

Title	マルチノード水冷GPUクラスタにおけるファシリティ資源を考慮した計算制御手法の検討
Author(s)	高橋, 亮真
Citation	
Issue Date	2025-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/19830
Rights	
Description	Supervisor: 篠田 陽一, 先端科学技術研究科, 修士 (情報科学)

修士論文

マルチノード水冷 GPU クラスタにおけるファシリティ資源を
考慮した計算制御手法の検討

高橋 亮真

主指導教員 篠田 陽一

北陸先端科学技術大学院大学
先端科学技術専攻
(情報科学)

令和7年3月

Abstract

In recent years, GPGPUs, which use GPUs to accelerate parallel computing, have become widespread, and the demand for large-scale GPU clusters in data centers is expanding in response to the AI boom. However, in computer clusters using GPUs, due to the nature of the workload, a small number of jobs occupy the majority of nodes in tasks such as machine learning and model training, making it difficult to design facilities based on traditional power design using statistical multiplexing. Therefore, the current options are to prepare expensive, large-capacity facilities to match the maximum power consumption of the GPU cluster, or to reduce the scale of the cluster, which is a major obstacle to the spread of GPU clusters. In this study, we propose control using scheduling as a method to efficiently operate GPU servers even in environments where such large-capacity facilities such as power supplies cannot be secured. Evaluation was performed using simulations. The reason for limiting the title of this study to water cooling is to simplify the simulation, as it would be necessary to delve into heat exhaust and airflow issues that are unrelated to the main topic if such large-scale facilities were used. This scheduler uses two approaches to improve operating efficiency with limited facilities resources in order to utilize as many computing resources as possible even in situations where such large-scale facilities resources cannot be secured. The first is power-based scheduling based on estimated power. Generally, most of the power consumption in a server is semiconductors such as CPU, GPU, and memory, and it is usually the device driver that controls these semiconductors to operate in a power-saving manner, and the content that the scheduler can do is limited. Therefore, the policy was to maximize the consumption of the given power budget rather than to operate with reduced power. Specifically, jobs are scheduled to reach the maximum power limit based on the estimated power consumption based on pod requests. However, even if the power is maximized, there is not enough power to fully operate another server in the situation, so we thought that there is a possibility that the computation nodes will be insufficient and task execution will be congested. Here, as the second approach, we decided to improve the operating efficiency and prevent job starvation due to blocking, referring to the "Tiresias" scheduler and the scheduler of K and Fugaku. Specifically, we adopted backfill and round-robin scheduling to improve the operating efficiency of the cluster and prevent deterioration of turnaround time and execution time.

The evaluation was performed assuming an NVIDIA DGX H100 with a maximum power consumption of 10.2kw and a kubernetes environment. The reason for selecting the DGX H100 server is that it is an NVIDIA product popular for AI HPC due to its convenience and ease of development, and that it has a common configuration for high-end GPU servers. The reason for selecting k8s software is that it is widely adopted in GPU clusters of companies in Japan and overseas due to its high fault tolerance and convenience, and is suitable as software for general GPU clusters. In addition,

the specification was set to evacuate preemption caused by round-robin to the main data cache SSD because evacuation consumes a large amount of capacity, including the contents of the working memory.

For the evaluation, trace data from cluster-trace-gpu-v2023 of the Alibaba Cluster Trace Program published by Alibaba in China was used. This data is trace data of a Kubernetes GPU cluster by Alibaba, and was adopted because it is considered to be close to the actual workload environment of the Kubernetes cluster previously determined. In the evaluation results using Cluster Trace data, the power constraints of the data center were limited to 45kw and 60kw, and the proposed scheduling method was applied to the GPU cluster. For comparison, a conventional method was used with a scheduler that uses only computational resources. Simulation results showed that the proposed method had a 92% faster turnaround time and 5.8% faster execution completion time under 45kW conditions. In addition, a comparison under 60kW with the power of one DGX H100 added showed that the turnaround time was 38.2% faster and the execution completion time was 6.6% faster. This shows that the proposed method can reduce job blocking while maintaining a higher power usage rate, and can perform scheduling that results in faster turnaround times and execution times. In performing round-robin scheduling, we assumed that the backup destination was the cache SSD of the DGX main unit, but we also investigated the impact of the performance difference of this drive. As a result, the faster drive had a higher number of preemption attempts, resulting in slower results. Similarly, an experiment was conducted in which the preemption interval was changed, and it was confirmed that the longer the preemption interval, the slower the execution completion time. In this way, the proposed method was able to demonstrate advantages in both turnaround time and execution time compared to the conventional method. However, this may not be an accurate value due to the rough moderation of the simulation (including time penalties). In the future, we hope to improve the fidelity and discover weak workloads by using more accurate moderation and trace data with different workload characteristics. This work provides a new approach to improve the operating efficiency of GPU servers in power-constrained environments and promotes the deployment of servers in such environments.

目次

第1章	はじめに	1
1.1	背景	1
1.2	目的	1
1.3	論文構成	2
第2章	先行研究	3
2.1	Gandiva: Introspective Cluster Scheduling for Deep Learning	3
2.2	Tiresias: A GPU Cluster Manager for Distributed Deep Learning	3
2.3	Gang Scheduling の切り替え間隔動的変更	4
2.4	京、富岳スケジューラ	4
第3章	提案スケジューリング方式	6
3.1	設計指針	6
3.1.1	想定される課題	6
3.1.2	解決手法	6
3.2	提案手法	7
第4章	想定環境	12
4.1	データセンタ	12
4.2	ファシリティ資源	12
4.3	サーバ	12
4.4	SSD	13
4.5	ソフトウェア	15
4.5.1	Alibaba Cluster Trace Program	15
4.5.2	事前評価	15
4.6	総括	19
第5章	評価	21
5.1	評価環境	21
5.1.1	評価軸	21
5.1.2	比較手法	21
5.2	シミュレータ	21
5.2.1	初期化	22

5.2.2	メインループ	22
5.2.3	グラフ描画	22
5.3	トレースデータの利用法	24
5.4	シミュレーション評価1	24
5.4.1	電源ファシリティ制限下での性能評価	24
5.4.2	クラスター台数制限下での性能評価	30
5.5	シミュレーション評価2	33
5.5.1	RAID	33
5.5.2	比較評価	34
5.6	シミュレーション評価3	36
5.7	評価まとめ	39
5.7.1	電力使用率の向上	39
5.7.2	ジョブのブロッキングの防止	39
5.8	考察	40
第6章	おわりに	41
6.1	展望	41
6.1.1	より正確な評価	41
6.1.2	ジョブの粒度	41
6.1.3	適切なプリエンブション	41
6.2	まとめ	42
	参考文献	43
	謝辞	46

目次

3.1	アルゴリズムフローチャート	10
4.1	DGX H100/200 System Topology [10]	13
4.2	Alibaba 無制限 推定消費電力トレース図	17
4.3	Alibaba 無制限 ターンアラウンドタイム分布図	18
4.4	Alibaba 91.8kW 推定消費電力トレース図	19
5.1	シミュレータフローチャート	23
5.2	提案手法 81.6kW 推定消費電力トレース図	25
5.3	提案手法 81.6kW ターンアラウンドタイム分布図	26
5.4	電力制御なし 81.6kW 推定消費電力トレース図	26
5.5	電力制御なし 81.6kW ターンアラウンドタイム分布図	27
5.6	提案手法 61.2kW 推定消費電力トレース図	27
5.7	提案手法 61.2kW ターンアラウンドタイム分布図	28
5.8	電力制御なし 61.2kW 推定消費電力トレース図	28
5.9	提案手法 61.2kW 6台 推定消費電力トレース図	31
5.10	提案手法 61.2kW 6台 ターンアラウンドタイム分布図	32
5.11	RRなし 61.2kW 6台 推定消費電力トレース図	32
5.12	RRなし 61.2kW 6台 ターンアラウンドタイム分布図	33
5.13	Micron SSD 61.2kW ターンアラウンドタイム分布図	35
5.14	Hynix SSD 61.2kW ターンアラウンドタイム分布図	36
5.15	クオンタム 15秒 ターンアラウンドタイム分布	37
5.16	クオンタム 30秒 ターンアラウンドタイム分布	38
5.17	クオンタム 60秒 ターンアラウンドタイム分布	38
5.18	クオンタム 120秒 ターンアラウンドタイム分布	39

表目次

4.1	キャッシュ用 SSD 性能一覧表	15
4.2	Alibaba 無制限 実行完了時間評価結果表	17
4.3	Alibaba 91.8kW 実行完了時間評価結果表	19
4.4	4 章 事前決定事項リスト	20
5.1	81.6kW 実行完了時間の評価結果表	25
5.2	61.2kW 実行完了時間の評価結果表	25
5.3	6 台 実行完了時間の評価結果表	31
5.4	SSD 毎 実行完了時間の評価結果表	35
5.5	61.2kW クォンタム時間毎 実行完了時間表	37

第1章 はじめに

1.1 背景

近年、生成 AI や科学計算分野といった大規模な並列処理において、本来コンピュータ・グラフィックスなどの画像処理を得意とするプロセッサである、GPU(Graphics Processing Unit)を利用して並列演算の高速化を行う、GPGPU(General-purpose computing on GPUs)コンピューティングが普及している。またこのような高度な並列計算をクラウドデータセンターで実行する需要も高まっており、GPUを搭載したサーバや、それを複数連携して一つのシステムとした、GPU搭載コンピュータ・クラスター(以下 GPU クラスタ)の需要が拡大している。

しかしこれら GPU クラスタは、消費電力が大きく熱密度の課題があるほか、少ないアプリケーションが計算資源を占有する、複数ノードが同じジョブを同時実行するといった特有のワークロード特性をもち、これにより従来型のデータセンターとは異なるファシリティ整備を必要としている。具体的には、従来型のデータセンターでは多くのアプリケーションが稼働することから統計多重効果を利用し、電源、冷却設備といったファシリティ資源設計は稼働率を目安に設計することで解決が行われてきたが、複数ノードが一斉に同じジョブを実行する GPU クラスタでは統計多重効果が発生しにくく、従来型の設計ではファシリティ資源が不足に陥ることがあるといった課題が存在している。この課題に対して現状としては、主に最大消費電力値に合わせた大量のファシリティを用意するか、ファシリティ規模に合わせシステム規模を小さくするといった解決策が用いられることが多く、コストや整備において大きな課題となっている。

1.2 目的

本研究では、そのようなファシリティ資源、特に電力の資源が制限された環境における GPU クラスタに対し、ソフトウェア制御を適用することによって従来手法より稼働効率を改善する手法を提案する。課題とするファシリティ資源を消費電力と定めた理由としては、現状 GPU データセンター整備において大きな課題となっているのが主に消費電力問題であること、そして発熱と消費電力は相関関係にあることなどからより重要性が高いターゲットであるとして決定した。提案した手法については、大規模な GPU サーバを稼働させられる環境がないこと、環境

があったとして制御アプリケーションに本スケジューラを実装するには技術的、時間的制約が厳しいことからトレースデータを使用したシミュレーションによってその性能、具体的には電力使用率の高さや処理完了までの時間、ターンアラウンドタイム (TAT) を基準に性能評価することとする。

タイトルに水冷 GPU クラスタ、と限定を行っている理由についてであるが、これは近年登場している GPU サーバは熱密度が高く、空冷では十分に冷却できないものが増えているほか、一般にデータセンタ環境のシミュレーションを行うとした場合、サーバ排気による排熱の干渉やエアフローといった要素に関するシミュレーションする必要性が出てしまう。しかし本研究においてこのような熱力学シミュレーション、エアフロー設計は本題ではないためこれを水冷システムであると前提することによって簡単化を行ったものである。

1.3 論文構成

本論文は6章で構成される。

1章では序論としてGPUデータセンタが抱える課題を示しながら、本研究の目的を述べる。

2章では調査した既存のスケジューリングアルゴリズムや実用例に関して述べ、提案手法への参考とした点を示す。

3章では1章で挙げた課題点に対して、それを緩和するための提案手法とそのアルゴリズムについて述べる。

4章は提案手法の評価を行うにあたり、想定した設備環境、ソフトウェア環境を説明する。

5章では4章で示した環境において、条件を変えながらスケジューリングをシミュレーションし、性能評価を行う。評価には従来型のファシリティ条件、ソフトウェア条件と提案手法との間で比較評価を行った。

最後に6章では、総括として本提案手法の今後の課題や展望について述べる。

第2章 先行研究

本章では、参考にした先行研究とその内容、提案手法の考案にあたって参考にした内容を述べる。

2.1 Gandiva: Introspective Cluster Scheduling for Deep Learning

本先行研究は、機械学習のトレーニングワークロードに特化したクラスタースケジューリング手法である。[1] 本スケジューラはジョブの詰め込み、マイグレーション、リソースモニタリングの3つの観点で効率化を図る。具体的な仕組みとしては、リアルタイムでサーバのリソースのモニタリングを行い、そのサーバで実行しているジョブのスループットが何に依存しているかを判断する。一つのクラスター内でジョブ間で資源の競合が発生しない組み合わせとなるよう、クラスター間でマイグレーションを行う。本先行研究のリアルタイムでのリソースを監視するというアイデアを参考に、提案手法においては pod の要求する CPU、GPU リソースから推定消費電力を計算し、それに基づいてジョブスケジューリングするアイデアを考案した。

2.2 Tiresias: A GPU Cluster Manager for Distributed Deep Learning

本先行研究は共有 GPU クラスタ環境において、GPU リソースと各ユーザの公平性を担保するという観点のスケジューラである。[2] 一般に、GPU クラスタにおいて深層学習ジョブなどの処理時間が長く、かつクラスター内を大規模に占有するジョブを実行した場合、短時間で済むジョブや低遅延を必要とするジョブが実行されるまで長時間待たされる場合があり、複数のユーザがいる共有 GPU クラスタにおいてそれはリソース分配が不公平であるといえる。本スケジューラはその公平性の改善を狙ったものである。特徴としては、ジョブの進行状況や占有率、経過時間、リソースの使用率を評価し、それを優先度に反映する。そして実行中のジョブより優先度が高いジョブが投入された場合、プリエンプションして後続の

優先度の高いジョブに入れ替える、といった優先度ベースでのアルゴリズムを採用している。

提案手法においては、本先行研究の公平性に対する手法を参考にし、ファシリティ資源に限られる GPU クラスタにおいても、ファシリティ資源に余裕のあるデータセンタと比較して大規模にクラスタを占有するジョブが投入された場合、短時間で処理ができるジョブがブロッキングに陥りやすくなると仮定を行った。そしてそのような状況に対して、先行するジョブのプリエンプションを行うことでブロッキング状態を防ぐことができるという基本設計とした。

2.3 Gang Scheduling の切り替え間隔動的変更

先行研究は、HPC でのジョブスケジューラに Gang Scheduling を搭載した Slurm を適用し、バッチ処理とのターンアラウンドタイムの改善、切り替えオーバーヘッドを性能評価したものである。[3]Gang Scheduling は並列動作を行っている複数のジョブを一斉にスケジューリングし、一定時間ごとにジョブを入れ替えて待ち時間の軽減を図るスケジューリング手法である。先行研究によると、実行時間が短いと判明しているジョブが投入されたタイミングでジョブの切り替え間隔を動的に短縮することで、全体のターンアラウンドタイムを短縮する効果が得られたとしている。そこで提案手法においては、本先行研究と Tiresias を参考としてターンアラウンドタイムの改善策にプリエンプションを伴うラウンドロビン制御を採用することとした。

ただし後述するが提案手法ではコンテナ環境を想定しているため、プリエンプションを実行した場合にコンテナ環境でのチェックポイント作成、ジョブの中断といった処理が発生し、一般的に Slurm 環境と比較して必要となる時間コストが多くなることが想定される。また通常、投入されるジョブが短時間で処理が完了するものであるか、長時間かかるものであるか実行前に判別することは困難である。そこで提案手法においては時間コストのオーバーヘッドは極力発生させないことを重視し、ブロッキングが発生する条件ではじめてラウンドロビン制御を実行することとした。

2.4 京、富岳スケジューラ

稼働率向上の具体例として、富士通が手掛けた理化学研究所のスーパーコンピュータ京、富岳のスケジューラについても参考にした。富士通の解説資料によると、京のスケジューラではシステム稼働率向上のために、インターコネクトへの最適化と Backfill の対応を行い、[4] 京の後継システムである富岳のスケジューラでは、さらなる IO への最適化と、backfill の改善策として後続ジョブを障害しない時間まで同じジョブを継続して実行できる制御の追加を行ったとしている。[5]Backfill

とは、計算資源の空きがある場合に大規模ジョブより先に小規模ジョブを実行し、稼働率の向上を狙う制御のことである。これらを参考に、本提案手法でも稼働率向上策として、空きノードへのジョブスケジューリングに **Backfill** を採用し、また後続のジョブがなく、ブロッキングの可能性がない場合にはラウンドロビンによるプリエンプションをやめる可変ラウンドロビン方式と定めた。

