Master Thesis

Significant Runtime Reduction for Asynchronous Multi-Fidelity Optimization on Zero-Cost Benchmarks

Shuhei Watanabe

Examiner:	Prof. Dr. Frank Hutter
Second Examiner:	Dr. Katharina Eggensperger
Advisers:	Neeratyoy Mallik
	Edward Bergman

University of Freiburg
Faculty of Engineering
Department of Computer Science
Machine Learning Lab

October 15th, 2023

Writing Period

11. 08. 2023 – 15. 10. 2023

Examiner

Prof. Dr. Frank Hutter

Second Examiner

Dr. Katharina Eggensperger

Advisers

Neeratyoy Mallik & Edward Bergman

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Abstract

As the research in deep learning evolved, its outstanding performance has attracted the attention of practitioners. However, its performance is still susceptible to its hyperparameter (HP) selection and it highlighted the importance of hyperparameter optimization (HPO). Since the training of a deep learning model typically takes hours to days, HPO is prohibitively expensive and it necessitates zero-cost benchmarks, i.e. benchmarks that query the performance metric and runtime of a specific HP configuration in a fraction such as tabular and surrogate benchmarks. Although zero-cost benchmarks reduce the runtime of experiments significantly in non-parallel setups, parallel setups cannot simply benefit from zero-cost benchmarks because each worker must communicate the return order of each HP configuration based on their queried runtime. In this thesis, we develop an easy-to-use Python package to resolve the issue. We first derive the formulation for the wrapper and provide the algorithm, which lets each worker communicate via file system and forces them to wait only for a negligible amount of time to obtain the exact return order. We comprehensively validate the correctness of our implementation using toy cases as proof of concept. In the experiments using zero-cost benchmarks, we demonstrate that our wrapper could finish the whole set of experiments 1.3×10^3 times faster compared to the implementation naïvely waiting for each queried runtime and show the baseline performance of optimizers used in major HPO libraries.

Acknowledgements

First and foremost, I would like to thank Frank for the opportunity to work with him and his group. Especially, I could not have presented some of my papers at international conferences and their workshops without his financial support (and thank Lina for handling the complicated paperwork many times!). I also would like to thank the co-authors of these papers – Archit, Noor, Ozaki-san, Nomura-san, Akimoto-sensei, Onishi-sensei, and Frank – although none of the topics are covered by this thesis. Talking about the beginning of this thesis, I really appreciate Neeratyoy for his offer to be my PhD supervisor especially because the aforementioned papers did not have proper supervisors due to the topic choices and I really had a hard time finding a PhD supervisor for my Master thesis. Originally, we were thinking of the enhancement of BOHB, but I realized that experiments on parallel setups would take a substantial amount of time even if we use tabular benchmarks. I really did not want to spend time for waiting results – naïve simulation defined in this thesis! –, so I created software to avoid it and talked to Neeratyoy about it. Although it was very spontaneous (, yet we later found out that this software was exactly what *HPOBench* seeks as written in its future work and that is how Katharina was appointed as a second examiner!), he accepted my proposal and we finally reached the end of my Master thesis. Also, I am grateful to Eddie for some advice on my code. I really enjoyed talking to him about readable Python coding and trusted his advice about Python coding the most because he is the only person who is actively catching up with the latest Python coding as far as I know.

When I look back, my Master study was a long journey. I first need to show my appreciation to Onishi-sensei from AIST for letting me study there before I came to Freiburg and Ito Foundation for International Education Exchange, Deutschland-stipendium, and ELIZA for their financial support. Since I came here during the COVID-19 pandemic time, I really could not make real friends, but my family and some friends from Tokyo – Paula, Øyvind, Serizawa, Chikaraishi – supported me mentally. I also would like to thank some of my few study friends – Simon, Baohe, Yedil,

and Goktug. Besides them, I really appreciate the company I will join – Preferred Networks Inc. – to permit me to extend my stay here for half a year. Finally, I thank all of my flatmates – Emi, Fenja, Jojo-san, Laurie, Marios, Melli, Oibinto, and Sarah (alphabetical order and no intention in the order, so no fighting please!) – for the cheerful year!

