Effectiveness of Moldable and Malleable Scheduling in Deep Learning Tasks

Ikki Fujiwara*, Masahiro Tanaka*, Kenjiro Taura† and Kentaro Torisawa*
*Data-driven Intelligent System Research Center (DIRECT), Universal Communication Research Institute,
National Institute of Information and Communications Technology (NICT)
3-5 Hikaridai, Seika-cho, Kyoto 619-0289, Japan
Email: {ikki, mtnk, torisawa}@nict.go.jp
†Department of Information and Communication Engineering,
Graduate School of Information Science and Technology, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan
Email: tau@eidos.ic.i.u-tokyo.ac.jp

***Abstract*—Research and development of deep learning (DL) applications often involves exhaustive trial-and-error, which demands that shared computational resources, especially GPUs, be efficiently allocated. Most DL tasks are moldable or malleable (i.e., the number of allocated GPUs can be changed before or during execution). However, conventional batch schedulers do not take advantage of DL tasks' moldability/malleability, inhibiting speedup when some GPU resources are unallocated. Another opportunity for speedup is to run multiple tasks concurrently on one GPU, which may improve the overall throughput because a single task does not always fully utilize the GPU's computational resources. We propose designing a batch scheduling system that exploits these opportunities to accelerate DL tasks. As a first step, this study conducts an extensive case study to evaluate the speedup of DL tasks when a scheduler treats them as moldable or malleable. That is, the scheduler adjusts the number of GPUs to be (or already) allocated to a task in response to the fluctuating availability of GPUs. Simulations using our real workload trace show that if the scheduler can allocate 1–4 GPUs to a task or assign 1–4 tasks to a GPU, then the average flow time of moldable/malleable DL tasks is shortened by at least 15.1%/42.5%, respectively, compared to a Rigid FCFS schedule in which one GPU is allocated to each task.**

I. Introduction

Herein we explore ways to utilize our in-house shared GPU cluster more efficiently to accelerate our research and development of deep learning (DL) applications. DL has recently realized breakthroughs in diverse fields, including computer vision, speech recognition, and language processing. Researchers at our institute study natural language processing technologies. Often constructing a good neural network (NN) is costly due to training with vast data and/or numerous hyperparameters. Consequently, researchers employ GPUs to obtain outcomes of their DL tasks as quickly as possible. However, GPUs need to be utilized efficiently because they are expensive resources. A promising yet unexplored way to improve the efficiency is to add smart mechanisms for resource allocation to a task scheduler.

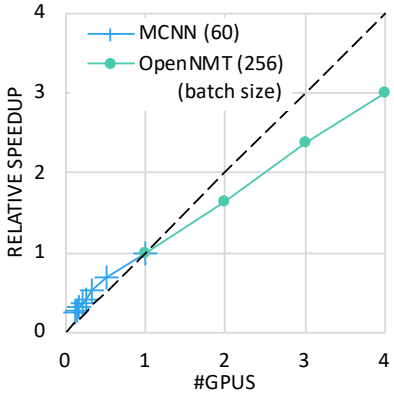
DL tasks often have two properties that are not exploited in conventional batch scheduling systems. First, DL tasks have inherent flexibility in the degree of parallelism. A DL task can be implemented so that is can change the amount of used resources without modifying the program because DL tasks are usually built on top of a scalable framework. Second, DL tasks have some predictability during their execution. For example, the runtime of a DL task can be estimated because it usually iterates a definite set of computation over given data and each iteration consumes almost the same amount of time.

We propose designing a batch scheduling system that implements two categories of mechanisms to exploit the flexibility and predictability of DL tasks. The first category includes moldability and malleability [1]. Moldability allows the scheduler to alter the amount of resources allocated to a task prior to execution (e.g., via a command line option), while malleability enables the scheduler to alter the allocated resources even in the middle of execution (e.g., via a suspend/resume mechanism). The second category is clairvoyance, which enables the scheduler to predict the remaining runtime of a task prior to execution (e.g., via the execution history of similar tasks) or during execution (e.g., via an extrapolation). To the best of our knowledge, this is the first attempt to explicitly exploit flexibility and predictability of DL tasks in a batch scheduling system.

As a first step towards the goal of designing a DL-aware moldable/malleable task scheduler, we conduct a case study to evaluate the effectiveness of each mechanism once implemented in a scheduler. Extensive simulations based on our real-world GPU workload reveal the following observations:

- Moldability improves scheduling quality. For example, the average flow time per task shortens by 15.1%–96.7% when a scheduler can allocate up to four GPUs to a task or assign up to four tasks to a GPU.
- Malleability can further improve scheduling quality given a moderate overhead for preemption.
- Clairvoyance does not improve scheduling quality if moldability or malleability is available.

The rest of this paper is organized as follows. Section II describes our motivation to use GPUs more efficiently. Section III formulates our problem. Section IV defines the scheduling algorithms that exploit moldability and malleability. Sections



#GPUs	Speedup
4	3.00
3	2.37
2	1.63
1	1.00
$1/2$	0.70
$1/3$	0.52
$1/4$	0.42
$1/5$	0.36
$1/6$	0.31
$1/7$	0.27
$1/8$	0.24

Fig. 1. Speedup curve observed in our benchmark. #GPU $= 1/n$ means concurrent execution of n tasks on one GPU.

V and VI present our case studies with those algorithms. Section VII reviews related works. Section VIII concludes this paper.

II. Backgrounds and Motivating Examples

It is a common practice to use hardware accelerators (e.g., GPUs) to speedup training in DL. GPUs are typically shared by multiple users via a batch scheduler because they are expensive resources. Users often submit a series of tasks that compute the same NN with different initial parameters as the initial parameter may significantly impact the learning results. When users simultaneously submit many tasks, the load of the GPU cluster can be high. In addition, users also experimentally explore the network structure because the effect of the network structure on the learning results is unpredictable. Unlike the survey of the initial parameters, reforming the network structure requires time. When some users are too busy to implement their networks, the cluster can become idle. The implementation and training cycle of NNs often leads to significant fluctuations in the load, especially when the number of users is small. However, it is impossible for users to predict the available number of GPUs when submitting their tasks.

To deal with the GPUs in our cluster, we are currently using a conventional batch scheduler, namely PBS Pro. PBS Pro can handle a GPU as a custom resource and can allocate the exact number of GPUs specified by the user. However, PBS Pro cannot automatically adjust the number of GPUs to be allocated in response to fluctuations in the system load. We are unaware of a batch scheduler product that controls resource allocation in order to leverage the flexibility of parallelism in DL tasks. In the following sections we investigate the source of the flexibility in DL tasks on GPUs.

A. Automatic Distributed Deep Learning

When some GPUs in a cluster are idle, they can be exploited to speed up training. Distributed DL using multiple GPUs has attracted attention from both researchers and practitioners. There are two major approaches for distributed learning: data parallelism and model parallelism. Data parallelism is currently the dominant approach to distributed DL. Although a program requires a significant implementation effort to run on a distributed environment in general, most modern DL frameworks such as Keras[1], Chainer [2], Torch[2], PyTorch[3], and TensorFlow [3] allow users to employ data parallelism easily.