第3章 提案スケジューリング方式

本章では現状の GPU クラスタの抱える課題点を基に、それを解決するために定めたスケジューラの仕様と考案したアルゴリズムを示す。

3.1 設計指針

データセンタの抱える課題点を基に、スケジューリングアルゴリズムの方針を決定した。想定される課題と、その解決にあたって重視した内容を示す。

3.1.1 想定される課題

本研究において想定している環境は、設置されている GPU クラスタに対して施設側の電源ファシリティが不足しているデータセンタである。具体的には、設置された GPU クラスタが一斉に稼働すると、GPU ラックの列、部屋、あるいは施設といった階層において GPU クラスタに供給するだけの電力供給能力を備えていない状況を想定している。そのため、制御なしにクラスタをすべて稼働させると電力超過を起こしてしまうほか、それを防ぐために稼働数を削減した場合、ファシリティが最大限確保されているデータセンタでは実行可能な機械学習やそのモデルトレーニングといった処理が制限されたり、一般にはジョブの処理性能が低下することが考えられる。

3.1.2 解決手法

本提案手法は施設の電力資源という制約環境を想定していることは既に示した。施設の電力資源は基本的に建設後の増強は困難であるため、課題解決にはサーバ側を工夫する必要があるが、これには2つの解決手法が考えられる。1つはサーバあたりの消費電力を何らかの手段で下げること、2つめとしてはサーバによる電力使用効率を向上させる手法である。前者の手法を採用する場合、サーバの消費電力の内訳としてはハードウェア的要素、特に CPU, GPU, メモリといった半導体が大半を消費するため、これら半導体の消費電力の効率化を行うには半導体の制御を行うデバイスドライバそのもの、あるいは OS 側からの制御を工夫する必要がある。これらは開発難易度が高く、また機器によって使用される半導体とその電

力特性も異なる可能性が考えられることから、本研究においては不適當であると考えた。そのためあくまでも本研究においては、電力資源の許す限り使用可能な GPU サーバを極限まで使用し、最大限の電力使用率を得られるように制御を設計することとした。具体的には、ジョブ実行時の電力に猶予がある場合に、ソフトウェアによって計算資源の空きのみならず、電力消費量をベースに電力資源に対してジョブを詰め込みスケジューリングを行うことで電力使用率を超過せず、しかし最大限となるように調整を行う。

なるべく多くの供給電力を活用できるようにした場合でも、電力の無駄が多くなかった場合など稼働可能台数が期待ほど増加しない場合も予想される。そうした環境下においては、処理に時間のかかるジョブを複数実行すると、後続ジョブのブロッキングが発生し、すぐに終了するジョブを投入してもなかなか処理が始まらず、実行時間とターンアラウンドタイムが悪化する可能性が考えられる。そのため、後続ジョブがブロッキングに陥らないような対策を備えることとした。

決定したスケジューリング方針を以下に箇条書きでまとめて示す。

- ・ 施設、部屋、ラック列といった階層ごとのファシリティ資源の中で割り当てられた電力資源を基にスケジューリングする。
- ・ 電力を最大限使用した場合でも、大規模データセンタより実行可能ノード数が少なくなることが予想されるため、ジョブのブロッキング対策を行う。

3.2 提案手法

以上の2つの指針を基にそれぞれの対策を設計としてまとめた。

まず、電力資源の管理に推定消費電力という概念を考案し、用いることとした。これはジョブが要求する計算リソースとその計算機の消費電力を掛け合わせることで、そのジョブが消費する電力を推定するものである。具体的には CPU リソース Res_CPU とこれを実行する CPU のスレッドあたり消費電力 $CPU_Pow/CPU_THREADS$ をかけたもの、GPU リソース Res_GPU と GPU の消費電力 Pow_GPU をかけたもの、GPU サーバのインターコネクトなどを含むその他消費電力 $OTHER_Pow$ の3つを足すことで、そのジョブが実行中のサーバの消費電力を概算するものである。次に推定消費電力式 3.1 として示す。

$$Est_Pow = Res_CPU * (CPU_Pow / CPU_THREADS) + Res_GPU * GPU_Pow + OTHER_Pow \quad (3.1)$$

例えば 32 スレッド 200W の CPU 1 台と 300W の GPU を 4 台搭載するシステムに、16 スレッド、4GPU のリソース要求がされた場合の計算機部分の推定消費電力は $100W + 1200W$ で $1300W$ となる。その他消費電力にはメモリやサーバ内インターコネクトや冷却ファンなどが含まれるが、これらの消費電力は推測が困難で

あるため、最悪の条件での簡単化として GPU サーバの最大定格消費電力から CPU と GPU それぞれの分の定格消費電力を差し引いたものを設定する。式 3.2 にその他消費電力の計算式を示す。

$$OTHER_Pow = Server_TDP - CPU_Pow * numCPU - GPU_Pow * numGPU \quad (3.2)$$

続いて、ブロッキング対策として Backfill と Round Robin Scheduler を採用した。Backfill は大規模な先行ジョブが実行できない場合、後続の小さいジョブを空きノードに充填するというアルゴリズムであり、今回は推定消費電力と要求する計算資源に基づいて実行可能な後続ジョブを充填を行うことでスループットを向上させる。ラウンドロビンは投入したジョブをクォンタム時間ごとに入れ替えるスケジューリングであり、同一ジョブがクラスタを長時間占有することによる後続ジョブブロッキングを防ぐ。デメリットとしてはプリエンプシヨンのためチェックポイント作成などジョブに関係ない退避処理を行う必要があり、さらにチェックポイント以後に計算された内容がロストするため大きな時間コストを必要とする。そのため本スケジューラでは計算資源の枯渇や推定消費電力が供給電力資源いっぱいになった場合など、後続ジョブのブロッキングが発生したタイミングでのみプリエンプシヨンを実行することとした。切り替えのクォンタム・タイムは論文を参考に 30 秒とした。以下に箇条書きでまとめて示す。

- ジョブの要求リソースから、最悪のパターンの消費電力として推定消費電力を求める。
- 推定消費電力と空き計算資源の許す限り、各階層のファシリティ資源の空きに基づいて詰め込みスケジューリングを行う。
- スケジューリングに失敗した場合、当該ジョブはキューに戻し、後続の小さいジョブでのスケジューリングを試みる。(backfill)
- 実行できるクラスタが不足した場合、ブロッキングが発生したとみなしてラウンドロビン制御を有効にし、後続ジョブのブロッキングを防ぐ。

以上の設計から、スケジューラとして順に動作するようにアルゴリズムの形にまとめた。作成したスケジューリングアルゴリズムを algorithm1 に、フローチャートを次の図 3.1 に示す。

Algorithm 1 ジョブスケジューリング

```
1: 入力: Job,task,Server
2: for N in Queue_Size do
3:   Job ← Queue.pop(task)
4:   Est_Pow = Job.Res_CPU * (CPU_P/CPU_THREADS) + Job.Res_GPU *
   GPU_P + OTHER_Pow
5:   if Pow_Facility > Est_Pow then
6:     ⋮                                ▷ ここで階層ごとの電力条件を判定
7:     if Pow_Server > Est_Pow then
8:       if available_CPU >= Job.Res_CPU & available_GPU >=
   Job.Res_GPU then
9:         Scheduling Success
10:        return
11:       else
12:         Scheduling Failed
13:         Queue.put(task)
14:         RR ← True
15:         continue
16:       end if
17:     else
18:       Scheduling Failed
19:       Queue.put(task)
20:       continue
21:     end if
22:   else
23:     Scheduling Failed
24:     Queue.put(task)
25:     continue
26:   end if
27: end for
```

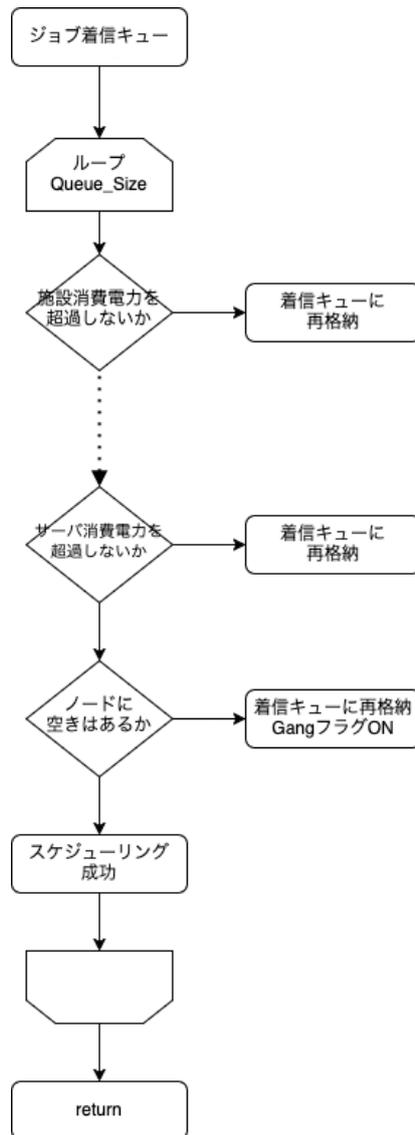


図 3.1: アルゴリズムフローチャート

スケジューラの挙動を順を追って説明する。まずジョブの着信キューはFIFO形式であり、着信したジョブから順にスケジューリングを試行する。FIFOキューに投入されたジョブは、推定消費電力を計算した後、逐次的に設備の階層ごとのリソースと、最後にノードの空き計算資源で照会して割り当て可能かスケジューリングが行われる。これをキューの長さの回数繰り返す、すべてのジョブで試行を行う。割り当て失敗したジョブはキューの最後尾に戻される。すると割り当て失敗したキューは次の時刻に着信するジョブより先に並ぶことになり、割り当て失敗しやすい要求リソースが大きいジョブが他の小さなジョブよりキューの先頭に来やすく、割り当てに成功しやすいと考えた。また先述通りGPUクラスタの空きがなくなった場合のみラウンドロビン実行することで、後続ジョブのターンアラウンドタイムの

改善と退避ペナルティの発生を最小限とする目的がある。

第4章 想定環境

本章では、提案手法をシミュレーション性能評価するにあたって、事前決定した想定環境を説明する。

4.1 データセンター

前述のように、GPU クラスターの設置を念頭に設計されたデータセンターにおいては、ファシリティリソースを大規模に確保する事例も珍しくなく、IIJ 白井データセンターキャンパスなど 50MW 級の受電能力を持つデータセンター [6] も登場している。しかし、本研究で想定されるデータセンターはそのように潤沢なファシリティ資源が整備されておらず、クラスター規模に対して電力が不足しているものとする。

4.2 ファシリティ資源

データセンターの受電能力に制限があり、これが本研究の背景にあることは既に示したが、データセンター内でも受電能力制限は存在する。一般的な施設にブレーカーがあるようにフロア、部屋、ラック列といった階層ごとにファシリティ資源には制限が存在しており、これらについても考慮しながらスケジューリングができる構造である必要がある。本研究のスケジューラにおいても機能は実装するが、あくまでも主題はデータセンターにおける受電制限を考慮するものであるため、部屋とラック列での制限消費電力はいずれも受電能力と等しく設定して実質無制限で評価を行うこととした。

4.3 サーバ

シミュレーションを行う GPU サーバとしては NVIDIA 社製、NVIDIA DGX H100 を想定した。これは、GPU コンピューティングで人気の高い NVIDIA 社のサーバであること、Dell[7]、Supermicro[8]、HPE[9] といったサーバベンダーにおいても類似した構成を持つ GPU サーバがラインナップされていることから選択した。

DGX H100 の内部構成に関して NVIDIA のユーザーガイドを引用する。[10]

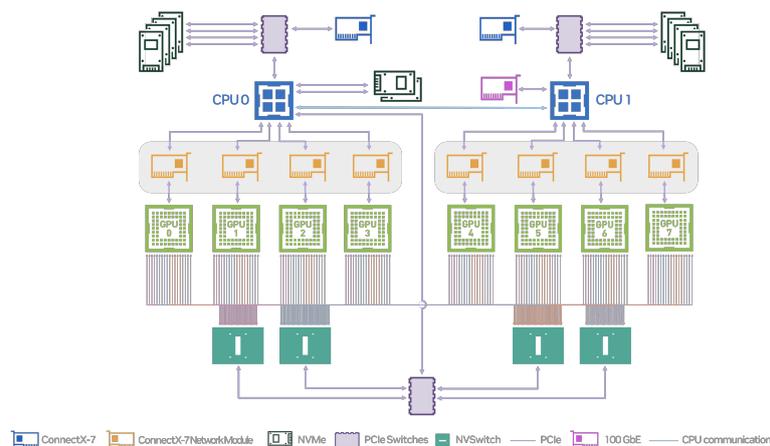


図 4.1: DGX H100/200 System Topology [10]

具体的な構成としては、Intel Xeon CPU1 台に対し、4 台の NVIDIA H100 SXM GPU と 4 台で RAID0 構成を取るデータキャッシュ用 NVMe SSD が接続され、それが 2 ユニット搭載された構造となる。OS は 2 台で RAID1 構成された 1.9TB の SSD に格納されており、片方の CPU に直結されている。主要スペックを次の表に示す。

- GPU NVIDIA H100 SXM 80GB (Upto 700W)[11] x8
- CPU Intel® Xeon® Platinum 8480C 2.0GHz 56Cores (Upto 350W)[12] x2
- OS:1.9TB NVMe SSD x2、Internal:3.84TB NVMe SSD x8 RAID0
- DRAM 2TB

DGX H100 の最大消費電力は 1 台あたり 10.2kW であり、その内訳の半分以上となる 5.6kW は 8 台の H100 SXM GPU による消費電力で消費されるものである。推定消費電力アルゴリズムにおいては、DGX H100 の最大消費電力から CPU 2 基と GPU 8 基の消費電力を差し引いた、3.9kW についてはその他消費電力として常に加算するものとした。これは、残りの電力は空冷ファンや NVLink スイッチ等に起因すると考えられるものの、それらがどのようなワークロードに対してどの程度消費電力を消費するかは推測できないため、最悪の消費電力パターンとして想定したものである。

4.4 SSD

SSD とは Solid State Drive の略称であり、フラッシュメモリを採用した、高速な補助記憶装置である。DGX サーバにおいては、ストレージノードのキャッシュと

して搭載されており、本提案手法では実行中のジョブの中断データという大きなデータを退避させる必要があるため、そこをプリエンブションしたジョブデータの中断データを格納場所として想定する。高速というものの、それは補助記憶装置としてであり、メモリより転送速度は遅いため DGX システム上ではここが一番のボトルネックとなる。

そこでデータキャッシュSSD がプリエンブションに与える影響の評価のため、DGX H100 にデータキャッシュ用に搭載されている SSD、およびそのパフォーマンスに関して調査を行ったが、手がかりは得られなかった。そこで DGX H100 が出荷開始された 2023 年第 1 四半期当時、最新かつ 3.84TB の容量を持つ U.3 SSD から表 4.1 に示す 4 製品をピックアップし、転送速度を調査した。これら SSD のカタログにおいても SK Hynix の製品を除き、AI アプリケーション向けと銘打たれているものであり、適切な想定であると判断した。CPU である Intel Xeon 8480C 自身はバス規格として PCIe Gen5 に対応しているが、2023 年初頭当時は PCIe Gen5 接続に対応したデータセンタ向け SSD は登場していなかったため、いずれも接続バスは PCIe Gen4 となっている。そのため、転送速度が 6900MB/s で頭打ちになっているものと考えられる。

表 4.1: キャッシュ用 SSD 性能一覧表

メーカー	型番	Seq.Read	Seq.Write
KIOXIA	KCM61RUL3T84 [13]	6900MB/s	4200MB/s
Micron	MTFDKCB3T8TFR-1BC15ABYY [14]	6800MB/s	5300MB/s
Samsung	MZQL23T8HCLS-00A07 [15]	6900MB/s	4100MB/s
SK Hynix	HFS3T8GEETX070N [16]	6500MB/s	4000MB/s

以上の調査結果より中央値を取り、シミュレーションにおいて想定する SSD の単体転送速度は読み込み 6850MB/s(6532.67MiB/s)、書き込み 4150MB/s(3957.75MiB/s) が妥当であると決定した。さらに NVIDIA の日本代理店である GDEP による解説記事によると、DGX H100 の標準設定ではこのキャッシュ SSD は RAID0 に設定されているが、RAID5 へ切り替えることも可能であると説明されている。本研究においては、標準の RAID0 での設定とする。[17]

4.5 ソフトウェア

ソフトウェア環境としては、kubernetes などコンテナ環境を想定した。これは環境整備の容易さ、耐障害性の高さから国内外の企業が保有する GPU クラスタなど実際の商用環境での採用実績が広く、近年の GPU クラスタの構成において一般的なソフトウェア構成であると判断した。またこれにより、シミュレータで実施する提案手法におけるプリエンブションはコンテナ単位でのプリエンブションを想定することとした。