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

Hyperparameter (HP) optimization of deep learning is crucial for strong performance [1, 2] and it surged the research on HP optimization (HPO) of deep learning. However, due to the heavy computational nature of deep learning, HPO is often prohibitively expensive and both energy and time costs are not negligible. This is the driving force behind the emergence of zero-cost benchmarks such as tabular and surrogate benchmarks, which enable yielding the (predictive) performance of a specific HP configuration in a small amount of time [3, 4, 5, 6].

Although these benchmarks effectively reduce the energy usage and the runtime of experiments in many cases, experiments considering runtimes between parallel workers may not be easily benefited as seen in Figure 1. For example, multi-fidelity optimization (MFO) [7] has been actively studied recently due to its computational efficiency [8, 9, 10, 11], but because of its asynchronous nature, the call order of each worker must be appropriately sorted out to not break the states that the actual experiments would go through. While this problem is naïvely addressed by making each worker wait for the runtime the actual deep learning training would take, each worker must wait for a substantial amount of time in this case, and hence it ends up wasting energy and time.

To address this problem, we developed an easy-to-use Python wrapper (see Figure 2 for the simplest codeblock and Figure 3 for the conceptual visualization) that automatically sorts out the right release order of evaluations for each worker in a fraction and forces each worker to sleep in order to match the apparent evaluation order from the optimizer perspective. Since optimizers have different characteristics and their libraries use different multi-core processing methods, our wrapper needs to be able to support such diverse libraries. For example, BOHB [10] uses file servers to share observations across workers and DEHB [11] uses `dask`, which is a multiprocessing-based library. Furthermore, Bayesian optimization (BO) typically takes more time to sample due to fantasization [12] or a long time for the optimization of acquisition

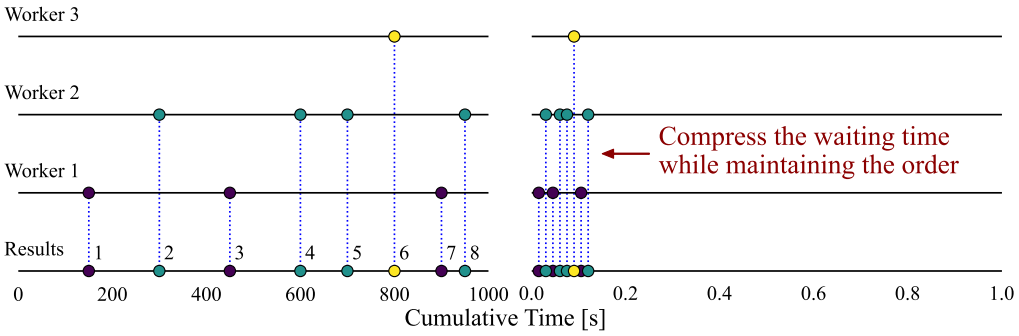


Figure 1.: The conceptual visualization of the runtime compression in an asynchronous optimization. We need to make sure the return order of each result is preserved after we compress the runtime because some optimizers such as Bayesian optimization train a surrogate model using the observations available at each iteration.

functions [13]. To clarify the application scope of our wrapper, we would specifically like to answer the following research questions:

RQ1: What kind of algorithm is valid for an optimizer that uses multiple workers concurrently depending on its sampling cost?

RQ2: Do the results obtained by our wrapper match the results obtained by naïvely sleeping for the runtime of each query?

Note that we call an optimizer that uses multiple workers concurrently *multi-core optimizer* hereafter. Even with the solutions to these research questions above, our wrapper still suffers from issues common in multi-core processing optimizers such as deadlock, a race condition, and latency due to the communication between workers. To alleviate these issues, we delve into the possibility of using only a single core to simulate asynchronous optimizations, and thus the third research question is:

RQ3: Is it possible to simulate asynchronous optimization only with a single core?

The single-core simulation enables researchers to perform experiments on parallel setups even without explicitly coding multi-core optimization. In this thesis, we will give the mathematical formulation for our algorithm under some assumptions and empirically test our algorithm on several test cases that could be the edge cases of possible wrapper algorithms.