The scalability of distributed DL depends on the type of NN. For data parallelism, distributed GPUs communicate with each other to synchronize the network parameters. This synchronization is a major limiting factor of scalability. Therefore, NNs with a limited number of parameters (e.g., ResNet and Inception-v4 for image classification) can be almost linearly scaled. Recently, data parallelism has been used to scale the training of an image classification network with up to thousands of compute nodes [4]. On the other hand, the scalability of NNs with more parameters such as for language processing is limited.

To evaluate the scalability of our language processing tasks, we have conducted a benchmark using OpenNMT, which is a typical implementation of natural language translation with Torch. As shown in Fig. 1, we have found that the speedup curve of OpenNMT is sublinear and non-decreasing. This is an important property when choosing an algorithm for task scheduling.

B. Fractional Allocation of GPUs

In our environment, users submit many tasks for parameter surveys. Most of these tasks are for training neural networks with complicated structures like Multi-column Convolutional NN (MCNN) [5] and Sequence-to-sequence [6] extended for language processing [7], [8]. A profile of their execution processes shows that a GPU's computational resources (e.g., streaming multiprocessors in NVIDIA's GPUs) have idle time due to the execution time of the host code and data exchange between the host and the GPU. Additionally, execution processes typically consume only around 20% of a GPU's device memory. Therefore, it is possible to run multiple tasks on one GPU, which may increase the utilization of the GPU's processors.

To confirm these assertions, we have conducted a benchmark using MCNN where the same n tasks are simultaneously executed on the same GPU. Figure 1 plots the average processing speed as #GPUs $= 1/n$. The concurrent execution of n tasks leads to a relative processing speed of each task higher than $1/n$. In other words, a task that processes n units of data per unit time when executed alone on a GPU will process more than one unit of data when executed together with $n-1$ other tasks on a GPU. Hence, the speedup curve of MCNN is also non-decreasing sublinear in the sense that increasing the fraction of GPUs (e.g., from $1/3$ to $1/2$) does not increase the speed linearly (e.g., by $3/2$ times).[4]

[1]https://keras.io/
[2]http://torch.ch/
[3]http://pytorch.org/
[4]This might be considered "subadditive", but we also use "sublinear" in this sense in this work.

III. Problem Statement

A. System Model

Our target is a homogeneous cluster of compute nodes under the control of a centralized resource manager or scheduler. The nodes are connected via a standard network (e.g., Ethernet or InfiniBand). Network-attached storage is accessible from any node. Each node is equipped with one or more GPUs. Tasks are submitted by users at arbitrary times and are queued in the scheduler. The arrival times of future tasks are unknown. When a new task arrives or a running task completes, the scheduler allocates the available GPUs to the tasks in the queue. If the available GPUs are insufficient, tasks remain queued. This configuration is classified as a batch-style online scheduling problem.

In addition to those of conventional batch scheduling system, users give their tasks extended attributes:

- $p_k^{max} \in \mathbb{N}$: Maximum number of GPUs to be allocated to task k.
- $p_k^{min} \in \{\frac{1}{n} \mid n \in \mathbb{N}\}$: Minimum fraction of GPUs to be allocated to task k. Fractional number $1/n$ means the task is allowed to run concurrently with $n-1$ other tasks on a GPU.
- $m_k^{req} \in \mathbb{N}$: Amount of GPUs device memory in bytes that task k task requires.

Given these attributes, the scheduler allocates GPUs to each task so that the allocated number of GPUs is between p_k^{max} and p_k^{min}, and the total amount of device memory required by the allocated tasks on a GPU does not exceed the amount of device memory provided by the GPU.

B. Task Model

We focus on DL tasks that train NNs. Depending on the number of GPUs used, a DL task typically takes several hours or days to train an NN. The exact runtime of a task may or may not be predicted prior to execution. (Herein both scenarios are tested.) The number of GPUs a task runs on can be adjusted just before execution (e.g., via a command-line option). This type of task is referred to as moldable. Additionally, a task may be suspended and resumed later on a different number of GPUs or even on a different group of nodes. This type of task is referred to as malleable. When suspended/resumed, the intermediate results are saved/loaded to/from shared storage.

C. Performance Metrics

Our goal is to identify an effective scheduling mechanism to obtain task results as quickly as possible. One user may train a predefined NN with huge data by running a task for several days, whereas another user may perform a parameter survey by running a bag-of-task consisting of thousands of smaller tasks. To capture those varying users' happiness, we employ the following metrics:

- Flow time (or turnaround time) t_k^{flo}: The time between a task's arrival and its completion. Shorter is better.

$$t_k^{flo} = t_k^{cmp} - t_k^{arr}$$

- Stretch (or slowdown) s_k: The task's flow time divided by its computational volume. Smaller is better.

$$s_k = \frac{t_k^{flo}}{v_k}$$

- System utilization u: Total computational volume of all the tasks divided by the number of GPUs and the makespan. Larger is better.

$$u = \frac{\sum_k v_k}{|G|(\max_k(t_k^{cmp}) - \min_k(t_k^{arr}))}$$

Notations: t_k^{arr} denotes the arrival time (when submitted by a user) of task k, t_k^{bgn} denotes the start time of task k, t_k^{cmp} denotes the completion time of task k, v_k denotes the computational volume of task k representing the task's runtime when run alone on a single GPU, and G denotes the set of GPUs.

As the majority of tasks in our workload belong to some bag-of-task, we introduce the notion of a *job*. A job consists of tasks that start during or within one minute after finishing the execution of another task of the same user. We will calculate flow time and stretch in both per-task and per-job manner.

IV. Scheduling Algorithms

As our purpose of this work is not to propose a novel scheduling algorithm, we employ four simple algorithms that exploit tasks' moldability/malleability in a straightforward manner. The four algorithms are classified into two categories: non-clairvoyant and clairvoyant. The non-clairvoyant algorithms do not use any knowledge about the runtime of each task. In contrast, the clairvoyant algorithms use the runtime of each task as prior knowledge, which can be obtained from the history of similar tasks or from user-defined walltimes.

A. Non-clairvoyant Algorithms

For the non-clairvoyant setting, we adopt the principle of the Equipartition strategy [9], which distributes available resources equally to all tasks. The rationale is that Equipartition maximizes the overall throughput when the tasks' speedup curve is non-decreasing and sublinear, which is the case for our DL tasks. (See Section 2.)

Equipartition assumes a continuous amount of resources, which can be arbitrarily partitioned like CPU cycles or memory capacity in a single node. This is not the case for GPUs because the amount of computational resource allocated to each task is not externally controllable[5]. For example, when three processes are running on one GPU, each process receives approximately $1/3$ of the streaming multiprocessors due to the GPUs warp scheduler, which cannot be controlled externally. On the other hand, when a single task uses multiple GPUs, it is desirable for the task to use these GPUs exclusively. For example, if Task #1 uses GPU #1 and GPU #2 while Task #2 simultaneously uses GPU #2, then it is difficult for Task #1 to synchronize its processes on both GPUs.

[5]We assume NVIDIA's Maxwell and Pascal architectures.