4.5.1 Alibaba Cluster Trace Program

シミュレーション評価に使用するデータとして中国 Alibaba が github で公開している、Alibaba Cluster Trace Program の cluster-trace-gpu-v2023 のトレースデータを使用した。[18] 本データは Alibaba が保有する Kubernetes を使用したオープン GPU クラスタのトレースデータであり、想定環境で示したようなコンテナを採用したワークロード例として適切であると考えたからである。トレースデータは csv 形式で配布されており、オープンノードで作成された pod の求める CPU、GPU リソース、メモリ容量、作成時間などが格納されている。CPU、GPU リソースはミリで、時間の単位は秒で記録されている。本研究においてスケジューリングする pod は便宜上ジョブと呼称している。

4.5.2 事前評価

想定環境を決定するにあたり、本トレースデータの必要とする CPU、GPU 数、最

大想定消費電力と通常稼働に必要なサーバ台数を確認するため、事前に電力とサーバ台数が無制限となる条件でトレースデータを適用して評価シミュレーションを行った。実際にはシミュレータの給電能力、サーバ台数に無限値を設定することは不可能であるため、十分に大きな給電能力として50MW、サーバ台数は4900台、各階層の電力制限は受電能力と同等としてシミュレーションを行った。図4.2に無制限環境での想定消費電力のシミュレーション結果を、また後の評価との比較用に図4.3に同環境でのターンアラウンドタイム分布図を、実行完了時間表を表4.2に示す。なお、TATと示すものはターンアラウンドタイムの略称である。

表 4.2: Alibaba 無制限 実行完了時間評価結果表

手法	平均 TAT(秒)	実行完了時間 (秒)
RR なし	25787.9	12902961

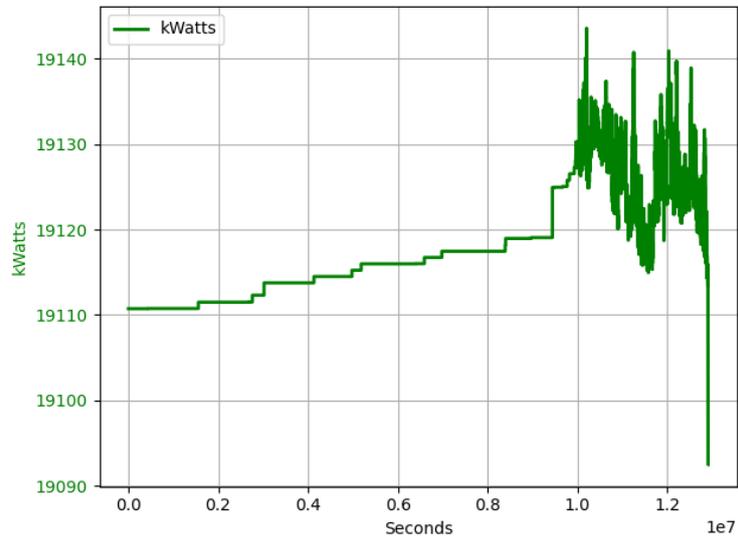


図 4.2: Alibaba 無制限 推定消費電力トレース図

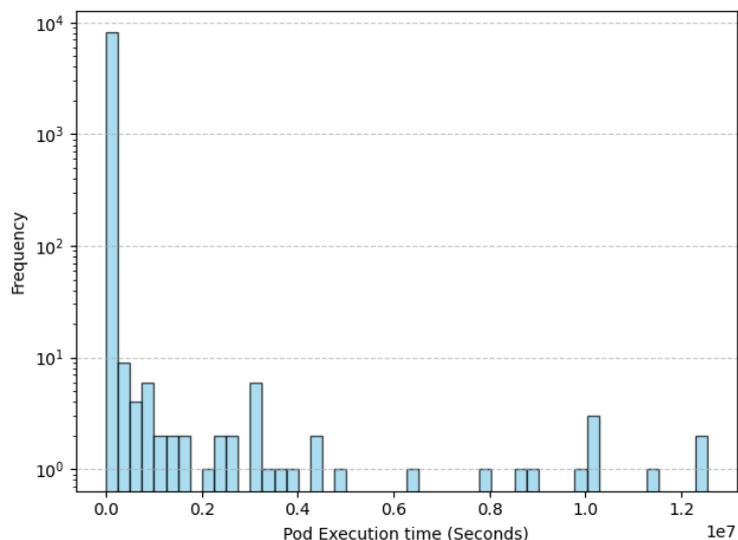


図 4.3: Alibaba 無制限 ターンアラウンドタイム分布図

実験結果として、本トレースデータの実行で CPU は最大 746 スレッド、GPU は最大 70 台必要であり、最大推定消費電力は 19.1522MW となった。DGX H100 1 台あたりの合計 CPU スレッド数は 224、GPU 台数は 8 台であるため、サーバは最低限 9 台必要である。また推定消費電力 19.1522MW のうち 19.110MW は DGX1 台につき 3.9kW 確保される、「その他消費電力」4900 台相当であるため、DGX の稼働において CPU, GPU といった計算機が消費するのは最大推定消費電力からその他消費電力を差し引いた、0.0422MW である。また平均ターンアラウンドタイム分布図からは、本トレースデータでは処理時間の短いジョブが大多数を占めていることがわかる。

必要 GPU 台数から、性能評価における標準設定として通常稼働できるサーバ台数は 9 台、電力は 91.8kw とした。念の為、91.8kw での実行結果が無制限時と同様であるか確認を行った。実行結果を図 4.4 と表 4.3 に示す。

表 4.3: Alibaba 91.8kW 実行完了時間評価結果表

手法	平均 TAT(秒)	実行完了時間 (秒)
RR なし	25787.9	12902961

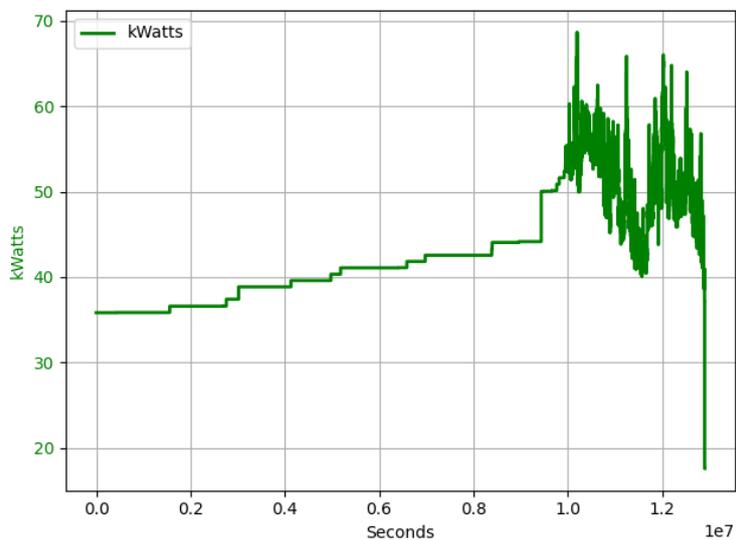


図 4.4: Alibaba 91.8kW 推定消費電力トレース図

給電能力 91.8kw、サーバ 9 台の制限において無制限と同じ時間で実行できることが確認できた。以後の実験ではこの電力条件、台数条件を変化させることで、電力が制限された環境をシミュレーションし、提案手法と従来手法での性能評価を行うこととした。

4.6 総括

本章で事前決定した内容をまとめる。決定したパラメータを表 4.4 に示す。

表 4.4: 4 章 事前決定事項リスト

- データセンタ: 受電能力 91.8kW
- GPU サーバ: DGX H100 消費電力 最大 10.2kW 9 台
- SSD: RAID0 読み込み 6850MB/s x4 書き込み 4150MB/s x4
- ソフトウェア環境: kubernetes
- トレースデータ: Alibaba cluster-trace-gpu-v2023

第5章 評価

5.1 評価環境

目的においても触れたが、本研究の目的は手法の提案と評価であること、実環境評価には複数の GPU サーバの整備が必要であること、もし環境があった場合でも本提案手法を適応した kubernetes スケジューラを実装して検証を行うことは技術的難易度が高いことからシミュレーションによって性能評価を行った。

5.1.1 評価軸

本提案手法の評価軸には、設計指針で重視した電力使用率の高さやターンアラウンドタイム、そして処理完了までの時間を基準に評価を行う。高い電力使用率を保ちつつ、処理完了とターンアラウンドタイムの両方の側面で素早く処理できるほど良いとする。

5.1.2 比較手法

スケジューリングには提案手法を適用したもの、現状のデータセンタを想定して提案手法を使用せずファシリティ条件に合わせて稼働数を絞ったものの二種類を用意し、性能比較を行う。従来手法はファシリティ資源に合わせた台数しかクラスタを設置しない方式を取り、スケジュールは計算資源の空き状況のみで判定し、Round Robin を一切行わないが、ただしスーパーコンピュータ京や富岳などで採用されていることを鑑みて、計算リソースベースの Backfill のみ有効化している。

5.2 シミュレータ

シミュレータは既存のものではなく、Python3 を用いてフルスクラッチビルドした。使用した Python バージョンは 3.12.7 である。シミュレータのプログラミング言語として Python を選定した理由としては、構造体の扱いやすさ、コーディングとデバッグの容易さ、matplotlib でグラフを出力できるといった面から選定した。次にシミュレータの挙動を時系列に説明する。

5.2.1 初期化

まず初期化としてデータセンタの設定、DGX サーバ、ラック列、部屋などのパラメータを初期化し台数分のその他消費電力を待機電力として各階層の推定消費電力に加算する。続いてトレースデータから読み取ったすべての pod の作成時刻、*creation_times* を、シミュレーションする時刻を格納する配列である *event_times* に登録する。これにより着信キューにジョブが着信し、スケジューリングの試行が発生するタイミングに必ずシミュレーションを発火することができる。以上が初期化の内訳である。

5.2.2 メインループ

メインループにおいては *event_times* ごとに処理を行う。最初にジョブの一覧をスキャンし、その時刻に作成されることになっているジョブがあればそのジョブを着信キューに登録する。次に着信キューを確認し、ジョブが着信している場合スケジュールを試み、成功した場合そのジョブが終了する予定時刻を *event_times* に登録する。続いて実行中のジョブリストを確認し、もし終了時刻となるジョブがあればそのジョブを削除、空いた箇所にジョブスケジューリングを行いたいのので *event_times* に現時刻+1 を追加する。1 ループの最後には、*event_times* が時間順に発火するようソーティングを行う。

5.2.3 グラフ描画

すべての *event_times* を完了し、すべてのタスクの処理が完了したら pyplot によりグラフを描画する。一連の動作は図 5.1 にシミュレータの動作フローチャートを示す。

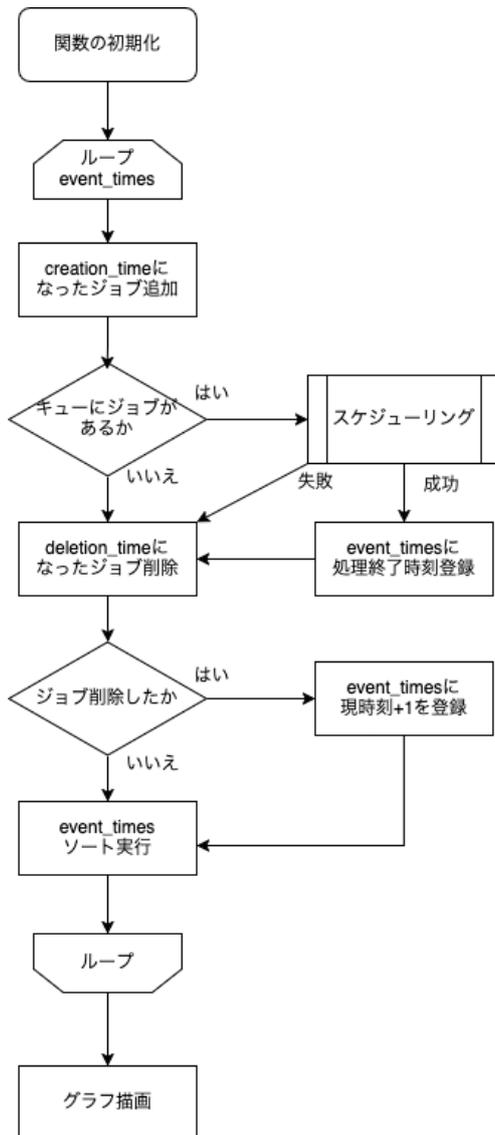


図 5.1: シミュレータフローチャート

前述の通り、本シミュレータは連続的時間軸で1秒ずつシミュレーションを実行する方式ではなく、実行時間短縮のためイベントが発生する時刻を配列化して離散的時間軸で実行する方式を採用している。具体的には上記フローチャートにおける *event_times* に格納された時刻のみを実行しており、この *event_times* には csv ファイルから読み取ったジョブ (pod) の作成時刻、スケジューリング後の処理完了時刻、あるいは Round Robin Scheduling の持ち回り切り替え時刻が登録される。ジョブの作成時間と完了時間が重複した場合など、同じ時刻が重複してシミュレーションされることがあるが、その部分は重複してシミュレーションされないようシミュレータ側で対策を行っている。

5.3 トレースデータの利用法

トレースデータの利用法を説明する。トレースデータには前述の通り、オープンノードで作成された pod の求める CPU、GPU リソース、メモリ容量、作成時間などが格納されている。リソース配分の判定に使用する想定消費電力の計算には pod の要求 CPU、GPU リソースを使用しており、具体的には CPU、GPU の要求リソース量と各計算機の最大消費電力の積から想定消費電力を算出している。またトレースデータにおいて CPU の要求リソースに 1000 ミリ、つまり 1 を超える CPU リソースを要求されるケースがあるが、それは複数スレッドの要求を示しており、スレッド分の推定消費電力を算出して推定消費電力とする。ジョブ処理にかかる時間は、シミュレーションにおいてはトレースデータの各 pod の終了時刻からキューイング時刻を引いた値とすることで擬似的に算出している。また、pod の要求メモリ容量はラウンドロビンモードによるプリエンブションシミュレーションにおける転送時間ペナルティの計算に使用している。

5.4 シミュレーション評価 1

5.4.1 電源ファシリティ制限下での性能評価

従来型のファシリティ設計では、電力ベースでのスケジューリング制御を行わないため、施設の最大供給電力によってクラスタの設置台数が制限されてしまう問題がある。例えば 9 台の DGX H100 を稼働させるためには、ピーク電力を見越して 91.8kW の電力資源を必ず整備する必要があった。しかし、本提案手法ではスケジューラが電力ベースのスケジューリング制御を行うため、クラスタの最大消費電力が施設の最大供給電力を上回る場合であっても動作を行うことが可能である。

そこで、標準供給電力から DGX クラスタ 1 台分少ない 81.6kW、3 台分少ない 61.2kW の給電能力条件において、従来型の電力制御を行わない backfill スケジューリングで 9 台のクラスタでトレースデータを処理した場合と、提案手法を利用して 9 台のクラスタで実行した場合の性能比較を行う。得られたターンアラウンドタイムの分布と推定消費電力は図 5.2 から 5.8 にかけて示し、81.6kW 条件での平均ターンアラウンドタイムと実行完了時間は表 5.1、61.2kW 条件での結果は表 5.2 にまとめて示す。平均ターンアラウンドタイムは小数点以下 2 桁で示す。