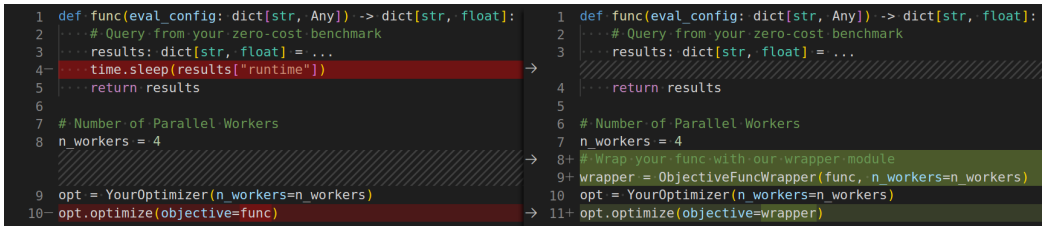


Figure 2.: The simplest codeblock example of how our wrapper works. The exact tool usage is discussed in Appendix B. **Left**: a codeblock example without our wrapper (naïve simulation). We let each function call sleep for the time specified by the queried result. Although this example seems very naïve, this guarantees the implementation correctness easily and researchers often use this implementation because it is hard to correctly simulate asynchronous optimization for optimizers from other researchers in a non-naïve simulation manner. **Right**: a codeblock example with our wrapper (multi-core simulation). Users only need to wrap the objective function with our module and remove the line for sleeping while reproducing the identical result obtained by the naïve simulation.

Last but not least, many optimizers have been invented so far, but they were mostly compared in the non-parallel setup. Therefore, the fourth research question is:

RQ4: Is it necessary to test the performance of optimizers on the parallel setup?

In the experiments, we use various open source software (OSS) optimizer libraries such as SMAC3 [14] and Optuna [15] on zero-cost benchmarks and we compare the changes in the performance based on the number of parallel workers to use. The experiments not only provide the baselines but also showed that our wrapper finished all the experiments 1.3×10^3 times faster than the naïve simulation (see Figure 2 (**Left**)). The implementation for the experiments in this thesis is also publicly available [1].

In summary, the contributions of this thesis are to:

1. provide the algorithms to simulate an asynchronous optimization using a multi-core optimizer in an online manner with the exact mathematical formulation (the answer to **RQ1**),

2. make the software publicly available and test the software using several examples to empirically validate that our implementation is correct irrespective of a sampling cost of an optimizer (the answers to **RQ2**),

[1] `https://github.com/nabenabe0928/master-thesis-experiment`

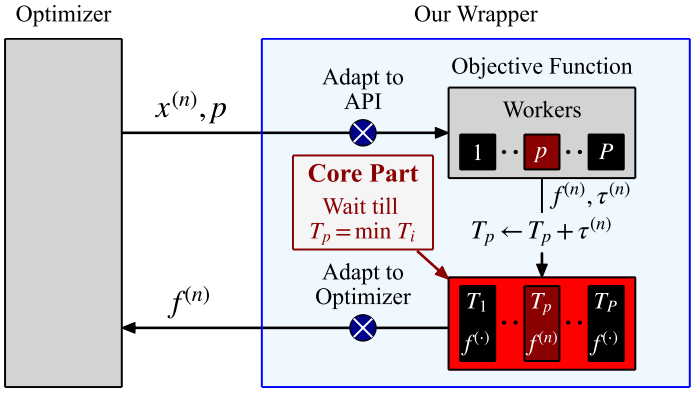


Figure 3.: The conceptual visualization of our wrapper. Our wrapper yields the exact return order to be reproduced by forcing each worker to sleep until the exact timing when it should return the latest result. Users only need to wrap their own function by our wrapper and there is no need to change optimizer parts.

3. provide an option to simulate only with a single core, which allows researchers to perform experiments in parallel setups even without coding a multi-core algorithm (the answer to **RQ3**), and

4. demonstrate that our experiment, whose code is available in public [2], could speed up the whole runtime of the series of experiments 1.3×10^3 times and that experiments for parallel setup are necessary by showing that the relative performance of major optimizers depend on the number of workers using a statistical test (the answer to **RQ4**).

[2] `https://github.com/nabenabe0928/master-thesis-experiment`

2. Background

In this chapter, we formally introduce asynchronous optimization in different scenarios and explain MFO, which is one of the most common problem setups for asynchronous optimization.