To support the fractional allocation mentioned above, we treat a GPU's computational resource as if it is continuously partitionable. We then modify the Equipartition algorithm so that each task receives either $1/n$ shared GPU or n dedicated GPUs, where $n \in \mathbb{N}$. "A task receives $1/n$ shared GPU" means that the task runs on a GPU together with up to $n-1$ other tasks running on the same GPU. Hereafter we use two terms "processor" and "the number of GPUs" interchangeably. Both mean the amount of computational resources of GPU(s). For example, "2 processors" means 2 dedicated GPUs and "0.25 processors" means $1/4$ of a shared GPU. We also consider the amount of GPUs' device memory as a constraint so that the total amount of allocated memory does not exceed the physical capacity. Resources other than GPUs (e.g., CPUs, host memory, etc.) can also be considered. However, other resources are beyond the scope of this study because GPUs are the most competitive resource in our environment.

Notations: Let $F = \{\frac{1}{n} \mid n \in \mathbb{N}\}$ be the set of unit fraction, Q be the set of queued tasks, G be the set of all GPUs, and A be the set of vacant GPUs without running tasks. For each task k, let p_k^{max} and p_k^{min} be the maximum and minimum amount of processor requested, respectively, p_k^{alc} be the amount of processor allocated, and m_k^{req} be the amount of device memory requested. Recall that $p_k^{max} \in \mathbb{N}$ and $p_k^{min} \in F$. For each GPU i, let R_i be the set of running tasks, $p_i^{avl} = 1 - \sum_{k \in R_i} p_k^{alc}$ be the available amount of processor, and m_i^{avl} be the available amount of device memory.

The algorithms are defined as follows.

- *Moldable Equipartition:*
  In the moldable setting, at each scheduling event when a new task arrives or a running task terminates, the scheduler scans the queued tasks in the arrival order and allocates available resources to each task as described below. Running tasks and already-allocated resources remain intact. Unless stated in the context, "allocate" implies that a specified amount of processor is allocated to task k from the first-found GPU i that meets $m_k^{req} \leq m_i^{avl}$.
  - If $\sum_{k \in Q} p_k^{min} \geq \sum_{i \in G} p_i^{avl}$, then allocate p_k^{min} to each task $k \in Q$.
  - If $\sum_{k \in Q} p_k^{max} \leq |A|$, then allocate p_k^{max} to each task $k \in Q$.
  - If $|Q| \leq |A|$, then:
    1) First, allocate 1 vacant GPU to each task $k \in Q$.
    2) Next, distribute the remaining vacant GPUs using the D'Hont method, regarding the available GPUs as parliamentary seats and the tasks as candidates, assuming each task k has p_k^{max} votes.
  - Otherwise:
    1) For each task k, choose GPU i that has the smallest number of running or pre-assigned tasks, and pre-assign k to i if $p_k^{min} \leq p_i^{avl}$.
    2) Let R'_i be the set of pre-assigned task on GPU i.
    3) For each task k pre-assigned on GPU i, choose the largest possible value $p_k \in F$ that meets $p_k \leq p_i^{avl}/|R'_i|$, and allocate p_k from i.
- *Malleable Equipartition:*
  In the malleable setting, at each scheduling event, the scheduler (re-)assigns both queued and running tasks to all resources, including already allocated ones. The algorithm is identical to that of the moldable setting, except that both queued and running tasks are considered. To see the maximum effect of malleability, re-allocation is unrestricted so that the scheduler can re-allocate a different amount of processors in a different set of nodes from where a task was once assigned. For example, a running task is suspended if the scheduler re-allocates zero processor or a running task shrinks/expands if the scheduler re-allocates less/more amount of processor. Moreover, a suspended task resumes if the scheduler re-allocates some processors and a running or suspended task migrates if the scheduler re-allocates processors in a different node. Hereafter we collectively call such re-allocations *preemption*.

B. Clairvoyant Algorithms

For the clairvoyant setting, we adopt a proportional allocation strategy, which allocates each task an amount of processor in proportion to the computational volume of the task, where the computational volume is the runtime of the task when run alone on one GPU. The rationale is that the proportional allocation can achieve the shortest makespan because all the tasks should complete simultaneously, provided that they have linear speedup curves and the resource is continuously partitionable. Although our system does not exactly meet these conditions, we try to allocate GPUs as proportional as possible.

Notations: Let v_k^{rem} be the remaining computational volume of task k and $f(v,p)$ be a mapping function from a task's computational volume v and an allocated amount of processor p to its runtime.

The algorithms are defined as follows.

- *Moldable Proportional:*
  In the moldable setting, at each scheduling event, the scheduler allocates available resources to queued tasks as described below. Running tasks and already-allocated resources are kept intact.
  - If $\sum_{k \in Q} p_k^{min} \geq \sum_{i \in G} p_i^{avl}$, then allocate p_k^{min} to each task $k \in Q$.
  - If $\sum_{k \in Q} p_k^{max} \leq |A|$, then allocate p_k^{max} to each task $k \in Q$.
  - Otherwise, calculate a target makespan t^{trg} as
    $$t^{trg} = \frac{\sum_{k \in Q} v_k}{\sum_{i \in G} p_i^{avl}}.$$
    Then for each task $k \in Q$ in a descending order of v_k^{rem}, choose a value $p_k \in \mathbb{N} \cup F$ so that $f(v_k^{rem}, p_k)$ becomes the closest value to t^{trg}, and allocate p_k.
- *Malleable Proportional:*
  In the malleable setting, the scheduler (re-)assigns both queued and running tasks to all resources using the same algorithm as above.

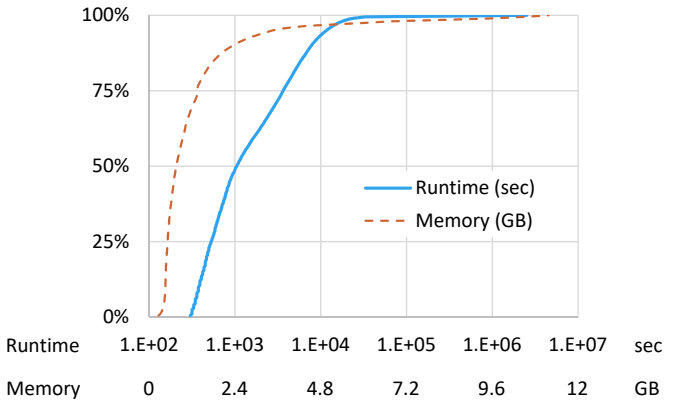


Fig. 2. Cumulative distribution of runtime and memory consumption for our GPU workload. Note that the runtime uses a logarithmic scale.

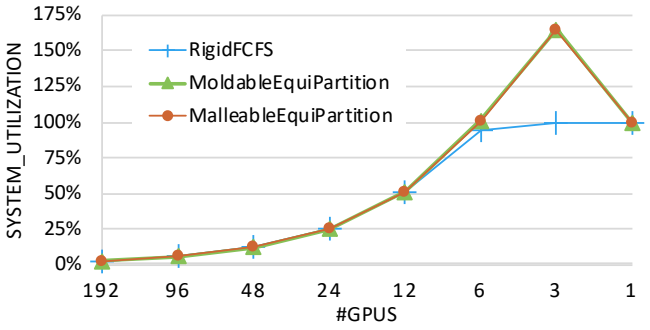


Fig. 3. Simulated system utilization for our GPU workload.

