some of the DL jobs can have a relatively long duration in the scale of hours, days, or weeks. This could

imply that LUCID can never reach the optimal JCT as it would involve breaking a job in multiple part, violating the preemption-free principle.

Let's craft such a situation using a heavily simplified problem: Let a system with two identical GPUs having to run three arbitrary jobs of length 1, 1, and 20 days. The best possible JCT obtained by Lucid is 20 days by giving a GPU to the longest job. As seen in the implementation, LUCID would probably pack the two small jobs and run this pair on a GPU, meaning one of the two hardware would be unused for the remaining 19 days. However, using preemption, the 20 days job can be broken into two 10 days jobs and LUCID could pack the 10 days and 1 days jobs together (10-10; 1-1; as it loves to pair by similarity) to achieve an overall max JCT of 10 days with a better overall GPU utilization with nine idle days on one GPU.

This limitation is relevant if and only if jobs can be broken. This is highly dependent on the type of DL jobs executed on the cluster: Classical whole model DL can effectively be broken up as the gradient update is a sum that can be computed in several parts, but distributed learning or layered models could cause issue to the system and bring the overhead of networking. Lucid's take on the subject seems clears: the advantages of being preemption-free outweighs the absolute error that can not be reduced compared to the perfect scheduling.

2.2.8. Going beyond…

The reasonable! Is it even possible to question the very core of one of the problems that LUCID and many of its predecessor paper addresses: the race to full cluster utilization ? Many of the papers discussed in this review have identified schedulers as one of the bottlenecks in the systems as proven by the relatively low GPU utilization on big clusters.

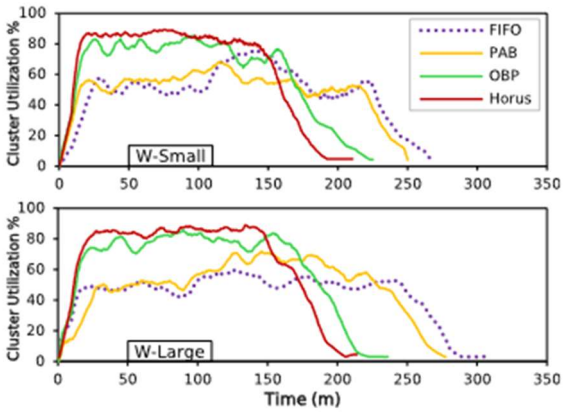

Figure 6 – Analyze of the GPU cluster's utilization under various scheduler, including Horus

However, some analyses cited through the paper such as the one displayed in the Figure 6 reveal the relatively low utilization to be between 60 to 80%, depending on the cluster and the scheduler. By jumping into the world of operating systems, a CPU system tailored to a given task would not necessarily be fit to be run at exactly 100% utilization: it is risked to correspond exactly to the average expected utilization, as sudden bursts of tasks could break the system and the scalability would be questionable. Could it then be within the realm of possibilities that the 60-to-80% utilization figure on the Amazon / Microsoft / others GPU clusters taken as examples are intentionally kept below the full utilization mark ?

Additionally, as the jobs are not tailored to the hardware specifications such as memory, it seems outright impossible to reach the maximum GPU utilization: GPUs are discretized amount of memory and computational power and jobs are discretized amount of work to execute. Although the job packing mechanism tries to resolve this by executing two jobs in one GP, therefore increasing the chances of using them fully, it seems futile to run after the full utilization metric.

2.3. Conclusion

Lucid is a scheduler that can be considered to be the heritage of both Tiresias and Horus, the former being at the basis of its ideas, and the latter being a cornerstone in metric analysis and preemption-free paradigm. It is an iteration that takes from both schedulers and adds novelty to create one that goes beyond their capacity, both in term of actual completion time and in term of a more complex metric – explainability. Lucid also achieves the feature of being an ad-hoc scheduler designed to work out-of-the-box with most models that can run locally on a GPU. Despite some setbacks, Lucid could be an important modern DL scheduler for big clusters if it can address some concerns such as being able to run on heterogeneous clusters following the footsteps of Gravel. Overall, Lucid respects its self-defined requirements by addressing each of the five concerns stated in the start of the paper, and the weak points identified are mostly around other areas of the algorithm that are not stated in its goals. It is also a well-described paper that takes its time to guide the reader through the details of the implementation without any major hole.

3. References

[1] M. C. a. K. G. S. U. o. M. A. A. Juncheng Gu, M. a. B. Yibo Zhu, M. a. U. Myeongjae Jeon, M. Junjie Qian, A. Hongqiang Liu and B. Chuanxiong Guo, "Tiresias: A GPU Cluster Manager for Distributed Deep Learning," *USENIX Symposium on Networked Systems Design and Implementation (NSDI),* 2019.

[2] K. S. F. K. A. P. M. Z. Deepak Narayanan, "Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads," *USENIX Symposium on Operating Systems Design and Implementation,* 2020.

[3] G. Yeung, D. Borowiec, R. Yang, A. Friday, R. Harper and P. Garraghan, "Horus: Interference-Aware and Prediction-Based Scheduling in Deep Learning Systems," *IEEE Transactions on Parallel and Distributed Systems,* 2021.

[4] M. Z. P. S. Y. W. T. Z. Qinghao Hu, "Lucid: A Non-intrusive, Scalable and Interpretable Scheduler for Deep Learning Training Jobs," *Asplos,* 2023.