2.1. Preliminaries and Notations

We first note that the n-th sample $\boldsymbol{x}^{(n)}$ and the sample of the n-th observation $\boldsymbol{x}_n$ are different concepts. More specifically, samples $\boldsymbol{x}^{(n)}$ are ordered by the sampling order, and observations $\boldsymbol{x}_n$ are ordered by the return order which means the i-th result $(\boldsymbol{x}_i, \tilde{f}_i)$ is delivered to a set of observations $\mathcal{D}_N$ earlier than the j-th result $(\boldsymbol{x}_j, \tilde{f}_j)$ for an arbitrary pair of (i, j) where $i < j$. For example, if there are two workers and an optimizer samples two HP configurations $\boldsymbol{x}$ and $\boldsymbol{x}'$ in this order, $\boldsymbol{x}^{(1)} \coloneqq \boldsymbol{x}$ and $\boldsymbol{x}^{(2)} \coloneqq \boldsymbol{x}'$. On the other hand, if the evaluations for $\boldsymbol{x}^{(1)}$ and $\boldsymbol{x}^{(2)}$ take 20 seconds and 10 seconds respectively, $\boldsymbol{x}_1 \coloneqq \boldsymbol{x}^{(2)}$ and $\boldsymbol{x}_2 \coloneqq \boldsymbol{x}^{(1)}$ because the result of $\boldsymbol{x}^{(2)}$ will come first and that of $\boldsymbol{x}^{(1)}$ will come next. Throughout this thesis, we use the following notation:

- $[i] \coloneqq \{1, \ldots, i\}$, a set of integers from 1 to i,
- $\mathcal{X}_d \subseteq \mathbb{R}$ (for $d \in [D]$), a domain of the d-th hyperparameter,
- $\mathcal{X} \coloneqq \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_D \subseteq \mathbb{R}^D$, a search space,
- $\boldsymbol{x} \in \mathcal{X}$, a hyperparameter configuration in the search space $\mathcal{X}$,
- $\boldsymbol{f}(\boldsymbol{x}) : \mathcal{X} \to \mathbb{R}^M$, an objective function [1],
- $\tau(\boldsymbol{x}) : \mathcal{X} \to \mathbb{R}_+$, a runtime function,

[1] In the thesis, we use a single-objective function with $M = 1$ for notational simplicity, but our wrapper is applicable to functions with $M > 1$ as well.

- $\tilde{f} \in \mathbb{R}$: an observation of the objective function given an HP configuration $\boldsymbol{x}$ with a noise ϵ,
- $\tilde{\tau} \in \mathbb{R}_+$: an observation of the runtime function given an HP configuration $\boldsymbol{x}$ with a noise ϵ,
- $\mathcal{D}_N \coloneqq \{(\boldsymbol{x}_n, \tilde{f}_n)\}_{n=1}^N$, a set of observations,
- $\boldsymbol{x}_n \in \mathcal{X}$, the n-th observation in $\mathcal{D}_N$,
- $\pi(\boldsymbol{x}|\mathcal{D}_N)$, a policy of an optimizer,
- $\boldsymbol{x}^{(n)} \in \mathcal{X}$, the n-th sample,
- $\tilde{t}^{(n)} \in \mathbb{R}_+$, the sampling time, i.e. the time interval between the end of the last evaluation and the beginning of the next evaluation, for the n-th sample $\boldsymbol{x}^{(n)}$,
- $\tilde{t}_n \in \mathbb{R}_+$, the sampling time, i.e. the time interval between the end of the last evaluation and the beginning of the next evaluation, for the sample of the n-th observation $\boldsymbol{x}_n$,
- $P \in \mathbb{Z}_+$: the number of parallel workers,
- $\boldsymbol{a}^{(n)}$, a set of arguments used to evaluate the n-th sample $\boldsymbol{x}^{(n)}$ such as random seed, intermediate model state, and fidelity parameters,
- $W_p : \mathcal{X} \to \mathbb{R}^2$, the p-th ($p \leq P$) worker that returns $\tilde{f}$ and $\tilde{\tau}$, or their estimations given a set of arguments $\boldsymbol{a}$,
- $p^{(n)} : \mathbb{Z}_+ \to [P]$: an index specifier of which worker processed the result of n-th sample $(\boldsymbol{x}^{(n)}, \tilde{f}^{(n)})$,
- $\mathcal{I}_p^{(N)} \coloneqq \{n \in [N] \mid p^{(n)} = p\}$, a set of the indices of samples the p-th worker processed,
- $\tilde{w}^{(n)} \in \mathbb{R}_+$, the waiting time for the n-th sample $\boldsymbol{x}^{(n)}$, i.e. the time between when the sampling for $\boldsymbol{x}^{(n)}$ is requested and when the sampling actually starts,
- $\tilde{w}_n \in \mathbb{R}_+$, the waiting time for the sample of the n-th observation $\boldsymbol{x}_n$, i.e. the time between when the sampling for $\boldsymbol{x}_n$ is requested and when the sampling actually starts, and
- $T_p^{(N)} \in \mathbb{R}_+$, the simulated runtime of the p-th worker using the results of up to the N-th sample (see Eq. (2) for the formal definition).