表 5.1: 81.6kW 実行完了時間の評価結果表

手法	平均 TAT(秒)	実行完了時間(秒)
電力制御なし	25788.0	12902961
提案手法	25788.0	12902961

表 5.2: 61.2kW 実行完了時間の評価結果表

手法	平均 TAT(秒)	実行完了時間(秒)
電力制御なし	失敗	失敗
提案手法	26605.0	12934909

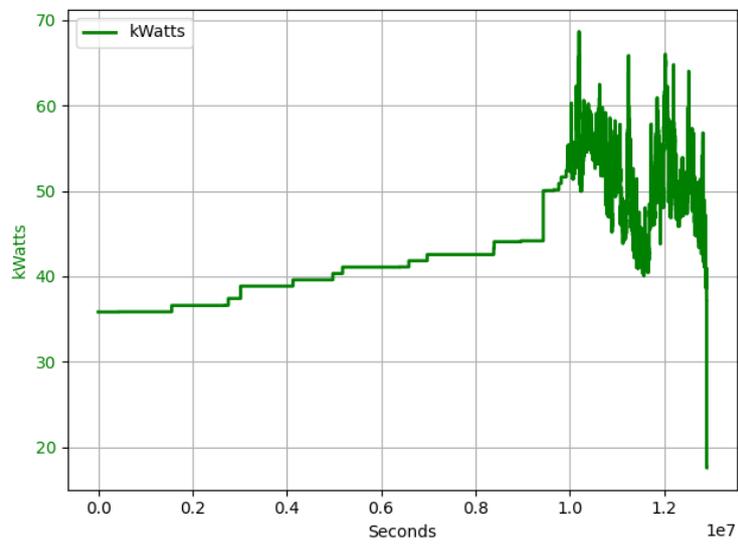


図 5.2: 提案手法 81.6kW 推定消費電力トレース図

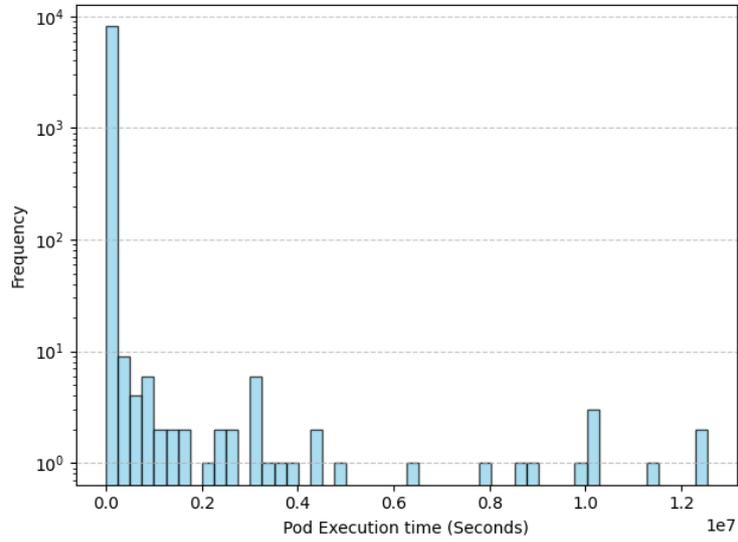


図 5.3: 提案手法 81.6kW ターンアラウンドタイム分布図

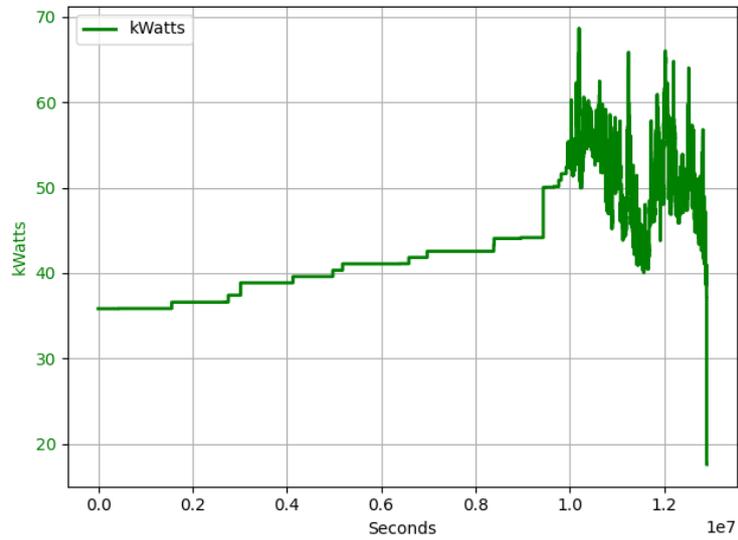


図 5.4: 電力制御なし 81.6kW 推定消費電力トレース図

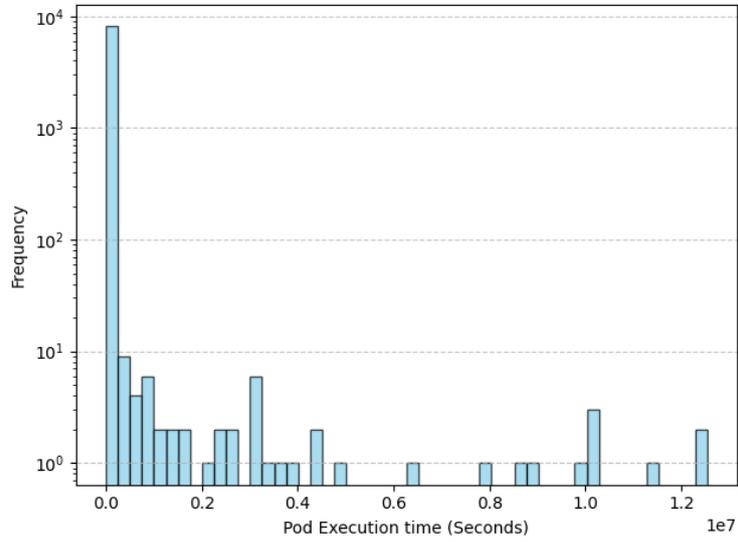


図 5.5: 電力制御なし 81.6kW ターンアラウンドタイム分布図

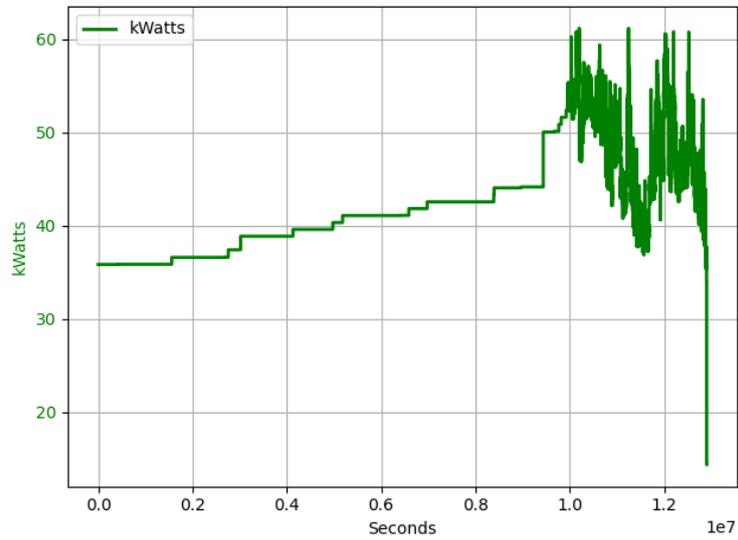


図 5.6: 提案手法 61.2kW 推定消費電力トレース図

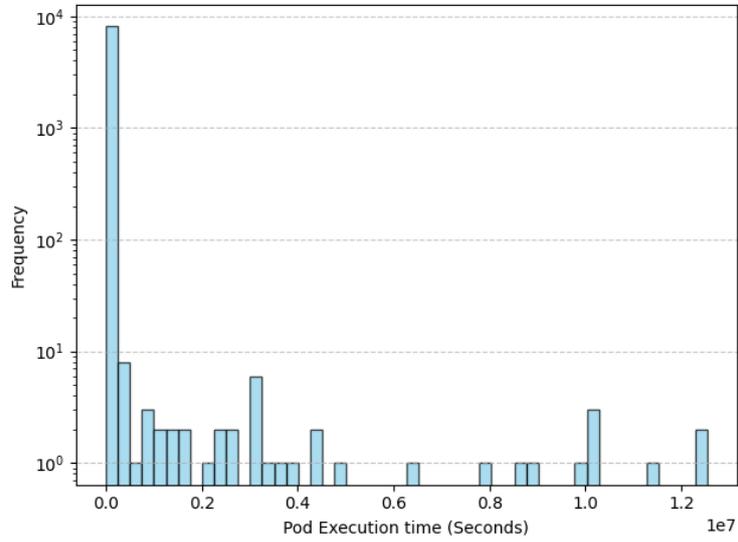


図 5.7: 提案手法 61.2kW ターンアラウンドタイム分布図

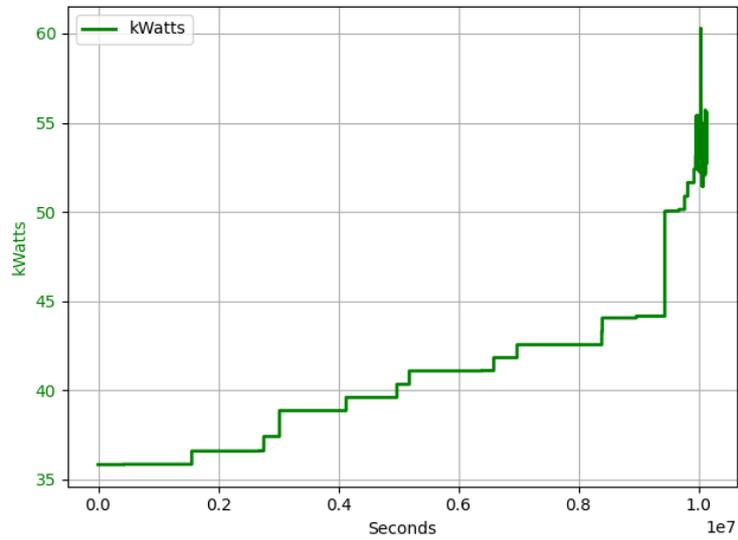


図 5.8: 電力制御なし 61.2kW 推定消費電力トレース図

シミュレーションの結果、81.6kW 条件において提案手法と従来手法、そして無制限時と比較しても平均ターンアラウンドタイム及び実行完了時間に差がないことがわかった。これは消費電力グラフからも読み取れる通り、本実験において推定消費電力が最大給電能力である 81.6kW に達しておらずペナルティの原因となるプリエンプション処理、あるいはブロッキングが発生していないため、双方においてタイムロスが発生していないことが原因であると考えられる。

61.2kW 条件においては、従来手法では電力制御を一切行っていなかったため 10134129 秒地点において、施設の電力制限を超過して実行に失敗してしまった。一方で提案手法では、電力資源に基づいたスケジューリングを行うことで消費電力が供給電力を超過する事態は無かったものの、81.6kW 各条件と比較して平均ターンアラウンドタイムが 3.0%悪化し、実行完了時間は 0.2%遅くなった。この結果から、提案手法は電力制限下において従来手法では給電能力を超過してしまうクラスタ数においては、ソフトウェアによる制御により消費電力の超過を防ぐことが可能であるが、ターンアラウンドタイムには若干の悪化が見られることがわかった。

5.4.2 クラスタ台数制限下での性能評価

続いて、1で実行に失敗した61.2kW条件において、従来手法と提案手法においてクラスタ台数を6台に揃えて実行完了時間とターンアラウンド時間のシミュレーション性能評価を行った。ターンアラウンドタイムの分布と推定消費電力を図5.9から図5.12にかけて示し、平均ターンアラウンドタイムと実行完了時間は表5.3にまとめて示す。

表 5.3: 6 台 実行完了時間の評価結果表

手法	平均 TAT(秒)	完了時間(秒)
RR なし	32195.4	13882687
提案手法	25352.8	13880826

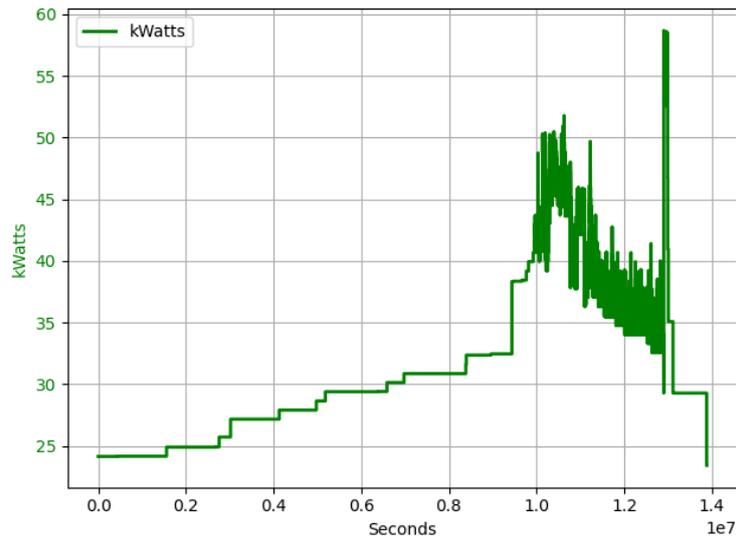


図 5.9: 提案手法 61.2kW 6 台 推定消費電力トレース図

クラスタ台数を揃えた条件において、提案手法は従来手法と比較して平均ターンアラウンドタイムが21.2%改善され、実行完了時間は0.01%改善された。この結果から、提案手法はジョブに対して小規模のクラスタにおいても従来手法と比較してターンアラウンド時間を短縮できることがわかった。これは提案手法がプリエンプションを伴うラウンドロビン制御を採用しているため、ジョブのブロッキングを防ぎ応答性を改善しているものと考えられる。一方で実行完了時間は大きく変わらない結果となったが、これはブロッキングを防いだ代わりにプリエンプションによるオーバーヘッドが発生したほか、ブロッキングによって発生した遅延が短かった可能性が考えられる。

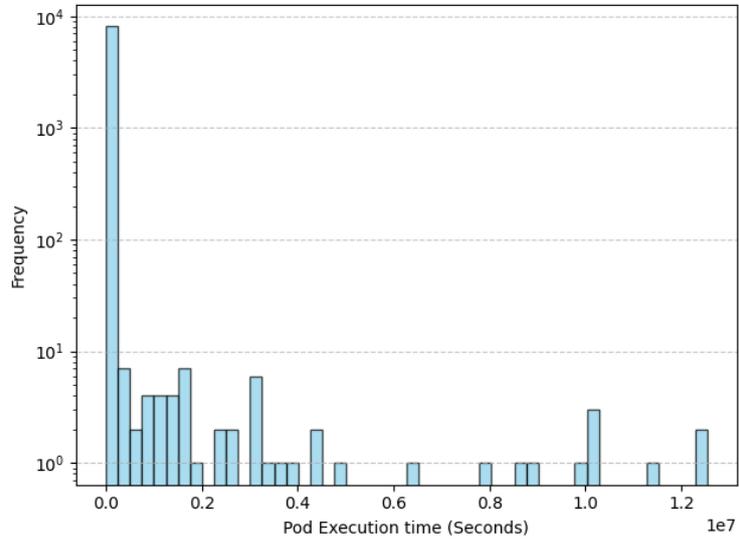


図 5.10: 提案手法 61.2kW 6 台 ターンアラウンドタイム分布図

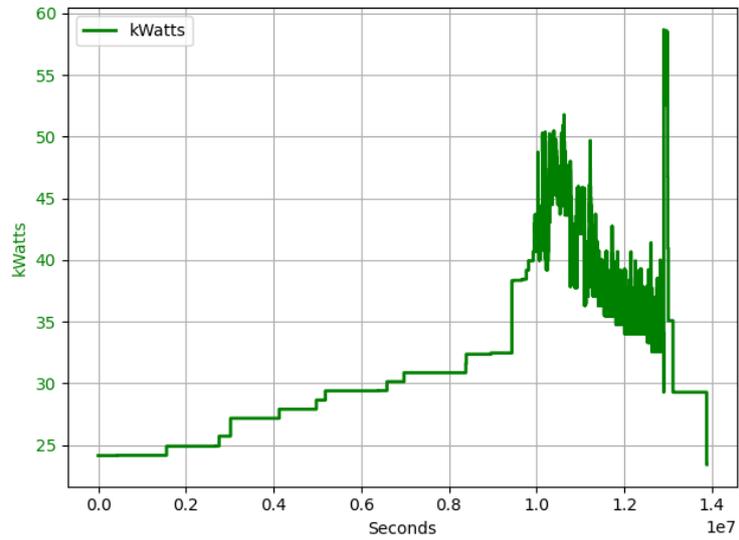


図 5.11: RR なし 61.2kW 6 台 推定消費電力トレース図

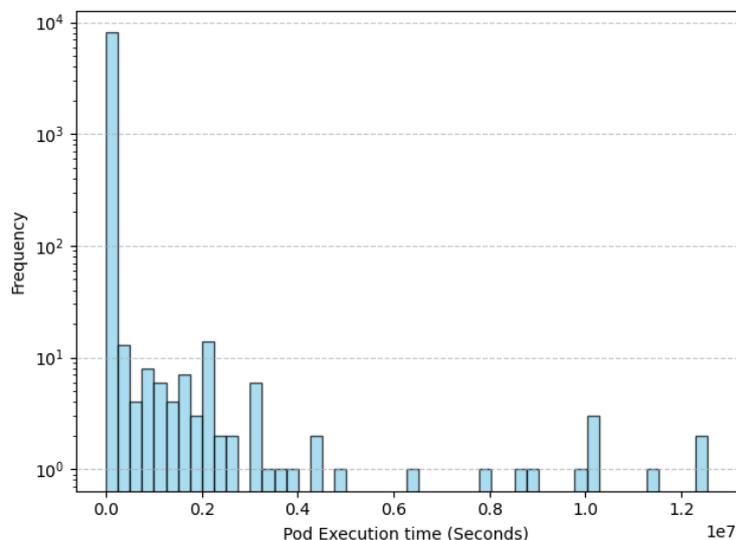


図 5.12: RR なし 61.2kW 6 台 ターンアラウンドタイム分布図

5.5 シミュレーション評価2

本提案手法では、ターンアラウンドタイム改善策として Gang Scheduling を参考にプリエンプションを伴う Round Robin 制御を採用しているが、コンテナ環境でプリエンプションを実行した場合にはチェックポイント作成処理、チェックポイントを超過した処理のロスといったタイムロスが発生する。これらの処理を行った際に、実際の環境では性能差がより大きくなる可能性がある。