V. Experiments with Our GPU Workload

A. Workload

We use a real-world workload trace of our in-house Linux cluster system with 42 NVIDIA Tesla M40 GPUs, where each GPU has a 12 GB device memory, observed for six months (from August 2017 through January 2018). Observations are performed by a daemon program that collects the information about each process running on each GPU every 15 seconds. A process observed on a GPU for more than 300 seconds is regarded as a task. Task k is described by start time t_k^{bgn}, completion time t_k^{cmp}, concurrency c_k, and memory consumption m_k. c_k and m_k represent the number of processes observed concurrently with the task on a GPU and the maximum amount of GPU device memory consumed by the task, respectively. The computational volume $v_k = (t_k^{cmp} - t_k^{bgn})/c_k$, which represents the runtime of task k when run alone on one GPU, is calculated for each task. The workload includes 14 unique users and 77677 tasks (2317 jobs). The concurrency is 16 at maximum and 3.95 on average. No task uses more than one GPU. Figure 2 shows the runtime and memory consumption distributions.

B. Simulator

We have developed a discrete-event simulator that implements the moldable/malleable scheduling algorithms. The simulator takes as input a number of nodes and a workload. The smaller the number of nodes, the more congested the system is. For all the tasks we set $(p^{min}, p^{max}) = (\frac{1}{4}, 4)$ for the moldable/malleable algorithms and $(p^{min}, p^{max}) = (1, 1)$ for the rigid algorithms. When allocating more or less than one GPU to a task k, the simulator calculates the task's remaining runtime $f(v_k^{rem}, p_k^{alc})$ according to the speedup curve of our real applications obtained by our benchmarks (Fig. 1). Though our benchmark includes only two applications, it is sufficient for our experiments because 97% of the total computational volume in our workload is generated by either MCNN-based or OpenNMT-based programs, and the latter may not share a GPU because it usually requires more than 90% of GPU's device memory.

With the malleable algorithms the scheduler may preempt task k and add a fixed amount of time t^{pmp} to its remaining runtime. We assume $t^{pmp} = 0$ in Sections V-D and V-E, and $t^{pmp} \in \{0, 150, 300\}$ seconds in Section V-F. We allow the scheduler to preempt task k only if $v_k^{rem} > 300$ seconds in order to prevent excessive preemptions.

As a baseline for comparison, the scheduler also implements two rigid scheduling algorithms that allocate the exact number of GPUs as requested.

- *Rigid FCFS:*
  A first-come-first-serve algorithm that allocates p_k^{max} to each task $k \in Q$ in the arrival order.
- *Rigid Shortest:*
  A shortest-task-first algorithm that allocates p_k^{max} to each task $k \in Q$ sorted by v_k in an ascending order.

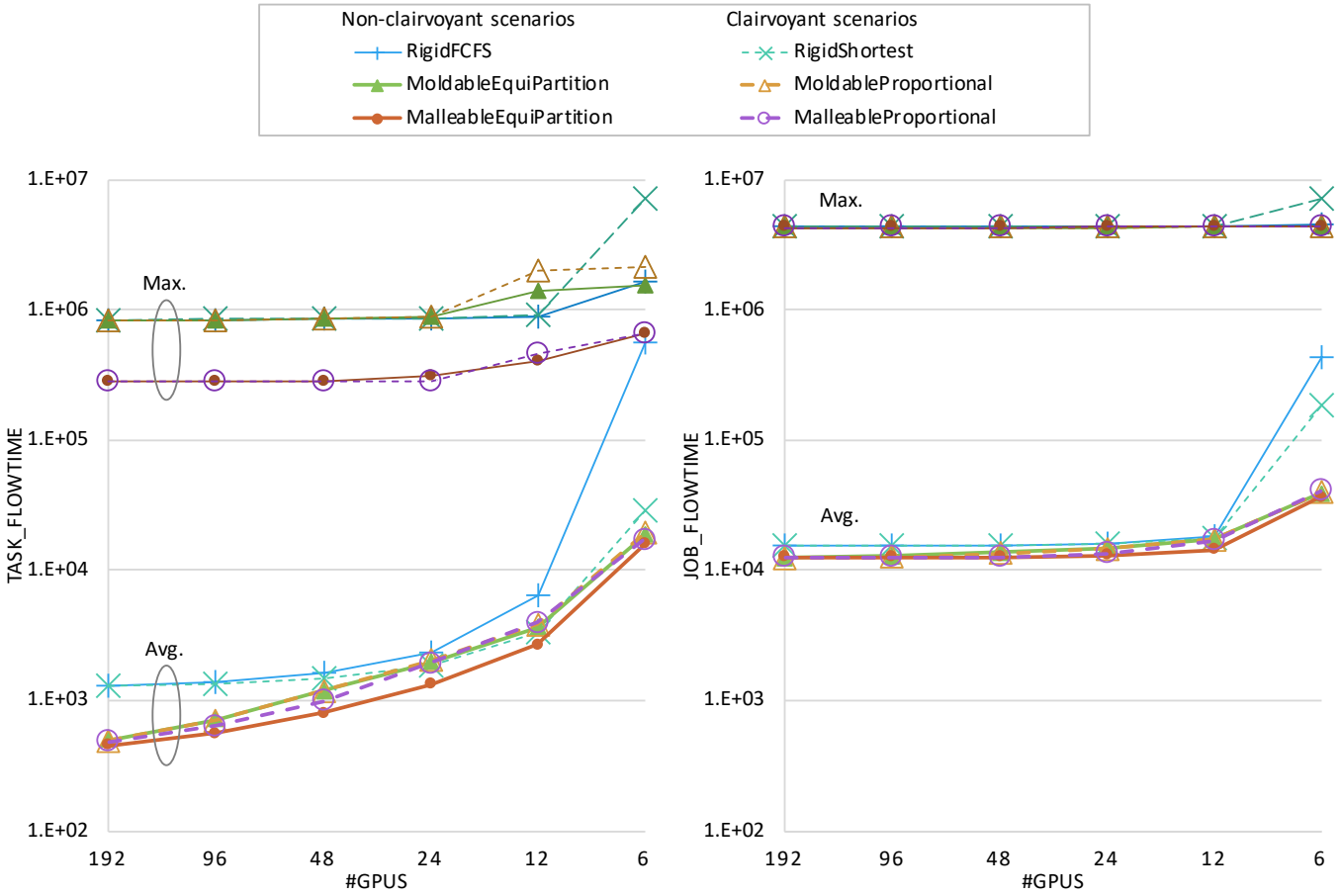
C. System Utilization

Figure 3 shows the system utilization as a reference in the following discussions. Only the results for the non-clairvoyant algorithms are shown because the clairvoyant algorithms yield almost identical results to their non-clairvoyant counterparts. The simulated system becomes saturated with six GPUs and the rigid algorithms keeps the system at 100% utilized with less GPUs. On the other hand, the moldable/malleable algorithms exploit the system beyond 100% because the fractional allocation of GPUs improves the overall throughput due to the sublinear speedup of our tasks. In the following figures we omit the plots for $|G| < 6$ as such a congested system is impractical.