Furthermore, we use the following terminologies:

Definition 1 (Optimizer)**.** *Given a search space $\mathcal{X}$ and a set of observations $\mathcal{D}_N$, an optimizer is used to sample an HP configuration based on a policy, i.e. $\boldsymbol{x} \sim \pi(\boldsymbol{x}|\mathcal{D}_N)$.*

Definition 2 (Surrogate Benchmark)**.** *Given a set of actual observations $\{(\boldsymbol{x}_n, \tilde{f}_n, \tilde{\tau}_n)\}_{n=1}^N$, a surrogate benchmark trains machine learning models for an objective metric $\hat{f}(\boldsymbol{x}) : \mathcal{X} \to \mathbb{R}$ and for runtime $\hat{\tau}(\boldsymbol{x}) : \mathcal{X} \to \mathbb{R}_+$, and can query the estimated objective metric and the estimated runtime for an arbitrary HP configuration $\boldsymbol{x}$ with a negligible amount of time $\mathcal{T} \ll \tilde{\tau}$.*

Definition 3 (Tabular Benchmark)**.** *Given a discrete search space $\mathcal{X} \coloneqq \{\boldsymbol{x}_n\}_{n=1}^{N_{\text{all}}}$, a tabular benchmark records an objective metric and runtime of all the possible HP configurations and can query the actual objective metric and the actual runtime for an arbitrary HP configuration $\boldsymbol{x} \in \mathcal{X}$ with a negligible amount of time $\mathcal{T} \ll \tilde{\tau}$.*

Examples of surrogate benchmarks are JAHS-Bench-201 [6] and the benchmarks introduced by Eggensperger *et al.* [3] and examples of tabular benchmarks are HPOLib [16] and NAS-Bench-201 [17]. Note that we assume that the query cost of these benchmarks is zero in this thesis. Furthermore, we define the following terminologies:

Definition 4 (Zero-Cost Benchmark)**.** *Zero-cost benchmark is a tabular or surrogate benchmark for a task that exhibits a large runtime $\tilde{\tau}$.*

Definition 5 (Actual Runtime)**.** *Actual runtime is the wall-clock time of an experiment. For example, an actual runtime of a training of a deep learning model with an HP configuration, i.e. $\tilde{f}$, would be several hours and an actual runtime of HPO that trains deep learning models to find an optimal HP configuration would be several days. On the other hand, an actual runtime of HPO that uses a zero-cost benchmark is typically several seconds to minutes.*

Definition 6 (Simulated Runtime)**.** *Simulated runtime is the estimated wall-clock time of an experiment using our wrapper. For example, a simulated runtime of HPO that uses a zero-cost benchmark with our wrapper should ideally coincide with the*

actual runtime of HPO that actually trains deep learning models even though the actual runtime of the experiment takes only several seconds to minutes.