しかし本シミュレータ、および使用したトレースデータからはそうしたコンテナのプリエンプション処理にどの程度負荷がかかり、またタイムロスがどの程度発生するのか推測することは不可能であるため、代替策としてプリエンプション処理を行った場合にメモリから SSD へデータ退避が発生することを想定して、転送待ちペナルティ時間を算出して加算している。具体的には、ジョブの要求メモリ容量から SSD の転送速度を利用して転送待ち時間を計算し、それを Round Robin スケジューリング時に加算するシミュレーション機能として実装している。想定環境で示したように、SSD はメーカーごとにパフォーマンスにばらつきがあるため、最も高速な製品と低速な製品でデータ転送に伴うタイムロスをシミュレーションし、比較を行った。

5.5.1 RAID

RAID とは、Redundant Array of Inexpensive Disks の略称であり、複数のハードディスクや SSD を束ねて 1 つのディスクとして扱いながら、冗長化あるいはパ

パフォーマンスの向上を図る仕組みである。

RAID0はストライピングとも呼称され、データをブロック単位に分割し同時に読み書きを行う方式である。使用可能領域はすべてのドライブの合計容量に、アクセス速度は構成台数に比例して向上するが、ドライブが1台でも破損した場合にデータの復元は不可能になる。DGXサーバのキャッシュSSDでは4台が構成されているため、読み書き速度はドライブ単体の4倍となる。

5.5.2 比較評価

比較は最もパフォーマンスの差が顕著であったSK Hynixの製品、Micron社の製品の2パターンにおいて比較を行う。比較はターンアラウンドタイム、実行完了時間で行う。供給電力条件が91.8kW,81.6kWにおいては提案手法においてプリエンプションが発生していなかったため、プリエンプションが起きていることが確認できた61.2kWとする。

表 5.4: SSD 毎 実行完了時間の評価結果表

手法	平均 TAT(秒)	実行完了時間 (秒)	総プリエンプション回数
Micron RAID0	25359.7	13880826	135596
Hynix RAID0	25354.0	13880826	130939

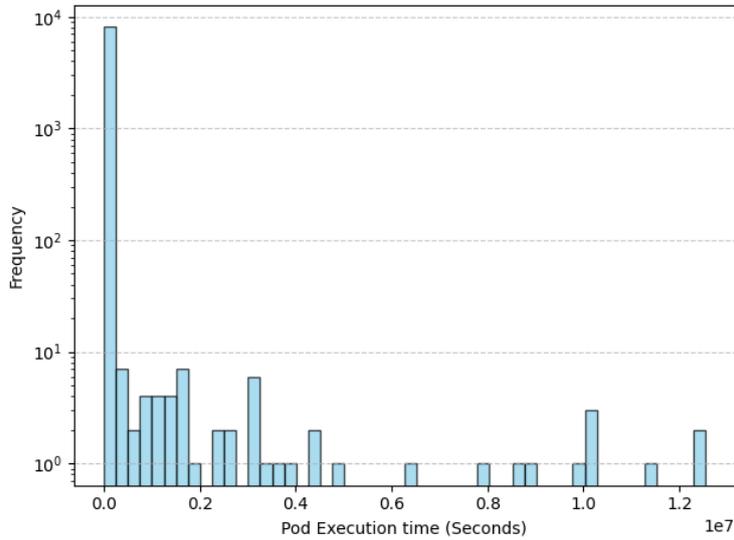


図 5.13: Micron SSD 61.2kW ターンアラウンドタイム分布図

シミュレーション結果からは比較的速度が遅く、プリエンプションによる pod データの転送待ち時間がかかる Hynix SSD のほうが完了時間、平均ターンアラウンドタイムともに早く完了する結果となった。原因を調査したところ、転送速度の速い Micron SSD では早くプリエンプション処理が完了するため、大きなジョブのプリエンプションが多く発生していることが確認できた。これにより、プリエンプションのペナルティが多く発生し、ターンアラウンドタイムが遅くなっていると考えられる。

本評価結果を受けて、プリエンプション間隔が与える実行完了時間への影響も確認を行った。

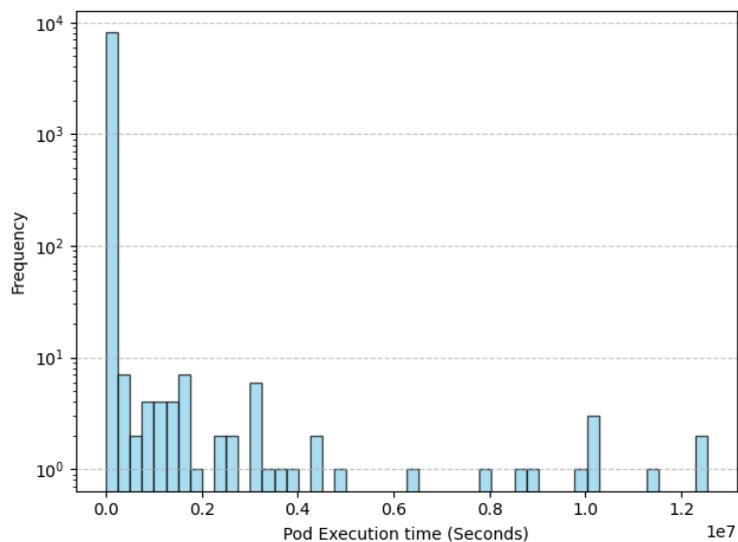


図 5.14: Hynix SSD 61.2kW ターンアラウンドタイム分布図

5.6 シミュレーション評価3

提案手法でのラウンドロビンのクォンタム値を変更すると、どのような形で実行完了時間やターンアラウンドタイムに影響を与えるかを評価する。具体的には、デフォルト値である 30 秒から半分の 15 秒、2 倍の 60 秒、4 倍の 120 秒と変更してプリエンプションが実行時間、ターンアラウンドタイムに与える影響を確認する。なお供給電力設定はプリエンプション処理が発生する 61.2kW、クラスタは 6 台、その他設定は 4.4 の標準条件で行う。

表 5.5: 61.2kW クォンタム時間毎 実行完了時間表

プリエンプション間隔 (秒)	平均 TAT(秒)	実行完了時間 (秒)	総プリエンプション回数
Quantum:15	25271.0	13880826	201122
Quantum:30	25359.7	13880826	135596
Quantum:60	24234.1	13089187	93709
Quantum:120	24532.4	13089337	66306

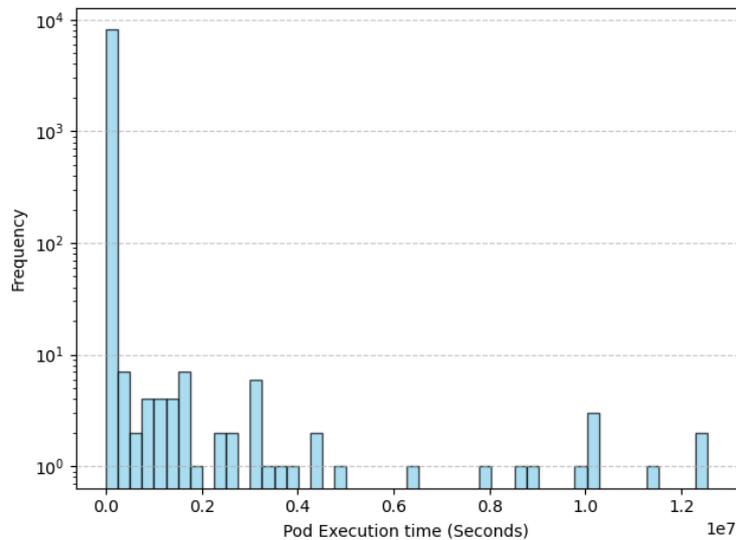


図 5.15: クォンタム 15 秒 ターンアラウンドタイム分布

シミュレーション結果から、クォンタム値が小さいほどプリエンプションが多く発生するため、転送待ちペナルティによってターンアラウンドタイムが遅くなることがわかった。逆にクォンタム値が大きくなるとプリエンプションが少なくなり、ターンアラウンドタイムと実行完了時間が短縮されることがわかった。しかし、120 秒の結果においてはクォンタムが長くなりすぎるためか、60 秒での結果と比較してターンアラウンドタイムと実行完了時間が微増する結果となっている。

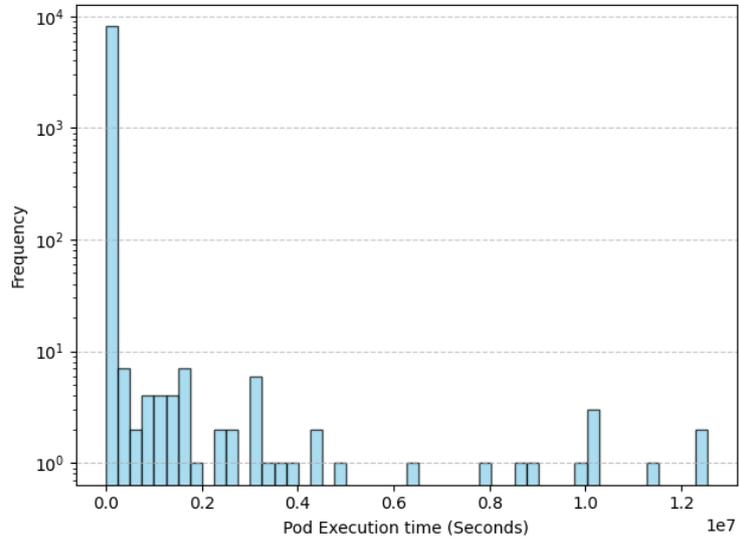


図 5.16: クォンタム 30 秒 ターンアラウンドタイム分布

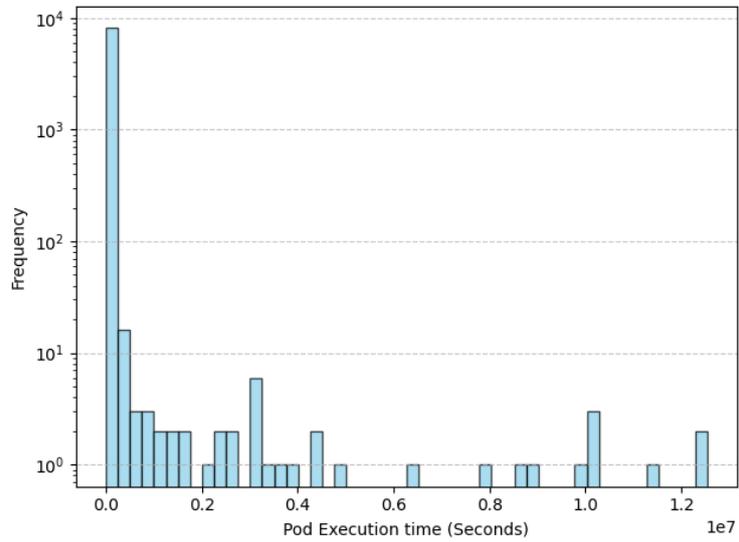


図 5.17: クォンタム 60 秒 ターンアラウンドタイム分布

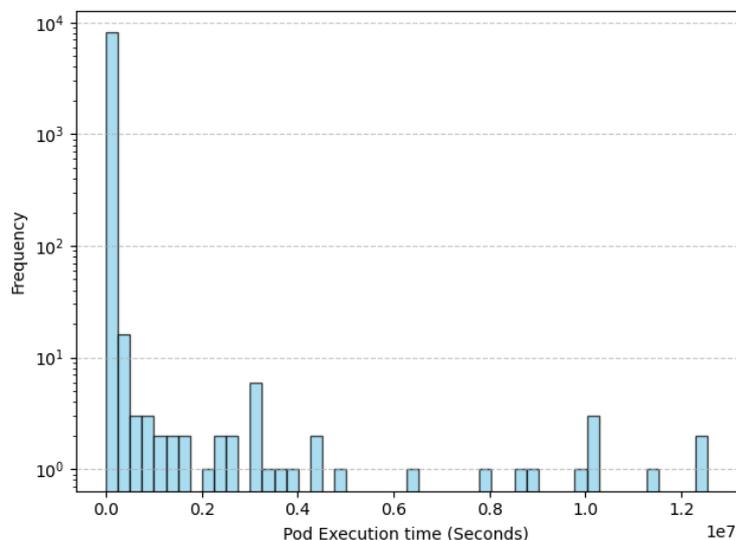


図 5.18: クォンタム 120 秒 ターンアラウンドタイム分布

5.7 評価まとめ

評価を通じて、提案手法が従来型のスケジューラと比較して、ソフトウェアによる電力制御と Round Robin 制御によって、限られた供給電力条件においてもターンアラウンドタイムを悪化を抑えながら稼働を可能とする優位性を示すことができた。

5.7.1 電力使用率の向上

まず、推定消費電力に基づく電力資源ベースのスケジューリングにより、最大消費電力に依存する制御手法より、高い電力使用率で動作させることができた。これは、ソフトウェア制御により、計算資源ではなく電力資源を中心にジョブを割り当てるスケジューリングの効果であり、特に電力制限を厳しく設定した評価 5.4.1 において、高い電力使用率と従来手法では不可能な数のクラスタの稼働を実現した。

5.7.2 ジョブのブロッキングの防止

また同時に施したジョブのブロッキング対策についても、シミュレーションにおいて優位性を示すことができた。クラスタ数を減らした 5.4.2 において、従来手法より早くタスクを実行完了可能であり、また平均ターンアラウンドタイムの成績においては従来手法比で 21.3% 高速、無制限時と比較しても 1.7% 低速に抑えるなどと、クラスタにおけるジョブブロッキングを低減する効果が確認できた。そ

の一方で、プリエンプション間隔を変更した実験において、最適な間隔ではなかった場合に平均ターンアラウンドタイムや実行完了時間が遅くなる可能性が示唆された。

5.8 考察

評価結果より、本提案手法は消費電力ベースのスケジューリングにより、目標としていた従来比での高い電力使用率と、同等のターンアラウンドタイムを達成することができた。特に電力条件と稼働可能台数の厳しい条件下において、GPU クラスタを非常に高い電力使用率で推移させ、また実行時間とターンアラウンドタイムを従来型の制御より短縮する効果を示した。これはトレースデータの事前評価でも明らかになったが、今回使用したデータでは処理時間の短いジョブがデータの多数を占めていたため、Round Robin と Backfill によってこれらジョブを、占有度の大きいジョブの片手間に効率的に処理できたことでブロッキングを防ぎながら実行時間の短縮が得られたと考えられる。

第6章 おわりに

6.1 展望

今後の展望としては、次のような点が挙げられる。

6.1.1 より正確な評価

前述の通り、本スケジューラはコンテナ環境を想定しているものであるが、シミュレーションにあたってプリエンブション時のチェックポイント作成時間ロス、GPU サーバのその他消費電力枠など、実験結果に大きく影響を与える可能性がある内容に関するいくつかの単純化を行っている。トレースデータに関する、同時に実行しなければならないコンテナの関係性などデータとして省略されている要素が存在する。今後の展望としてはそれらモデレーションを単純化した部分の精度を高めた形で追加していくことで、ブロッキングによるタイムロスとプリエンブションペナルティの精度を向上できる。

6.1.2 ジョブの粒度

本研究において使用した Alibaba トレースデータでは表面化しなかったが、本スケジューラでは粒度の細かいジョブを優先する Backfill を採用しており、こうした粒度の細かいものを優先するスケジューラは一般に大規模にノードを占有するジョブが多い場合に断片化を引き起こすとされている。GPU ノードでの断片化はネットワーク距離の悪化を意味し、実際の環境では演算性能の低下に繋がる可能性が考えられる。そのため、Alibaba のみならず、様々な傾向を持ったトレースデータで評価を行い、本提案手法のワークロードに対する得意・不得意を解明することが挙げられる。

6.1.3 適切なプリエンブション

そして最後にコンテナの最適なプリエンブション頻度に関するモデル化が考えられる。5.6の実験でも触れたが、プリエンブション間隔が短すぎる設定では過度なプリエンブションにより、転送ペナルティが大きくなり、また大きすぎる場合

はブロッキング抑制効果が低くなり、ターンアラウンドタイムや実行完了時間の悪化につながったと考えられる。さらにプリエンプシオンにあたっては実行中のコンテナデータを SSD に退避させるとしたが、SSD は消耗品であるためあまりにプリエンプシオン頻度を高めると SSD へのアクセス頻度が高まり、実際のクラスターで適用した場合にドライブを摩耗してしまう恐れがある。

6.2 まとめ

本研究は、ファシリティ資源制約、特に電力制約の大きいデータセンターにおいて GPU クラスターをソフトウェア制御によって、電力制限の制限近くまで稼働させ、厳しい電力条件においても稼働の実現とパフォーマンスの改善を狙う目的であった。本提案スケジューラは推定消費電力という概念を用い、消費電力が供給電力を超過しないよう制御を行い、かつ実行可能なジョブを最大限にスケジューリングするというその目的を十分に満たし、限られたファシリティ資源環境での GPU クラスター稼働という課題に消費電力ベースのスケジューリングという解決策を示すことができたといえる。