D. Results for Non-clairvoyant Scenarios

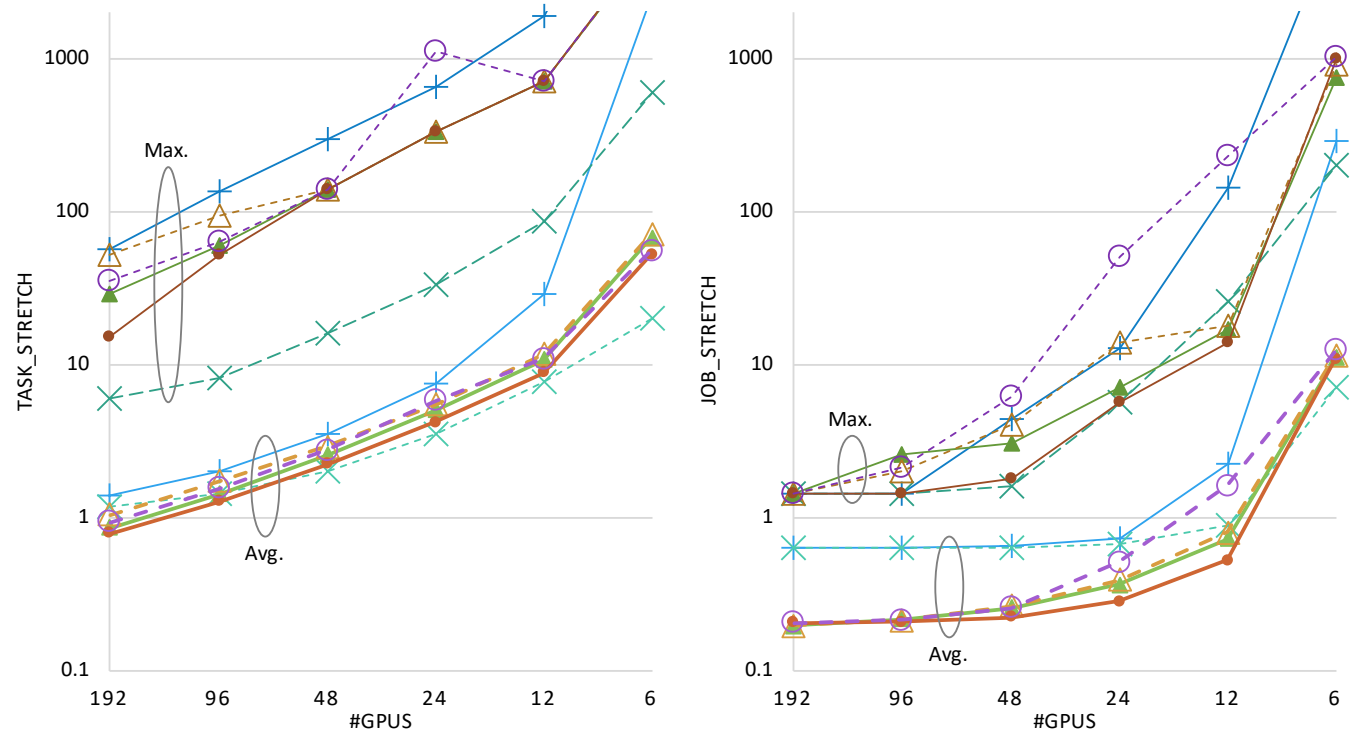
First, let us focus on the solid lines in Fig. 4, which represent the non-clairvoyant scenarios where the scheduler does not use prior knowledge about the tasks' runtime. We compare the schedules generated by the Moldable Equipartition and Malleable Equipartition algorithms to that by the Rigid FCFS algorithm.

The upper half of Fig. 4 shows the flow time in seconds. The results for the per-task flow time indicate that moldability and malleability improve the average flow time up to 96.7% and 97.2% over the rigid schedule, respectively, when the system is fully loaded ($|G| = 6$) (Fig. 4a). These improvements are



(a) Flow time per task.

(b) Flow time per job.



(c) Stretch per task.

(d) Stretch per job.

Fig. 4. Simulation results for our GPU workload.

at least 15.1% and 42.5%, respectively. Although moldability does not improve the maximum flow time in most cases, malleability improves it by up to 66.6%. The reason is that once task k is assigned to a lesser amount of processors than p_k^{max}, only the malleable scheduler allows the task to expand to p_k^{max} as more processors become available.

The per-job flow time displays a smaller difference between the algorithms (Fig. 4b). However, moldability and malleability still improve the average per-job flow time at least 6.7% and 17.5%, respectively. The maximum per-job flow time is almost the same for all schedules.

The lower half of Fig. 4 shows the stretch. Figure 4c clearly shows the advantage of moldability and malleability. Both equally improve the maximum stretch by 50.0% through 62.5% over the rigid schedules when the system is modestly loaded ($12 \leq |G| \leq 48$). The results for the average stretch are consistent with those for the flow time. The minimum improvement brought to the average stretch by moldability and malleability is 26.7% and 36.2%, respectively. Note that a stretch can be less than 1.0, which indicates that more than one processor is allocated to a task. The per-job stretch is the jobs' runtime divided by the jobs' total computational volume (Fig. 4d). Moldability/malleability improves the maximum per-job stretch by around 90% when the system is fully loaded ($|G| = 6$). Remarkably, Malleable Equipartition achieves a maximum per-job stretch comparable or better than that of Rigid Shortest, which uses prior knowledge about the tasks' runtime. Furthermore, both Moldable Equipartition and Malleable Equipartition achieve better average per-job stretches than their clairvoyant counterparts when the system is not fully loaded ($|G| \geq 12$).

In summary, moldability provides undoubted improvements over rigid schedules to minimize stretch and average flow time. Its advantage is particularly noticeable when the system load is low or very high. Malleability may yield even better schedules, but we withhold judgment since the simulations herein assume that preemption lacks overhead. We refer the reader to Section V-F for a discussion on the effect of preemption overhead.

E. Results for Clairvoyant Scenarios

The dotted lines in Fig. 4 represent the clairvoyant scenarios, where the scheduler uses prior knowledge about the tasks' runtime. We speculated that the use of more information would improve the schedules. However, the results do not support this hypothesis. Schedules with the Moldable Proportional and Malleable Proportional algorithms are inferior or almost identical to their non-clairvoyant counterparts. For example, the average per-task flow times at $|G| = 12$ are 3820 seconds with Moldable Proportional and 3636 seconds with Moldable Equipartition. The difference is more noticeable when comparing the stretch. The reason is that our proportional algorithms often allocate a tiny fraction of processor to small tasks when the system is highly loaded. This behavior can be understood as the goal of our proportional algorithm is not to minimize the flow time but to minimize the makespan. In fact, the per-job flow time (which best represents the users' happiness) is almost identical with the Equipartition algorithms and the Proportional algorithms.