Definition 7 (Worker)**.** *After an optimizer samples an HP configuration $\boldsymbol{x}$, an optimizer passes $\boldsymbol{x}$ to a worker and the worker processes $\boldsymbol{x}$ to obtain $\tilde{f}$.*

Definition 8 (Runtime of Worker)**.** *When an experiment uses a zero-cost benchmark, the actual runtime of a worker for an HP configuration $\boldsymbol{x}$, which is a query cost, is almost zero while the simulated runtime is $\tilde{\tau}$. When an experiment does not use a zero-cost benchmark, both the actual runtime and the simulated runtime of a worker for an HP configuration $\boldsymbol{x}$ are $\tilde{\tau}$.*

Definition 9 (Ask-and-Tell Interface [2])**.** *Ask-and-tell interface is an interface of an optimizer library such that both a method that asks for an HP configuration $\boldsymbol{x}$ and a method that tells a result $(\boldsymbol{x}, \tilde{f})$ to the optimizer are provided to users.*

2.2. Parallel Optimization

In this section, we describe the formal definitions of each problem setup.

2.2.1. Asynchronous Optimization with Cheap Optimizer

Assume we have a zero-cost benchmark, P workers, and a cheap optimizer, i.e. a sampling time $\tilde{t}^{(n)}$ is negligible, if we use the notations from Section 2.1, then the simulated runtime of the p-th worker is computed as follows:

$$T_p^{(N)} \coloneqq \sum_{n \in \mathcal{I}_p^{(N)}} \tilde{\tau}^{(n)}. \tag{1}$$

In turn, the $(N+1)$-th sample will be processed by the worker that will be free for the first time, and thus the index of the worker for the $(N+1)$-th sample is specified by $\mathrm{argmin}_{p \in [P]} T_p^{(N)}$. Since the sampling time $\tilde{t}^{(n)}$ is negligible, the freed worker immediately receives another configuration $\boldsymbol{x}^{(N+1)}$ and no less than one worker

[2] See `https://optuna.readthedocs.io/en/stable/tutorial/20_recipes/009_ask_and_tell.html`

will be free simultaneously in this scenario. In other words, $\boldsymbol{x}^{(N+1)}$ is sampled from $\pi(\boldsymbol{x}|\mathcal{D}_{N-P+1})$ for $N+1 > P$.

2.2.2. Asynchronous Optimization with Expensive Optimizer

Assume we have a zero-cost benchmark, P workers, and an expensive optimizer, i.e. a sampling time $\tilde{t}_n$ is non-negligible, if we use the notations from Section 2.1, then the simulated runtime of the p-th worker is computed as follows:

$$T_p^{(N)} \coloneqq \sum_{n \in \mathcal{I}_p^{(N)}} (\tilde{w}^{(n)} + \tilde{t}^{(n)} + \tilde{\tau}^{(n)}). \tag{2}$$

In the same vein, the $(N+1)$-th sample will be processed by one of the workers that is free at the moment, and thus the index of the worker for the $(N+1)$-th sample is $\operatorname{argmin}_{p \in [P]} T_p^{(N)}$. Since the sampling time is non-negligible, (an)other worker(s) $W_{p'}$ may finish their evaluation during a sampling for the freed worker W_p, i.e. $\exists p' \in [P] \setminus \{p\}, T_p^{(N)} < T_{p'}^{(N)} < T_p^{(N)} + \tilde{t}^{(N+1)}$. It may cause waiting times for some samplings, i.e. $\tilde{w}^{(n)} > 0$. If an optimizer can sample multiple HP configurations in parallel, the waiting time $\tilde{w}^{(n)}$ becomes always zero. Therefore, the simulated runtime becomes:

$$T_p^{(N)} \coloneqq \sum_{n \in \mathcal{I}_p^{(N)}} (\tilde{t}^{(n)} + \tilde{\tau}^{(n)}). \tag{3}$$

Since the waiting times are always zero and the $(N+1)$-th sample is obtained from $\pi(\boldsymbol{x}|\mathcal{D}_{N-P+1})$, this setup is identical to the one in Section 2.2.1 in principle. However, if an optimizer needs to sample HP configurations one by one, we may have $N-P+1$ observations or more when the $(N+1)$-th sample is sampled. This implies that the one-to-one exchange (Formally defined in Def. 16) as in Section 2.2.1 does not hold in this scenario.