また、稼働させる GPU クラスターが与えられるジョブに対して小さい場合においても、Backfill と Round Robin Scheduler を併せたブロッキング低減スケジューリングにより、従来スケジューリングと比較して完了時間とターンアラウンドタイム性能の低下を抑えながら低電力での稼働を実現することができた。本研究の成果が、小型 GPU クラスターやオンプレミス LLM サーバといった需要に対するソフトウェア的解決法の一助となることが期待される。

関連図書

- [1] Wencong Xiao, Beihang University & Microsoft Research; Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, and Nipun Kwatra, Microsoft Research; Zhenhua Han, The University of Hong Kong & Microsoft Research; Pratyush Patel, Microsoft Research; Xuan Peng, Huazhong University of Science and Technology & Microsoft Research; Hanyu Zhao, Peking University & Microsoft Research; Quanlu Zhang, Fan Yang, and Lidong Zhou, Microsoft Research:”Gandiva: Introspective Cluster Scheduling for Deep Learning”.usenix.2018.<https://www.usenix.org/conference/osdi18/call-for-papers>,(2025年1月訪問)
- [2] Juncheng Gu, Mosharaf Chowdhury, and Kang G. Shin, University of Michigan, Ann Arbor; Yibo Zhu, Microsoft and Bytedance; Myeongjae Jeon, Microsoft and UNIST; Junjie Qian, Microsoft; Hongqiang Liu, Alibaba; Chuanxiong Guo, Bytedance.”Tiresias: A GPU Cluster Manager for Distributed Deep Learning”.usenix.2019. <https://github.com/alibaba/clusterdata>,(2025年1月訪問)
- [3] 西本 伊織, 山内 利宏, 小林 諭, 加藤 純, 三輪 真弘, 谷口 秀夫:”HPCにおけるGang Scheduling の切替間隔動的变化手法”, 研究報告システムソフトウェアとオペレーティング・システム (OS) Vol15,2024-OS-162 pp.1-7
- [4] 平井浩一 井口裕次 宇野篤也 黒川原佳:”スーパーコンピュータ「京」の運用管理ソフトウェア”. 富士通株式会社,<https://www.fujitsu.com/downloads/JP/archive/imgjp/jmag/vol163-3/paper09.pdf>,(2025年1月訪問).
- [5] 宇野 篤也 末安 史親 関澤 龍一:”スーパーコンピュータ「富岳」の運用系ソフトウェア”. 富士通株式会社,<https://www.fujitsu.com/jp/documents/about/resources/publications/technicalreview/2020-03/article10.pdf>(2025年1月訪問).
- [6] 株式会社インターネットイニシアティブ 広報部:”IJ, 最新省エネ技術・AIを採用した「白井データセンターキャンパス」が完成”. 株式会社インターネットイ

- ニシアティブ.2019.<https://www.iiij.ad.jp/news/pressrelease/2019/0410.html>,(2025年1月訪問).
- [7] Dell. "PowerEdge XE9680 高密度 GPU サーバー".Dell Inc.2025 <https://www.dell.com/ja-jp/shop/ipovw/poweredge-xe9680> (2025年1月訪問)
- [8] Supermicro. "GPU SuperServer SYS-821GE-TNHR".Micro Computer, Inc.2025 {<https://www.supermicro.com/ja/products/system/GPU/8U/SYS-821GE-TNHR>} (2025年1月訪問)
- [9] Hewlett Packard Enterprise Development LP. "HPE Cray XD670". Hewlett Packard Japan,G.K .2025 <https://www.hpe.com/jp/ja/product-catalog/detail/pip.hpe-cray-xd670.1014737116.html> (2025年1月訪問)
- [10] NVIDIA."NVIDIA DGX H100/H200 User Guide".NVIDIA Corporation.2024.<https://docs.nvidia.com/dgx/dgxh100-user-guide/introduction-to-dgxh100.html>,(2025年1月訪問).
- [11] NVIDIA."NVIDIA H100 Tensor コア GPU 概要".NVIDIA Corporation.2023.<https://resources.nvidia.com/ja-jp-gpu>,(2025年1月訪問).
- [12] Intel."インテル® Xeon® Platinum 8480C プロセッサ".Intel Corporation.2023. <https://www.intel.co.jp/content/www/jp/ja/products/sku/231730/intel-xeon-platinum-8480c-processor-105m-cache-2-00-ghz/specifications.html>,(2025年1月訪問).
- [13] KIOXIA."KIOXIA CM6-R シリーズ".KIOXIA Corporation.2024. <https://www.kioxia.com/content/dam/kioxia/ja-jp/business/ssd/enterprise-ssd/asset/productbrief/eSSD-CM6-R-product-brief-J.pdf>,(2025年1月訪問)
- [14] Micron."Micron 7450 NVMe™ SSD".Micron Technology,Inc.2024. <https://jp.micron.com/content/dam/micron/global/public/documents/products/technical-marketing-brief/7450-nvme-ssd-tech-prod-spec.pdf>,(2025年1月訪問)
- [15] Samsung semiconductor."Samsung MZQL23T8HCLS-00A07".Samsung.2024. <https://semiconductor.samsung.com/jp/ssd/>

datacenter-ssd/pm9a3/mzql23t8hcls-00a07/, (2025 年 1 月
訪問)

[16] SK Hynix.”SK Hynix PE8000 series”.SK hynix INC. <https://product.skhynix.com/products/ssd/essd/pe8000.go>, (2025 年 1 月訪問)

[17] GDEP.”速報! DGX H100 イントロダクションガイド”.GDEP ADVANCE, Inc. 2023. https://www.gdep.co.jp/information/report/dgXH100_introduction_guide/, (2025 年 1 月訪問)

[18] qzweng.”Alibaba Cluster Trace Program”.Alibaba. 2023. <https://github.com/alibaba/clusterdata>, (2025 年 1 月訪問).

謝辞

今回の研究を行うにあたり,多くの方々のご助言なくして本研究の執筆は不可能でした。心より深く御礼申し上げます。

研究の進行および発表にあたり,主指導教員の北陸先端科学技術大学院大学 篠田 陽一教授,副指導の宇多 仁 准教授には数多くのご指導,ご鞭撻,ご助言をいただき,それらなくして本論文の執筆は困難であったことを深く感謝申し上げます。またインターシップ指導教員の丹 康雄 教授,田中 清史 教授には中間発表や修士論文審査会において,研究の方向性や進行,評価手法に関して貴重なご助言を頂戴いたしましたことを深く感謝いたします。最後に,本研究室メンバーの中川 颯馬 氏,金田 昂大 氏,高田 敦生 氏,吉川 健太 氏,Yao Yudie 氏,中村 一貴 氏,西村 優典 氏,Lu Xingchen 氏,三浦 良介 氏,家族の助力なくして本論文は執筆できなかったことに感謝の意を表し,謝辞とさせていただきます。

付録

本研究で作成した、提案手法のシミュレータのソースコードを次に示す。

Listing 6.1: simulator9.py

```
1
2     # Ryoma Resorce scheduler simulator with Round
      Robin
3
4     # IMPORTANT
5     # 文中にとある部分は間違いです。GANGRound と読み替えてく
      ださい Robin
6     # The part in the sentence that says "GANG
      scheduling" or "GANG" is misuse. Please read it
      as Round Robin Scheduling.
7
8     import pandas as pd
9     import matplotlib.pyplot as plt
10    from queue import Queue
11
12    csv_file_path = "openb_pod_list_default.csv"
13    DC_POW = 61.2
14    num_Of_DGXs = 9
15    variable_RR = True
16    RR_INTERVAL = 30
17
18    # 定数の設定
19    #DC_POW = 20.4 # 単位で設定 kW
20    DGX_PER_RACK = 1
21    ROOM_POW_LIMIT = DC_POW 一部屋あたりの最大消費電力
      # (kw)
22    NUM_OF_ROOMs = 1 サーバルーム数#
23    RACKROW_POW_LIMIT = DC_POW ラック列あたりの最大消費電
      力# (kw)
24    NUM_OF_Rackrow = num_Of_DGXs ラック列の総数#
25    DGX_MAX_TDP = 10.2 #DGX の最大消費電力は
      H100Max 10.2kw
26    DGX_SSD_READBW = 6800*4 #のボトルネック計算用 の帯域
      RRDGXSSD単位は, MB/s
27    DGX_SSD_WRIBW = 5300*4 #のボトルネック計算用 の帯域
      RRDGXSSD単位は, MB/s
```

```

28     H100_POW = 0.7 # の消費電力は台 H1001700W
29     XEON_POW = 0.35 #Xeon 8480の消費電力は台 C1350W
30     XEON_TH_POW = XEON_POW/112 あたりの消費電力#1
31     DGX_BASIC_POW = DGX_MAX_TDP - XEON_POW*2 -
        H100_POW*8
32
33     # 計算機だけなら 6.3kw
34     # 残りは 3.9kw
35     # DGX_num = 8
36     #
37
38     # RR の有効、無効
39     RR = False
40     #variable_RR = True
41     #RR_INTERVAL = 30
42
43     glo_TAT = 0
44     completed_counter = 0
45
46     # 追加: clock と active_Jobs の数を記録するリスト
47     clock_list = []
48     active_gpus_count = []
49     active_jobs_count = []
50     secondqueue_count = []
51     watt_count = []
52     current_Watt = 0
53     TAT = {
54     }
55     RR_activations = 0
56
57
58     def init_Data():
59         global csv_file_path
60         # ファイルのパス CSV
61         #csv_file_path = "testcsv_82.csv"
62         # データを読み込む
63         data = pd.read_csv(csv_file_path)
64
65         # 空欄の scheduled_time を creation_time で埋める
66         data['effective_scheduled_time'] = data['
            scheduled_time'].fillna(data['creation_time
            '])
67         # execution_duration の計算
68         data['execution_duration'] = data['
            deletion_time'] - data['
            effective_scheduled_time']

```

```

69         return data
70
71     # 数を初期化する関数 GPU
72     def init_GPU_Array():
73         return num_Of_DGXs
74
75     def init_DGX_Resources(num_dgx):
76         """リソースの初期化。
77         DGX各には
78         DGX8 と GPU224 が割り当てられる。CPU
79         """
80         global current_Watt
81         dgx_resources = []
82         for i in range(num_dgx):
83             rackrow_id = i // DGX_PER_RACK # ラック列を
            循環
84             room_id = int(i // (num_Of_DGXs /
            NUM_OF_ROOMS)) # サーバルームを
            計算
85             dgx_resources.append({
86                 'id': i,
87                 'available_gpus': 8,
88                 'available_cpus': 224,
89                 'rackrow': rackrow_id,
90                 'room': room_id,
91                 'power_usage': DGX_BASIC_POW # 各に初期待
            機電力 DGX3.9を設定 kW
92             })
93             current_Watt += DGX_BASIC_POW
94         #breakpoint()
95         return dgx_resources
96
97     def allocate_resources(dgx_resources, task):
98         """タスクをスケジュールするためにリソースを割り当てる。
99         DGX
100        """
101         global RR
102         global current_Watt
103         for dgx in dgx_resources:
104             # タスクの追加消費電力を計算
105             additional_power = ((task['cpu_milli'] +
            1000 - 1) // 1000 * XEON_TH_POW) + task
            ['num_gpu'] * task['gpu_milli'] // 1000
            * H100_POW
106             #print(f"Current + additional power: {
            additional_power+current_Watt}")
107

```

```

108 # データセンター全体の消費電力制限
109 #total_power_usage = sum(room_usage) # 現在の
      全サーバールームの消費電力
110 if current_Watt + additional_power > DC_POW:
111     if variable_RR: 可変#RR 割当不可になると発動
      RR
112         RR = True
113         print("RR activated")
114     else:
115         print("DC POWER IS OVER!!")
116         exit(-1)
117     return False # データセンター全体の消費電力制
      限を超えるため、割り当てられず
118 # サーバルームの消費電力制限を超えないか
119 if room_usage[dgx['room']] +
      additional_power > ROOM_POW_LIMIT:
120     #print(f"Room {dgx['room']} is full. {
      room_usage[dgx['room']] +
      additional_power}")
121     continue # サーバルームの消費電力制限を超え
      る場合、これをスキップ DGX
122 # ラック列の消費電力制限を超えないか
123 if rackrow_usage[dgx['rackrow']] +
      additional_power > RACKROW_POW_LIMIT:
124     #print(f"Rackrow {dgx['rackrow']} is
      full. {rackrow_usage[dgx['rackrow']]
      + additional_power}")
125     #print(f"rackrow_usage: {rackrow_usage},
      at {dgx['rackrow']}")
126     continue # ラック列の消費電力制限を超える場合、
      これをスキップ DGX
127 # 各の最大消費電力 () を超えないか DGXDGX_MAX_TDP
128 if dgx['power_usage'] + additional_power >
      DGX_MAX_TDP:
129     #print(f"DGX {dgx['id']} is full. {dgx['
      power_usage'] + additional_power}")
130     continue #の消費電力制限を超える場合、これをス
      キップ DGXDGX
131 # とのリソースが十分かどうかを確認 GPU CPU
132 if dgx['available_gpus'] >= int(task['
      num_gpu']) and dgx['available_cpus'] >=
      int((task['cpu_milli'] + 1000 - 1) //
      1000):
133     # リソースの割り当て
134     dgx['available_gpus'] -= int(task['
      num_gpu'])
135     dgx['available_cpus'] -= int((task['
      cpu_milli'] + 1000 - 1) // 1000)

```

```

136         dgx['power_usage'] += additional_power
137         rackrow_usage[dgx['rackrow']] +=
            additional_power # ラック列の消費電力を
            更新
138         room_usage[dgx['room']] +=
            additional_power # サーバルームの消費電
            力を更新
139         current_Watt += additional_power
140         # タスクに割り当てられたを記録 DGX
141         task['assigned_dgx'] = dgx['id']
142         print(f"Resources allocated: DGX {dgx['id']
            '}}, GPUs: {dgx['available_gpus']},
            CPUs: {dgx['available_cpus']}")
143         print(f"PowerUsage is now {current_Watt
            }")

144
145         if current_Watt > DC_POW:
146             breakpoint()
147             return True
148         #print(f"Insufficient resources for Task {task
            ['name']}.")
149         if variable_RR: 可変#RR 割当不可になると発動 RR
150             RR = True
151             #breakpoint()
152             return False # 十分なりソースがない場合帰る
153
154     def release_resources(dgx_resources, task):
155         """タスク完了時に使用したリソースを解放する。
156
157         """
158         global current_Watt
159         for dgx in dgx_resources:
160             if dgx['id'] == task['assigned_dgx']:
161                 released_power = (task['cpu_milli'] +
                    1000 - 1) // 1000 * XEON_TH_POW +
                    task['num_gpu'] * task['gpu_milli']
                    // 1000 * H100_POW
162                 dgx['available_gpus'] += int(task['
                    num_gpu'])
163                 dgx['available_cpus'] += int((task['
                    cpu_milli'] + 1000 - 1) // 1000)
164                 dgx['power_usage'] -= released_power
165                 rackrow_usage[dgx['rackrow']] -=
                    released_power # ラック列の消費電力を
                    戻す
166                 room_usage[dgx['room']] -=
                    released_power # サーバルームの消費電力

```

```

167         を戻す
168         current_Watt -= released_power
169         return
170     def execute_Task(clock, queue, event_times,
171                     dgx_resources):
172         """タスクを検査して
173         creation 時刻になったか判定し、適切な処理を行う。
174         """
175         temp_list = []
176         for i in range(queue.qsize()):
177             task = queue.get() # キューからタスクを取得
178
179             if any(job['name'] == task['name'] for job
180                   in active_Jobs): 重複を検出して削除
181                 #
182                 print(f"Duplicate task '{task['name']}'
183                       found in active_Jobs. Discarding the
184                       task.")
185                 continue
186
187             if clock < task['creation_time']:
188                 temp_list.append(task) # まだcreation 時刻
189                 に満たないものを保持
190             else:
191                 # creation 時刻を迎えたタスクをscheduler に
192                 投げ込む
193                 #print("Throw:", task['name'])
194                 if not scheduler(task, clock, event_times
195                                 , dgx_resources):
196                     temp_list.append(task) # スケジューリン
197                     グに失敗したらtemp_list に戻す
198
199             # 条件を満たすタスクを再度キューに戻す
200             for task in temp_list:
201                 queue.put(task)
202         return queue
203
204     def execute_Penalty(memory_mib, bandwidth):
205         return int(memory_mib // bandwidth)
206
207     def scheduler(task, clock, event_times,
208                 dgx_resources):
209         """タスクをスケジューリングする関数。スケジューリングが成
210         功した場合は、失敗した場合は
211
212         True False を返す。