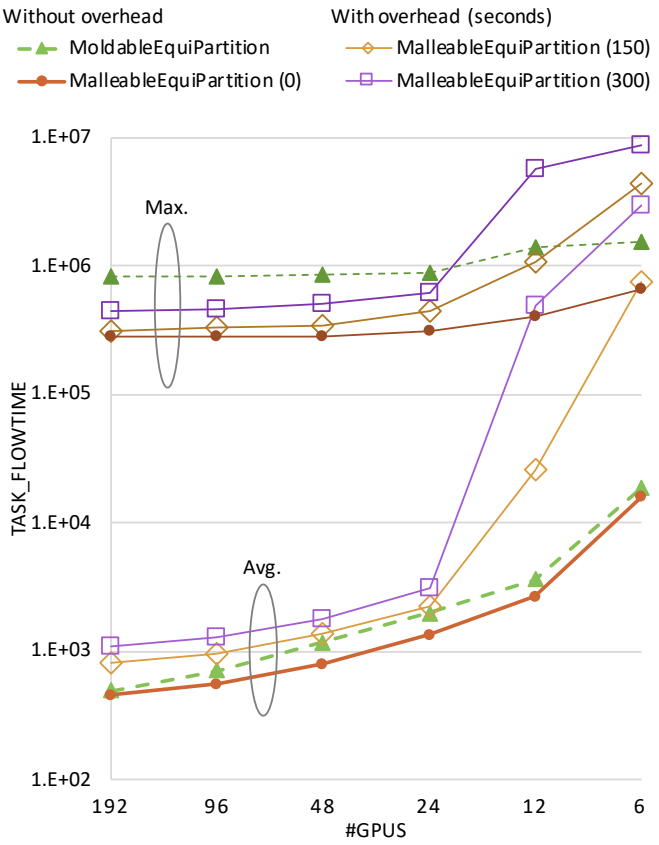


Fig. 5. Simulation results with preemption overhead for our GPU workload.

One may notice an outstanding advantage of Rigid Shortest in terms of the per-task stretch (Fig. 4c). This is not surprising because the shortest-task-first algorithm minimizes the maximum stretch of interactive tasks by prioritizing small tasks. In our use cases, however, most of the short-running tasks belong to a long-running parameter survey job, and we seldom care about the stretch of an individual task. Consequently, we introduce the per-job stretch (Fig. 4d). When comparing the maximum per-job stretch, Rigid Shortest is no longer outstanding and is often worse than the Equipartition algorithms, which do not use information about the tasks' runtime.

F. Overhead for Preemption

Preemption of a DL task involved by the malleable schedulers may take a long time especially when a task is moved from one node to another. To clarify the trade-off between the benefit of malleability and the penalty of preemption, we conducted additional simulations with varying preemption overheads.

Fig. 5 shows the per-task flow time with the Malleable Equipartition algorithm assuming $t^{pmp} \in \{0, 150, 300\}$ seconds. The results with the Moldable Equipartition algorithm is also shown for comparison. As far as the system is lightly loaded ($|G| \geq 24$), the merit of the malleable algorithm is

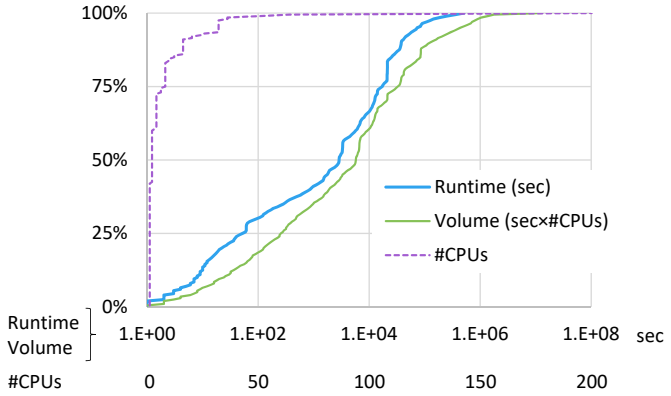


Fig. 6. Cumulative distribution of observed runtime, volume and number of CPUs for the HPC2N workload. Volume is the runtime multiplied by the number of CPUs. Note that the runtime and the volume use a logarithmic scale.

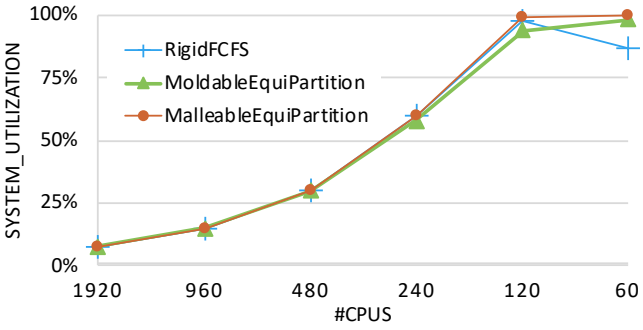


Fig. 7. Simulated system utilization for the HPC2N workload.

preserved in terms of maximum flow time. However, when the system is heavily loaded ($|G| \leq 12$), the task suffer from frequent preemptions and the benefit of malleability disappears. Therefore, a malleable scheduling algorithm must be designed with explicit consideration of empirical overhead for preemption.

There is a broad design space between non-preemptive scheduling and arbitrarily preemptive scheduling, and exploring the design space should be our future work. We refer the reader to Edmonds' work [10] for theoretical analysis of scheduling with different levels of preemption.

VI. Experiments with a Standard Workload

The previous section showed that our DL tasks would certainly profit from moldability/malleability. To evaluate the effectiveness of moldability/malleability in case of general tasks, we also perform experiments with the HPC2N workload from the Parallel Workloads Archive [11]. The reason for choosing this workload is that it contains almost complete information and many previous studies have used it. It contains 202871 tasks observed for 3.5 years (from July 2002 through January 2006) in a Linux cluster with 240 CPUs. Each task k is described by a start time t_k^{bgn}, a finish time t_k^{cmp}, and a number of CPUs p_k^{req}. We calculate a volume $v_k = (t_k^{cmp} - t_k^{bgn})p_k^{req}$ for each task. Figure 6 shows the distribution of the runtime, the volume and the number of CPUs.

Simulation is performed by our in-house simulator treating a CPU in the workload as a GPU in the simulator. We set $(p_k^{min}, p_k^{max}) = (1, 240)$ for the moldable/malleable algorithms and $(p_k^{min}, p_k^{max}) = (p_k^{req}, p_k^{req})$ for the rigid algorithms. We assume a linear speedup and no overhead for preemption as we do not have a reliable estimation of these values. Other settings and scheduling algorithms are the same as described in the previous section.

A. Results for online scheduling

Figure 7 shows the system utilization. All the scheduling algorithms (including those not shown in the figure) yields almost identical values. As the simulated system saturates with 120 CPUs, we omit the plots for 60 CPUs in the following plots.

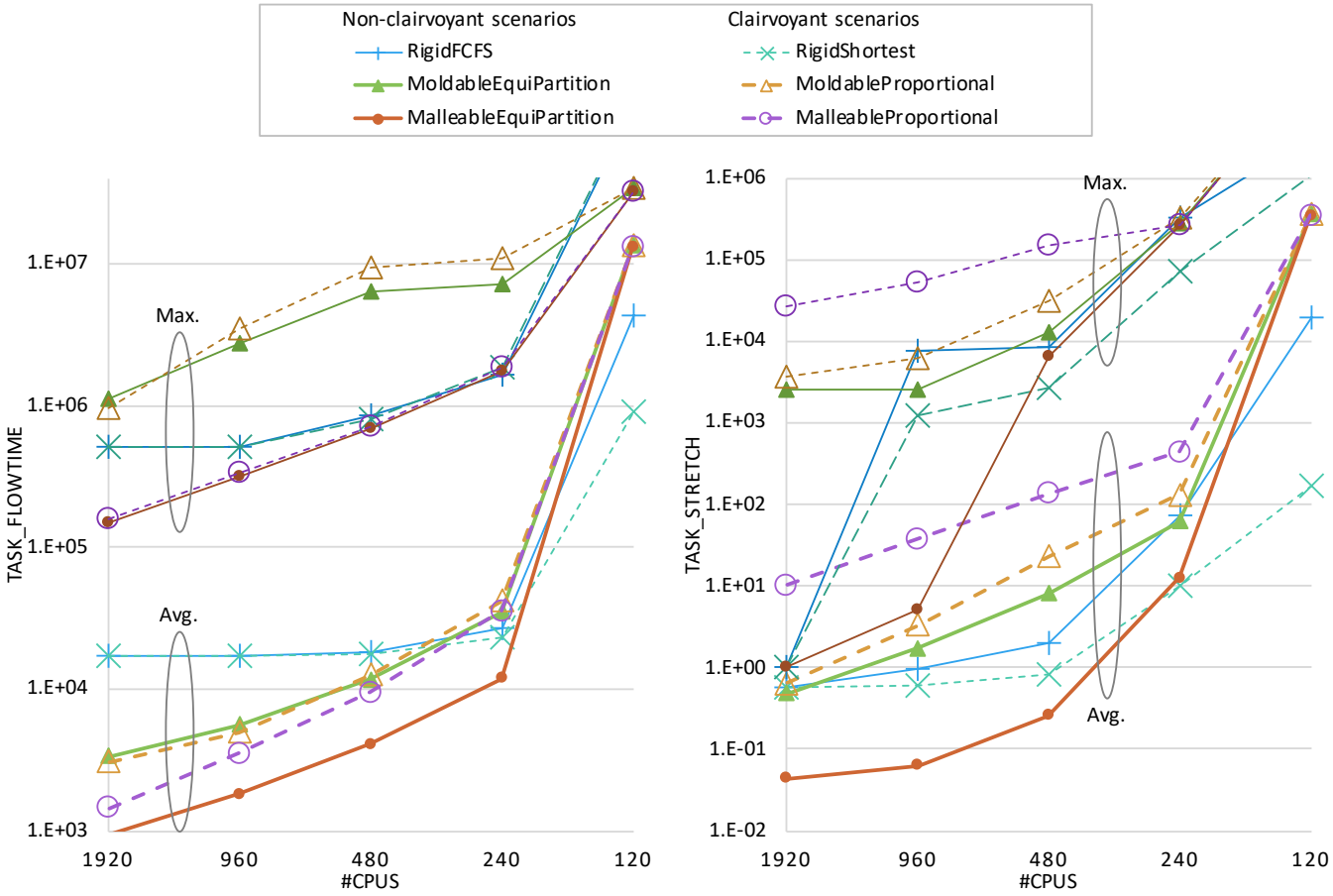
Figure 8 shows the flow time in seconds and the stretch. When the system is lightly loaded, we reach the same conclusion as before: moldability and malleability improves the average flow time compared to the rigid schedules. However, when the system is modestly or heavily loaded (#CPUs ≤ 240), the average per-task flow time of the moldable schedules become worse than that of the rigid schedule. The reason is that the tasks could not benefit from fractional allocations since we set $p_k^{min} = 1$. For example, assume a heavily loaded system and two tasks with $p_1^{req} = p_2^{req} = 2$ and $v_1 = v_2 = 4$. With the Rigid algorithm they run one after another using two CPUs and the average flow time would be $\frac{2+4}{2} = 3$; while with the Equipartition algorithm they run in parallel each using one CPU and the average flow time would be $\frac{4+4}{2} = 4$. This is an inherent disadvantage of the Equipartition algorithm in terms of average flow time. If we allow a fractional allocation and the tasks have a sublinear speedup curve, then the disadvantage could be offset by the sublinear speedup of tasks assigned to one CPU.

Overall, through the experiments with the HPC2N workload we confirmed that the schedule of general CPU tasks can also be improved by leveraging the tasks' moldability/malleability.

B. Results for offline scheduling

So far we only considered online scheduling scenarios, where the tasks arrive at their observed arrival time and the scheduler processes each task in a timely manner. To see the effect of moldable/malleable scheduling from a broader viewpoint, we also conducted an offline scheduling simulations. Herein we set $t_k^{arr} = 0$ for all k, i.e, all the tasks arrive at once.

Fig. 9 shows the results for the offline scheduling simulations for the HPC2N workload. Makespan indicates the maximum completion time of all the tasks. Plots for the clairvoyant algorithms are omitted because they yield identical results to their non-clairvoyant counterparts. As shown in the figures, the Rigid and Malleable algorithms result in similar performance. Malleable Equipartition achieves at most 2.7% shorter makespan than that of Rigid FCFS. In contrast, the



(a) Flow time per task.

(b) Stretch per task.

Fig. 8. Simulation results for the HPC2N workload.

Moldable algorithm results in worse performance than the other two. The reason is that, in the offline scenarios, there are so many tasks in the queue that the Moldable Equipartition scheduler allocates $p_k^{min} = 1$ processor to every task. This causes some large tasks remain running slowly in the end.

VII. Related Work

A. Online Scheduling

Our work is related to the theory of online scheduling problem, where a scheduler must deal with arrived tasks at hand without knowing future tasks. While the majority of traditional online scheduling studies aims at minimizing the makespan [12], we consider the flow time and its stretch as more relevant performance metrics to our use cases. In terms of average flow time, Edmonds [9] revealed that if all the tasks have a strictly sublinear speedup curve, then the Equipartition algorithm yields a 2-competitive schedule[6]. This is the reason why we employed the Equipartition approach in our non-clairvoyant scenarios. In terms of makespan, it is proven that one may compose a 2-competitive batch-style online algorithm by performing an offline algorithm repeatedly [12]. We might improve the scheduling quality of our clairvoyant scenarios by employing a more sophisticated algorithm in place of our naive proportional algorithm.