```

```

202         """
203         global RR
204         global RR_activations
205
206         if task['execution_duration'] < 1:
207             return False
208         if allocate_resources(dgx_resources, task): #
                リソースの割り
                当て
209             #TAT['Task'].append(task['name'])
210             if RR and task['execution_duration'] >
                RR_INTERVAL:
211                 gang_sum =RR_INTERVAL + execute_Penalty(
                task['memory_mib'],DGX_SSD_READBW
                *0.954) + execute_Penalty(task['
                memory_mib'],DGX_SSD_WRIBW*0.954)
212                 #breakpoint()
213                 RR_activations += 1
214                 task['deletion_time'] = int(clock +
                gang_sum)
215                 task['gang'] = True
216             else:
217                 task['deletion_time'] = int(clock + task
                ['execution_duration'])
218                 task['gang'] = False
219
220                 active_Jobs.append(task)
221                 event_times.append(task['deletion_time']) #
                集合に追
                加
222                 print(f"Task {task['name']} scheduled on DGX
                {task['assigned_dgx']}.")
223                 return True
224             else:
225                 #print(f"Task {task['name']} cannot be
                scheduled due to insufficient resources
                .")
226                 return False # スケジューリングに失敗
227
228     def process_completed_tasks(clock, active_jobs,
                second_Queue, dgx_resources):
229         """完了したタスクを処理し、必要に応じてリソースを解放
                する。
230
231         """
232         global RR
233         global completed_counter

```

```

234     global glo_TAT
235
236     completed_tasks = [task for task in active_jobs
237         if task['deletion_time'] == clock]
238
239     for task in completed_tasks:
240
241         if RR and task['gang']:
242             task['execution_duration'] -= RR_INTERVAL
243             second_Queue.put(task) # 再スケジューリング
244             用にキューに戻す
245             print(f"Task {task['name']} rescheduled
246                 with {task['execution_duration']} ms
247                 remaining.")
248         else:
249             print(f"Task {task['name']} completed.")
250             TAT[task['name']] = []
251             TAT[task['name']].append(clock - task['
252                 creation_time'])
253             glo_TAT += clock - task['creation_time']
254             completed_counter += 1
255             release_resources(dgx_resources, task) # リ
256             ソースを解放
257             active_jobs.remove(task)
258             if variable_RR 可変:#RR
259                 RR = False
260     return len(completed_tasks) # 完了したタスクの数を
261     返す
262
263 def process_new_tasks(clock, data, queue):
264     """新しいタスクをキューに追加する処理
265
266     """
267     global oldclock
268     if oldclock == clock:
269         return
270
271     oldclock = clock
272     new_tasks = data[data['creation_time'] == clock
273 ]
274     for _, task in new_tasks.iterrows():
275         queue.put(task.to_dict())
276         #print(f"Task added to queue: {task.to_dict
277             ()}")
278
279 # シミュレーションの初期設定
280 clock = 0
281 second_Queue = Queue() # 着信キュー

```

```

272     first_Queue = Queue() # コスト超過時の優先待機キュー
273     active_Jobs = [] # 実際に稼働している配列 GPU
274     oldclock = -1
275
276     if __name__ == "__main__":
277         # の台数を初期化 DGX
278         num_Of_DGXs = init_GPU_Array()
279         print("DGX cluster is", num_Of_DGXs)
280         print("DC Power is", DC_POW, "kW")
281         # のリソース状態を初期化 DGX
282         dgx_resources = init_DGX_Resources(num_Of_DGXs)
283         print("Initialized DGX resources:",
284               dgx_resources)
285
286         data = init_Data()
287         rackrow_usage = [DGX_PER_RACK * DGX_BASIC_POW]
288             * NUM_OF_Rackrow # 各ラック列の消費電力を初
289             期化
290         room_usage = [DGX_BASIC_POW * (num_Of_DGXs //
291             NUM_OF_ROOMs)] * NUM_OF_ROOMs # 各サーバールー
292             ムの消費電力を初期化
293
294         # 全イベント時間（開始時間と終了時間）をリスト化し、
295         ソート
296         event_times = sorted(
297             list(data['creation_time']))
298
299         # 時計の進行に応じてシミュレーションを実行
300         for next_clock in event_times:
301             clock = next_clock
302
303             # 新しいタスクの処理
304             process_new_tasks(clock, data, second_Queue)
305             if not second_Queue.empty():
306                 second_Queue = execute_Task(clock,
307                     second_Queue, event_times, dgx_resources
308                 )
309
310             # 完了したタスクの処理は処理した数return
311             deleted_Tasks = process_completed_tasks(
312                 clock, active_Jobs, second_Queue,
313                 dgx_resources)
314             if deleted_Tasks != 0:
315                 if not second_Queue.empty():
316                     print("TRYING BACKFILL")
317                     event_times.append(clock+1)

```

```

309
310         clock_list.append(clock)
311         secondqueue_count.append(second_Queue.qsize
           ())
312
313         total_active_gpus = sum(int(task['num_gpu'])
           for task in active_Jobs)
314         total_active_cpus = sum(int((task['cpu_milli
           ']' + 1000 - 1) // 1000) for task in
           active_Jobs)
315
316         print(f"Clock: {clock}, Total active GPUs in
           use: {total_active_gpus} / {num_Of_DGXs
           * 8}, Watt: {current_Watt}")
317
318         active_gpus_count.append(total_active_gpus)
319         active_jobs_count.append(len(active_Jobs))
320         watt_count.append(current_Watt)
321         event_times.sort()
322
323         print(f"Clock: {clock}")
324         #breakpoint()
325         print(f"All jobs are completed at: {clock}")
326         print(f"Total Average TAT is: {glo_TAT/
           completed_counter}")
327         print(f"RR allowed: {RR_activations}")
328
329 plt.figure(figsize=(10, 6))
330
331 # 双対軸を作成
332 fig, ax1 = plt.subplots()
333 """
334 # つ目の軸1: Active GPUs
335 color = 'blue'
336 ax1.set_xlabel('Clock')
337 ax1.set_ylabel('Number of Active GPUs', color=
           color)
338 ax1.plot(clock_list, active_gpus_count, label='
           Number of Active GPUs', color=color, marker='o
           ')
339 ax1.tick_params(axis='y', labelcolor=color)
340 ax1.legend(loc='upper left')
341
342 # つ目の軸2: Second Queue Jobs
343 ax2 = ax1.twinx() # Twin the x-axis
344 color = 'red'

```

```

345     ax2.set_ylabel('Number of Queue Jobs', color=color
346     )
347     ax2.plot(clock_list, secondqueue_count, label='
348     Number of Queue Jobs', color=color, marker='x')
349     ax2.tick_params(axis='y', labelcolor=color)
350     ax2.legend(loc='upper right')
351
352     color = 'green'
353     ax1.set_ylabel('Active Jobs', color=color)
354     ax1.plot(clock_list, active_jobs_count, label='
355     Number of Active Jobs', color=color, marker='o
356     ')
357     ax1.tick_params(axis='y', labelcolor=color)
358     ax1.legend(loc='upper left')
359
360     color = 'green'
361     plt.xlabel('Seconds')
362     ax1.set_ylabel('kWatts', color=color)
363
364     # 棒グラフに変更
365
366     ax1.step(clock_list, watt_count, where='post',
367     label='kWatts', color=color, linewidth=2)
368
369     ax1.tick_params(axis='y', labelcolor=color)
370     ax1.legend(loc='upper left')
371     #ax1.set_ylim(0, DC_POW+5)
372     # タイトルとグリッド
373     fig.tight_layout()
374     plt.grid(True)
375
376     # グラフを表示
377     plt.show()
378
379     """
380
381     # キー（横軸）と値（縦軸）を取得
382     tasks = list(TAT.keys())
383     values = [v[0] for v in TAT.values()] # 各リストの
384     最初の値を取得
385
386     # ヒストグラムの作成
387     plt.hist(values, bins=50, color='skyblue',
388     edgecolor='black', alpha=0.7)
389     # 縦軸を対数スケールに設定

```

```

384     plt.yscale('log')
385
386     # ラベルとタイトル
387     plt.xlabel('Pod Execution time (Seconds)')
388     plt.ylabel('Frequency')
389
390     plt.grid(axis='y', linestyle='--', alpha=0.7)
391     plt.tight_layout()
392     plt.show()
393     """
394     """

```

続けて、従来手法 **backfill** 用のシミュレータのソースコードを次に示す。

Listing 6.2: simulator9b.py

```

1     # no resorce controll backfill version
2
3     import pandas as pd
4     import matplotlib.pyplot as plt
5     from queue import Queue
6
7
8     csv_file_path = "openb_pod_list_default.csv"
9     DC_POW = 81.6
10    num_Of_DGXs = 9
11    variable_RR = False
12    RR_INTERVAL = 30
13
14    # 定数の設定
15    #DC_POW = 20.4 # 単位で設定 kW
16    DGX_PER_RACK = 1
17    ROOM_POW_LIMIT = DC_POW 一部屋あたりの最大消費電力
18    # (kw)
19    NUM_OF_ROOMs = 1 サーバルーム数#
20    RACKROW_POW_LIMIT = DC_POW ラック列あたりの最大消費電
21    力# (kw)
22    NUM_OF_Rackrow = num_Of_DGXs ラック列の総数#
23    DGX_MAX_TDP = 10.2 #DGX の最大消費電力は
24    H100Max 10.2kw
25    DGX_SSD_READBW = 6532.67*4 #のボトルネック計算用 の
26    帯域 GANGDGXSSD単位は, MiB/s
27    DGX_SSD_WRIBW = 3957.75*4 #のボトルネック計算用 の帯
28    域 GANGDGXSSD単位は, MiB/s
29    H100_POW = 0.7 # の消費電力は台 H1001700W
30    XEON_POW = 0.35 #Xeon 8480の消費電力は台 C1350W
31    XEON_TH_POW = XEON_POW/112 あたりの消費電力#1
32    DGX_BASIC_POW = DGX_MAX_TDP - XEON_POW*2 -
33    H100_POW*8

```

```

28     # 計算機だけなら 6.3kw
29     # 残りは 3.9kw
30     # DGX_num = 8
31     #
32
33     # GANG の有効、無効
34     RR = False
35     #variable_RR = True
36     #RR_INTERVAL = 30
37
38     glo_TAT = 0
39     completed_counter = 0
40
41     # 追加: clock と active_Jobs の数を記録するリスト
42     clock_list = []
43     active_gpus_count = []
44     active_jobs_count = []
45     secondqueue_count = []
46     watt_count = []
47     current_Watt = 0
48     TAT = {
49     }
50     gang_activations = 0
51
52
53     def init_Data():
54         global csv_file_path
55         # ファイルのパス CSV
56         #csv_file_path = "testcsv_82.csv"
57         # データを読み込む
58         data = pd.read_csv(csv_file_path)
59
60         # 空欄の scheduled_time を creation_time で埋める
61         data['effective_scheduled_time'] = data['
62             scheduled_time'].fillna(data['creation_time
63             '])
64
65         # execution_duration の計算
66         data['execution_duration'] = data['
67             deletion_time'] - data['
68             effective_scheduled_time']
69         return data

```

```

70     def init_DGX_Resources(num_dgx):
71         """リソースの初期化。
72         DGX各には
73         DGX8 と GPU224 が割り当てられる。CPU
74         """
75         global current_Watt
76         dgx_resources = []
77         for i in range(num_dgx):
78             rackrow_id = i // DGX_PER_RACK # ラック列を
              循環
79             room_id = int(i // (num_of_DGXs /
              NUM_OF_ROOMS)) # サーバルームを
              計算
80             dgx_resources.append({
81                 'id': i,
82                 'available_gpus': 8,
83                 'available_cpus': 224,
84                 'rackrow': rackrow_id,
85                 'room': room_id,
86                 'power_usage': DGX_BASIC_POW # 各に初期待
              機電力 DGX3.9を設定 kW
87             })
88             current_Watt += DGX_BASIC_POW
89         #breakpoint()
90         return dgx_resources
91
92     def allocate_resources(dgx_resources, task):
93         """タスクをスケジュールするためにリソースを割り当てる。
94         DGX
95         """
96         global RR
97         global current_Watt
98         for dgx in dgx_resources:
99             # タスクの追加消費電力を計算
100             additional_power = ((task['cpu_milli'] +
              1000 - 1) // 1000 * XEON_TH_POW )+ task
              ['num_gpu'] * task['gpu_milli'] // 1000
              * H100_POW
101             #print(f"Current + additional power: {
              additional_power+current_Watt}")
102
103
104             if dgx['available_gpus'] >= int(task['
              num_gpu']) and dgx['available_cpus'] >=
              int((task['cpu_milli'] + 1000 - 1) //
              1000):
105                 # データセンター全体の消費電力制限

```

```

106         #total_power_usage = sum(room_usage) # 現在の
           全サーバールームの消費電力
107         if current_Watt + additional_power >
           DC_POW:
108             print(f"", current_Watt +
                   additional_power, "kW attempted. DC
                   POWER IS OVER!!")
109             breakpoint()
110
111             plt.figure(figsize=(10, 6))
112             # 双対軸を作成
113             fig, ax1 = plt.subplots()
114             color = 'green'
115             plt.xlabel('Seconds')
116             ax1.set_ylabel('kWatts', color=color)
117             # 棒グラフに変更
118             ax1.step(clock_list, watt_count, where
                      ='post', label='kWatts', color=
                      color, linewidth=2)
119             ax1.tick_params(axis='y', labelcolor=
                             color)
120             ax1.legend(loc='upper left')
121             #ax1.set_ylim(0, DC_POW+5)
122             # タイトルとグリッド
123             fig.tight_layout()
124             plt.grid(True)
125             # グラフを表示
126             plt.show()
127
128             exit(-1)
129         # リソースの割り当て
130         dgx['available_gpus'] -= int(task['
           num_gpu'])
131         dgx['available_cpus'] -= int((task['
           cpu_milli'] + 1000 - 1) // 1000)
132         dgx['power_usage'] += additional_power
133         rackrow_usage[dgx['rackrow']] +=
           additional_power # ラック列の消費電力を
           更新
134         room_usage[dgx['room']] +=
           additional_power # サーバルームの消費電
           力を更新
135         current_Watt += additional_power
136         # タスクに割り当てられたを記録 DGX
137         task['assigned_dgx'] = dgx['id']
138         print(f"Resources allocated: DGX {dgx['id
           ']}, GPUs: {dgx['available_gpus']},

```

```

        CPUs: {dgx['available_cpus']})
139     print(f"PowerUsage is now {current_Watt
           }")
140
141     if current_Watt > DC_POW:
142         breakpoint()
143         return True
144     #print(f"Insufficient resources for Task {task
           ['name']}.")
145     if variable_RR: 可変#GANG 割当不可になると発動 GANG
146         RR = True
147     #breakpoint()
148     return False # 十分なリソースがない場合帰る
149
150     def release_resources(dgx_resources, task):
151         """タスク完了時に使用したリソースを解放する。
152
153         """
154         global current_Watt
155         for dgx in dgx_resources:
156             if dgx['id'] == task['assigned_dgx']:
157                 released_power = (task['cpu_milli'] +
                                   1000 - 1) // 1000 * XEON_TH_POW +
                                   task['num_gpu'] * task['gpu_milli']
                                   // 1000 * H100_POW
158                 dgx['available_gpus'] += int(task['
                                   num_gpu'])
159                 dgx['available_cpus'] += int((task['
                                   cpu_milli'] + 1000 - 1) // 1000)
160                 dgx['power_usage'] -= released_power
161                 rackrow_usage[dgx['rackrow']] -=
                                   released_power # ラック列の消費電力を
                                   戻す
162                 room_usage[dgx['room']] -=
                                   released_power # サーバルームの消費電力
                                   を戻す
163                 current_Watt -= released_power
164                 return
165
166     def execute_Task(clock, queue, event_times,
167                     dgx_resources):
168         """タスクを検査して
169         creation 時刻になったか判定し、適切な処理を行う。
170         """
171         temp_list = []
172         for i in range(queue.qsize()):

```

```

172         task = queue.get() # キューからタスクを取得
173
174         if any(job['name'] == task['name'] for job
175               in active_Jobs): 重複を検出して削除
176             #
177             print(f"Duplicate task '{task['name']}'
178                   found in active_Jobs. Discarding the
179                   task.")
180             continue
181
182         if clock < task['creation_time']:
183             temp_list.append(task) # まだcreation 時刻
184             に満たないものを保持
185         else:
186             # creation 時刻を迎えたタスクをscheduler に
187             投げ込む
188             #print("Throw:", task['name'])
189             if not scheduler(task, clock, event_times
190                             , dgx_resources):
191                 temp_list.append(task) # スケジューリン
192                 グに失敗したらtemp_list に戻す
193
194         # 条件を満たすタスクを再度キューに戻す
195         for task in temp_list:
196             queue.put(task)
197         return queue
198
199     def execute_Penalty(memory_mib, bandwidth):
200         return int(memory_mib // bandwidth)
201
202     def scheduler(task, clock, event_times,
203                 dgx_resources):
204         """タスクをスケジューリングする関数。スケジューリングが成
205         功した場合は、失敗した場合は
206
207         True False を返す。
208         """
209         global RR
210         global gang_activations
211
212         if task['execution_duration'] < 1:
213             return False
214         if allocate_resources(dgx_resources, task): #
215             リソースの割り
216             当て
217             #TAT['Task'].append(task['name'])
218             if RR and task['execution_duration'] >