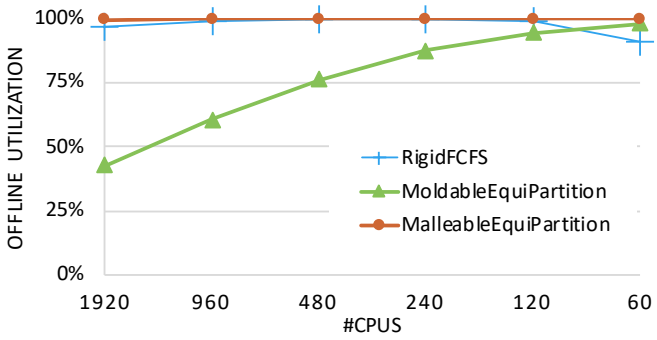
[6]A σ-competitive online schedule is at most σ times worse than an offline schedule using perfect knowledge.

B. Moldability/malleability-aware Scheduling

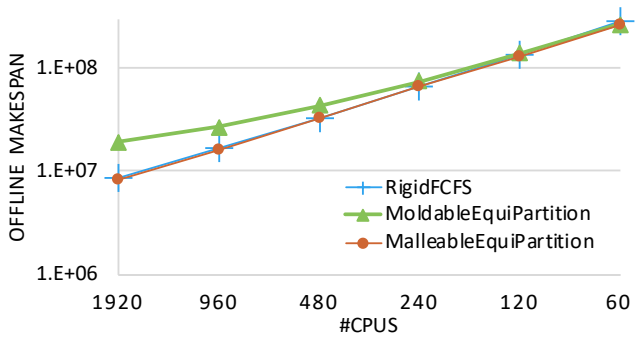
Several scheduling techniques for moldable/malleable tasks have recently been proposed with the goal of minimizing makespan [13]–[15] or stretch [16]–[18]. The key issue in designing a moldability/malleability-aware scheduler is the interface between the scheduler and tasks, which is essential for the scheduler to monitor and control the tasks. A noteworthy implementation based on Torque and Charm++ has been developed by Prabhakaran et al. [19]; however, their implementation is not available to the public. Unfortunately, we are not aware of such a scheduling product that leverages the inherent moldability/malleability of emerging DL applications built on top of high-level DL frameworks.

VIII. Conclusion

This study aims to realize an effective design for a batch scheduling system that accelerates DL tasks by exploiting their inherent flexibility (i.e., moldability and malleability). A moldable DL task can change the number of GPUs just prior to beginning its execution, while a malleable DL task can



(a) System utilization.



(b) Makespan.

Fig. 9. Offline scheduling simulation results for the HPC2N workload.

change the number of GPUs during execution. A scheduler that supports moldability/malleability may accelerate a DL task by allocating more GPUs than required for the task when some GPUs are vacant. Even if all GPUs are tasked, a scheduler may improve the utilization of computational resources by assigning more than one task concurrently to the same GPU.

To evaluate the speedup of a DL task due to moldability/malleability-aware scheduling, we conducted extensive simulations using our GPU workload trace. The results demonstrate that moldability-aware scheduling using a simple Equipartition algorithm can significantly improve the scheduling quality. For instance, if a scheduler can allocate 1–4 GPUs to a task or assign 1–4 tasks to a GPU, then the average flow time of DL tasks is shortened by 15.1%–96.7% compared to a rigid FCFS schedule in which one GPU is allocated to each task. The benefit of malleability-aware scheduling may be offset by the overhead for preemption. With our proportional allocation algorithm, the use of prior knowledge about the tasks' runtime does not improve the scheduling quality when moldability or malleability is available.

Our future goal is to establish a computing environment that accelerates DL tasks as much as possible by maximizing the use of GPUs without users' effort. Our next step is to co-design a scheduler and DL framework so that they can exchange the necessary information for the moldability/malleability-aware scheduling such as the supported minimum/maximum number of GPUs, estimated runtime on different number of GPUs, etc.

References

[1] D. G. Feitelson and L. Rudolph, "Toward convergence in job schedulers for parallel supercomputers," in *Job Scheduling Strategies for Parallel Processing*, 1996, vol. 1162/1996, no. Section 5, pp. 1–26.

[2] S. Tokui, K. Oono, S. Hido *et al.*, "Chainer: a next-generation open source framework for deep learning," in *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, 2015.

[3] M. Abadi, A. Agarwal, P. Barham *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," 2016. [Online]. Available: http://arxiv.org/abs/1603.04467

[4] T. Kurth, M. Smorkalov, J. Deslippe *et al.*, "Deep learning at 15PF," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC17)*, 2017, pp. 1–11.

[5] D. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2012)*, jun 2012, pp. 3642–3649.

[6] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," pp. 3104–3112, 2014.

[7] J.-H. Oh, K. Torisawa, C. Kruengkrai *et al.*, "Multi-Column Convolutional Neural Networks with Causality-Attention for Why-Question Answering," in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining (WSDM 2017)*, 2017, pp. 415–424.

[8] C. Kruengkrai, K. Torisawa, C. Hashimoto *et al.*, "Improving Event Causality Recognition with Multiple Background Knowledge Sources using Multi-Column Convolutional Neural Networks," in *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI-17)*, 2017, pp. 3466–3473.

[9] J. Edmonds, "Scheduling in the dark," *Theoretical Computer Science*, vol. 235, no. 1, pp. 109–141, mar 2000.

[10] J. Edmonds, D. D. Chinn, T. Brecht, and X. Deng, "Non-Clairvoyant Multiprocessor Scheduling of Jobs with Changing Execution Characteristics," *Journal of Scheduling*, vol. 6, no. 3, pp. 231–250, 2003.

[11] D. Feitelson, "Parallel Workloads Archive," 2005. [Online]. Available: http://www.cs.huji.ac.il/labs/parallel/workload/

[12] J. Sgall, "On-line scheduling," in *Online algorithms*, 1998, pp. 196–231.

[13] G. Sabin, M. Lang, and P. Sadayappan, *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, E. Frachtenberg and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4376.

[14] R. A. Dutton and W. Mao, "Online scheduling of malleable parallel jobs," *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pp. 136–141, 2007.

[15] R. Bleuse, S. Hunold, S. Kedad-Sidhoum, F. Monna, G. Mounié, and D. Trystram, "Scheduling Independent Moldable Tasks on Multi-Cores with GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2689–2702, 2017.

[16] S. Muthukrishnan, R. Rajaraman, a. Shaheen, and J. Gehrke, "Online scheduling to minimize average stretch," *40th Annual Symposium on Foundations of Computer Science*, vol. 34, no. 2, pp. 433–443, 1999.

[17] E. Saule, D. Bozda, and U. V. Catalyurek, "A Moldable Online Scheduling Algorithm and Its Application to Parallel Short Sequence Mapping," in *Ipdps 2010*, 2010, pp. 93–109.

[18] M. Stillwell, F. Vivien, and H. Casanova, "Dynamic fractional resource scheduling for HPC workloads," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, no. Vm. IEEE, 2010, pp. 1–12.

[19] S. Prabhakaran, M. Neumann, S. Rinke, F. Wolf, A. Gupta, and L. V. Kale, "A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications," *Proceedings - 2015 IEEE 29th International Parallel and Distributed Processing Symposium, IPDPS 2015*, pp. 429–438, 2015.