```

```

RR_INTERVAL:
207     gang_sum =RR_INTERVAL + execute_Penalty(
        task['memory_mib'],DGX_SSD_READBW) +
        execute_Penalty(task['memory_mib'],
        DGX_SSD_WRIBW)
208     #breakpoint()
209     gang_activations += 1
210     task['deletion_time'] = int(clock +
        gang_sum)
211     task['gang'] = True
212 else:
213     task['deletion_time'] = int(clock + task
        ['execution_duration'])
214     task['gang'] = False
215
216     active_Jobs.append(task)
217     event_times.append(task['deletion_time']) #
        集合に追
        加
218     print(f"Task {task['name']} scheduled on DGX
        {task['assigned_dgx']}.")
219     return True
220 else:
221     #print(f"Task {task['name']} cannot be
        scheduled due to insufficient resources
        .")
222     return False # スケジューリングに失敗
223
224 def process_completed_tasks(clock, active_jobs,
        second_Queue, dgx_resources):
225     """完了したタスクを処理し、必要に応じてリソースを解放
        する。
226
227     """
228     global RR
229     global completed_counter
230     global glo_TAT
231
232     completed_tasks = [task for task in active_jobs
        if task['deletion_time'] == clock]
233     for task in completed_tasks:
234
235         if RR and task['gang']:
236             task['execution_duration'] -= RR_INTERVAL
237             second_Queue.put(task) # 再スケジューリング
                用にキューに戻す
238             print(f"Task {task['name']} rescheduled

```

```

        with {task['execution_duration']} ms
            remaining.")
239     else:
240         print(f"Task {task['name']} completed.")
241         TAT[task['name']] = []
242         TAT[task['name']].append(clock - task['
            creation_time'])
243         glo_TAT += clock - task['creation_time']
244         completed_counter += 1
245         release_resources(dgx_resources, task) # リ
            ソースを解放
246         active_jobs.remove(task)
247         if variable_RR 可変:#GANG
248             RR = False
249     return len(completed_tasks) # 完了したタスクの数を
        返す
250
251 def process_new_tasks(clock, data, queue):
252     """新しいタスクをキューに追加する処理
253
254     """
255     global oldclock
256     if oldclock == clock:
257         return
258
259     oldclock = clock
260     new_tasks = data[data['creation_time'] == clock
        ]
261     for _, task in new_tasks.iterrows():
262         queue.put(task.to_dict())
263         #print(f"Task added to queue: {task.to_dict
            ()}")
264
265     # シミュレーションの初期設定
266     clock = 0
267     second_Queue = Queue() # 着信キュー
268     first_Queue = Queue() # コスト超過時の優先待機キュー
269     active_Jobs = [] # 実際に稼働している配列 GPU
270     oldclock = -1
271
272     if __name__ == "__main__":
273         # の台数を初期化 DGX
274         num_Of_DGXs = init_GPU_Array()
275         print("DGX cluster is", num_Of_DGXs)
276         print("DC Power is", DC_POW, "kW")
277         # のリソース状態を初期化 DGX

```

```

278     dgx_resources = init_DGX_Resources(num_Of_DGXs)
279     print("Initialized DGX resources:",
           dgx_resources)
280
281     data = init_Data()
282     rackrow_usage = [DGX_PER_RACK * DGX_BASIC_POW]
           * NUM_OF_Rackrow # 各ラック列の消費電力を初
           期化
283     room_usage = [DGX_BASIC_POW * (num_Of_DGXs //
           NUM_OF_ROOMs)] * NUM_OF_ROOMs # 各サーバールー
           ムの消費電力を初期化
284
285     # 全イベント時間（開始時間と終了時間）をリスト化し、
           ソート
286     event_times = sorted(
287         list(data['creation_time'])
288     )
289
290     # 時計の進行に応じてシミュレーションを実行
291     for next_clock in event_times:
292         clock = next_clock
293
294         # 新しいタスクの処理
295         process_new_tasks(clock, data, second_Queue)
296         if not second_Queue.empty():
297             second_Queue = execute_Task(clock,
           second_Queue, event_times, dgx_resources
           )
298
299         # 完了したタスクの処理は処理した数return
300         deleted_Tasks = process_completed_tasks(
           clock, active_Jobs, second_Queue,
           dgx_resources)
301         if deleted_Tasks != 0:
302             if not second_Queue.empty():
303                 print("TRYING BACKFILL")
304                 event_times.append(clock+1)
305
306         clock_list.append(clock)
307         secondqueue_count.append(second_Queue.qsize
           ())
308
309         total_active_gpus = sum(int(task['num_gpu'])
           for task in active_Jobs)
310         total_active_cpus = sum(int((task['cpu_milli
           ' ] + 1000 - 1) // 1000) for task in
           active_Jobs)

```

```

311
312         print(f"Clock: {clock}, Total active GPUs in
              use: {total_active_gpus} / {num_Of_DGXs
              * 8}, Watt: {current_Watt}")
313
314         active_gpus_count.append(total_active_gpus)
315         active_jobs_count.append(len(active_Jobs))
316         watt_count.append(current_Watt)
317         event_times.sort()
318
319         print(f"Clock: {clock}")
320         #breakpoint()
321         print(f"All jobs are completed at: {clock}")
322         print(f"Total Average TAT is: {glo_TAT/
              completed_counter}")
323         print(f"GANG allowed: {gang_activations}")
324
325     plt.figure(figsize=(10, 6))
326
327     # 双対軸を作成
328     fig, ax1 = plt.subplots()
329     """
330     # つ目の軸1: Active GPUs
331     color = 'blue'
332     ax1.set_xlabel('Clock')
333     ax1.set_ylabel('Number of Active GPUs', color=
334                   color)
335     ax1.plot(clock_list, active_gpus_count, label='
336             Number of Active GPUs', color=color, marker='o
337             ')
338     ax1.tick_params(axis='y', labelcolor=color)
339     ax1.legend(loc='upper left')
340
341     # つ目の軸2: Second Queue Jobs
342     ax2 = ax1.twinx() # Twin the x-axis
343     color = 'red'
344     ax2.set_ylabel('Number of Queue Jobs', color=color
345                   )
346     ax2.plot(clock_list, secondqueue_count, label='
347             Number of Queue Jobs', color=color, marker='x')
348     ax2.tick_params(axis='y', labelcolor=color)
349     ax2.legend(loc='upper right')
350
351     color = 'green'
352     ax1.set_ylabel('Active Jobs', color=color)
353     ax1.plot(clock_list, active_jobs_count, label='

```

```

        Number of Active Jobs', color=color, marker='o
    ')
349 ax1.tick_params(axis='y', labelcolor=color)
350 ax1.legend(loc='upper left')
351
352 color = 'green'
353 plt.xlabel('Seconds')
354 ax1.set_ylabel('kWatts', color=color)
355
356 # 棒グラフに変更
357
358 ax1.step(clock_list, watt_count, where='post',
          label='kWatts', color=color, linewidth=2)
359
360 ax1.tick_params(axis='y', labelcolor=color)
361 ax1.legend(loc='upper left')
362 #ax1.set_ylim(0, DC_POW+5)
363 # タイトルとグリッド
364 fig.tight_layout()
365 plt.grid(True)
366
367 # グラフを表示
368 plt.show()
369
370 """
371
372 # キー（横軸）と値（縦軸）を取得
373 tasks = list(TAT.keys())
374 values = [v[0] for v in TAT.values()] # 各リストの
          最初の値を取得
375
376
377 # ヒストグラムの作成
378 plt.hist(values, bins=50, color='skyblue',
          edgecolor='black', alpha=0.7)
379 # 縦軸を対数スケールに設定
380 plt.yscale('log')
381
382 # ラベルとタイトル
383 plt.xlabel('Pod Execution time (Seconds)')
384 plt.ylabel('Frequency')
385
386 plt.grid(axis='y', linestyle='--', alpha=0.7)
387 plt.tight_layout()
388 plt.show()
389 """

```


また上記シミュレータのコーディング補助において、ChatGPTを使用した。使用したモデルは無料プランの GPT-4o である。以下に使用した ChatGPT のプロンプトと出力を示す。

プロンプト:python3 でスケジューリングシミュレータを作りたいと思います。csv ファイルを配列に読み込み、clock 変数を 1 ずつ増やしながら、時間に応じて csv ファイルをデータスケジューラに与えるコードを書いてもらえますか？

Listing 6.3: chatGPT output1

```
1 以下に、を使ったスケジューリングシミュレータのコード例を示します。  
   このコードでは、ファイルからタスクデータを読み込み、変数  
   をずつ増やしながら、時間に応じてタスクをスケジューラに  
   渡して実行する仕組みを作成しています。  
2  Python3CSVclock1データ例  
3  CSV以下の形式のファイルを想定します。  
4  
5  CSV  
6  
7  start_time,duration,task_name  
8  0,3,Task A  
9  2,5,Task B  
10 5,2,Task Cコード  
11  
12 Python  
13  
14 import csv  
15 import time  
16  
17 class Scheduler:  
18     def __init__(self, csv_file):  
19         self.tasks = self.load_tasks(csv_file)  
20         self.clock = 0  
21  
22     def load_tasks(self, csv_file):  
23         """ファイルからタスクデータを読み込む CSV"""  
24         tasks = []  
25         with open(csv_file, 'r') as file:  
26             reader = csv.DictReader(file)  
27             for row in reader:  
28                 tasks.append({  
29                     "start_time": int(row["start_time  
30                         "]),  
31                     "duration": int(row["duration"]),  
32                     "task_name": row["task_name"]  
33                 })  
34         return tasks
```

```

35     def run(self):スケジューラのメインループ
36         """
37         print("Starting scheduler...")
38         while self.tasks or self.clock <= max(task["
39             start_time"] for task in self.tasks):
40             print(f"Clock: {self.clock}")
41             self.execute_tasks()
42             self.clock += 1
43             time.sleep(1) # シミュレーション用に秒待機1
44
45     def execute_tasks(self):現在の時刻に対応するタスク
46     を実行
47     """
48     for task in list(self.tasks): # タスクリストを
49     コピーして反復
50         if task["start_time"] == self.clock:
51             print(f"Executing {task['task_name']}
52                 (Duration: {task['duration']}
53                 seconds) ")
54             # 実行後にタスクを削除
55             self.tasks.remove(task)
56
57     # ファイルのパスを指定 CSV
58     csv_file_path = "tasks.csv"
59
60     # スケジューラを実行
61     scheduler = Scheduler(csv_file_path)
62     scheduler.run() 実行方法上記のデータ例を
63
64     CSV tasks.csv という名前で保存します。上記のコードを
65     ファイル (例
66     Python: scheduler.) として保存します。pyターミナルま
67     たはコマンドプロンプトで以下のコマンドを実行します。
68
69     python3 scheduler.pyコードの仕組みデータの読み込み
70
71     CSV:
72     load_tasks メソッドでファイルを読み込み、タスク情
73     報をリストとして保持します。CSVスケジューラのメ
74     インループ
75
76     :

```

73 run メソッドでclock をずつ増やししながら、各タイムス
 テップでタスクをチェックして実行します。1タスク
 の実行

74

75

76

:現在の

clock に一致するstart_time を持つタスクを見つけ、
実行後にタスクリストから削除します。改良案実行
中のタスクを追跡し、終了時刻を計算してリソース
管理をシミュレーションする。複数のタスクが同時
に実行可能な並列処理を実装する。グラフやログの
出力を追加して可視化を行う。必要に応じて拡張し
てみてください！

プロンプト:clock を+1 していると実行に時間がかかってしまいます。次のイベントまで clock をスキップできませんか？

Listing 6.4: chatGPT output2

```
1 時計
2     (clock) をずつ増やすのではなく、次のイベント（タスクの開始ま
      たは終了）が発生する時間までスキップすることで、シミュレー
      ションの効率を大幅に向上できます。1以下にその実装例を示し
      ます。改良コード例
3
4
5
6     import pandas as pd
7
8     # ファイルのパス CSV
9     csv_file_path = "/mnt/data/openb_pod_list_default.csv"
10
11    # データを読み込む
12    data = pd.read_csv(csv_file_path)
13
14    # 空欄のscheduled_time をcreation_time で埋める
15    data['effective_scheduled_time'] = data['
      scheduled_time'].fillna(data['creation_time'])
16
17    # execution_duration の計算
18    data['execution_duration'] = data['deletion_time'] -
      data['effective_scheduled_time']
19
20    # シミュレーションの初期設定
21    clock = 0
22    current_tasks = [] # 現在のタスクを保持するリスト
23
24    # 全イベント時間（開始時間と終了時間）をリスト化し、ソート
25    event_times = sorted(
26        list(data['effective_scheduled_time']) +
27        list(data['effective_scheduled_time'] + data['
      execution_duration']))
28    )
29
30    # 時計の進行に応じてシミュレーションを実行
31    for next_clock in event_times:
32        # 次のイベント時間までスキップ
33        clock = next_clock
34
35        # 新しいタスクをcurrent_tasks に追加
36        new_tasks = data[(data['effective_scheduled_time']
      == clock)]
```

```

37     current_tasks.extend(new_tasks.to_dict('records'))
38
39     # 経過したタスクを削除
40     current_tasks = [
41         task for task in current_tasks
42         if clock < task['effective_scheduled_time'] +
           task['execution_duration']
43     ]
44
45     # ログを表示
46     print(f"Clock: {clock}")
47     print(f"Current tasks: {[task['name'] for task in
           current_tasks]}")改善点の説明
48
49
50
51     event_times の準備: (タスクの開始時間) と
52     effective_scheduled_time
           effective_scheduled_time + (タスクの終了時間)
           をリスト化し、すべてのイベント時間を取得します。
           execution_duration これらをソートして順番に処理し
           ます。
53
54
55     for ループでイベント処理:
56         clock を次のイベント時間 () に直接ジャンプさせます。
           next_clock イベント処理
57
58     :新しいタスクの追加と、終了したタスクの削除を行います。時間
           効率の向上
59
60
61     :単位ずつ進める代わりに、必要な時刻にジャンプするため、処理
           速度が大幅に向上します。
           1出力例以下はログの一部の例です:
62
63
64
65
66
67
68     Clock: 0
69     Current tasks: ['openb-pod-0000']
70     Clock: 427061
71     Current tasks: ['openb-pod-0000', 'openb-pod-0001']
72     Clock: 1558381
73     Current tasks: ['openb-pod-0001', 'openb-pod-0002']

```

74 Clock: 12537496

75 Current tasks: [] 応用この方法は大量のデータや長期間のシミュレーションにも適しています。必要に応じて、イベント処理時にさらに複雑なロジックを追加できます。必要に応じて他の機能を組み込む方法についてもお手伝いできます！

プロンプト: 棒グラフの縦軸を対数にしてください

Listing 6.5: chatGPT output3

```
1
2 ChatGPT:
3 ChatGPT縦軸を対数スケールにすることで、分布の偏りや小さな頻度
  も見やすくなります。以下が縦軸を対数にしたヒストグラムの
  コード例です！コード例
4
5
6
7
8 import matplotlib.pyplot as plt
9
10 # ヒストグラムの作成（縦軸を対数スケールにする）
11 plt.hist(values, bins=20, color='skyblue', edgecolor
  = 'black', alpha=0.7)
12
13 # 縦軸を対数スケールに設定
14 plt.yscale('log')
15
16 # ラベルとタイトル
17 plt.xlabel('Execution Duration (Values)')
18 plt.ylabel('Log Frequency')
19 plt.title('Log-Scaled Frequency Distribution of
  Execution Durations')
20 plt.grid(axis='y', linestyle='--', alpha=0.7)
21
22 # レイアウトを調整して表示
23 plt.tight_layout()
24 plt.show() 補足
25
26
27
28 plt.yscale('log')：これで縦軸を対数スケールに変更してい
  ます。
29 bins：個の区間に分けていますが、値が多い部分や少ない部分の
  詳細を確認する場合、=2020を増減させることで調整可能で
  す。binsグリッド線：縦軸にグリッド線を引くことで、値の
  分布が読みやすくなっています。試すポイント範囲指定 ()
30
31
32
33
34 optional必要に応じて範囲を制限するため、
35 range=(min_value, max_value)を追加します：
36
37 plt.hist(values, bins=20, range=(0, 15000000),
```

```
color='green', edgecolor='black', alpha=0.7)細  
かい分布確認をやに増やすと、分布がより詳細に確認でき  
ます。
```

38

39

40

```
bins3050これで分布の偏りや外れ値を視覚的に捉えやすくなりま  
す！試してみてください
```