2.2.3. Multi-Fidelity Optimization (MFO)

Multi-fidelity optimization (MFO) [7] is an optimization that uses cheaper-to-evaluate low-fidelity functions instead of the true objective function; see Section 3.2 for more details of existing MFO methods. For example, when we optimize an HP configuration of a deep learning model with training epochs of 200, 20-epoch training of the deep learning model with an HP configuration $\boldsymbol{x}$ would be a low-fidelity function for the

training with the same HP configuration. As the goal of MFO is to save runtime as much as possible, practitioners often combine MFO with asynchronous optimization. More formally, we define a fidelity space as $\mathcal{Z} \coloneqq \mathcal{Z}_1 \times \cdots \times \mathcal{Z}_K$, typically taken as $[0,1]^K$ and we also stick to $[0,1]^K$ in this thesis, and a function that takes a fidelity vector as $g(\boldsymbol{x}, \boldsymbol{z})$ where $\boldsymbol{z} \in [0,1]^K$, we assume $f(\boldsymbol{x}) = g(\boldsymbol{x}, \mathbf{1}^K)$ is the highest-fidelity function, and $\mathbf{1}^K \coloneqq [1, \ldots, 1] \in [0,1]^K$. Then a low-fidelity function of $f(\boldsymbol{x})$ is an arbitrary function $g(\boldsymbol{x}, \boldsymbol{z})$ that takes $\boldsymbol{z} \neq \mathbf{1}^K$. In MFO, our aim is to optimize the highest-fidelity function $f(\boldsymbol{x})$ while evaluating low-fidelity functions to save the actual runtime. Although this is an abuse of notation, suppose the runtime function τ takes the fidelity vector as well and we have multiple fidelity vectors $\{\boldsymbol{z}_l\}_{l=1}^L$ [3] such that $|f(\boldsymbol{x}) - g(\boldsymbol{x}, \boldsymbol{z}_1)| \geq |f(\boldsymbol{x}) - g(\boldsymbol{x}, \boldsymbol{z}_2)| \geq \cdots \geq |f(\boldsymbol{x}) - g(\boldsymbol{x}, \boldsymbol{z}_L)|$ (and typically $\tau(\boldsymbol{x}, \boldsymbol{z}_1) \leq \tau(\boldsymbol{x}, \boldsymbol{z}_2) \leq \cdots \leq \tau(\boldsymbol{x}, \boldsymbol{z}_L)$), then a function $g(\boldsymbol{x}, \boldsymbol{z}_i)$ is said to have lower fidelity than $g(\boldsymbol{x}, \boldsymbol{z}_j)$ for an arbitrary pair of (i, j) where $i < j$. The reason why MFO is called *multi-fidelity* is that there are multiple choices of fidelity vectors $\boldsymbol{z}$, but not the dimension of the fidelity space K is larger than 1. Therefore, single-fidelity optimization, which uses only $\boldsymbol{z} = \mathbf{1}^K$, is a simple single-objective optimization problem. In the research context, many methods consider $K = 1$ only, and *multi-fidelity* sometimes implies MFO with $K = 1$ and *many-fidelity* implies MFO with $K > 1$. Related to MFO, we define the following terminologies and we consistently use these terminologies throughout this thesis:

Definition 10 (Many-Fidelity Optimization)**.** *Many-fidelity optimization is the multi-fidelity optimization with $K > 1$.*

Definition 11 (Continual Setup)**.** *For the multi-fidelity optimization with $K = 1$, if an evaluation of $g(\boldsymbol{x}, z)$ can be continued from $g(\boldsymbol{x}, z')$ for $z' < z$, we call this setup of multi-fidelity optimization "continual setup".*

For example, when we have a neural network with an HP configuration $\boldsymbol{x}$ trained for 20 epochs and we would like to train the same network with $\boldsymbol{x}$ for 100 epochs, we can restart the training from 20 epochs rather than starting it from scratch. On the other hand, if we train a neural network with an HP configuration $\boldsymbol{x}$ trained using only 10% of a full dataset and we would like to train the same network with $\boldsymbol{x}$ on the full dataset, we cannot restart from the result on 10% of the full dataset. The first setup is continual setup and the second setup is non-continual setup. Furthermore, we define the following terminology:

[3] $\boldsymbol{z}_L$ takes $\mathbf{1}^K$, which is the highest fidelity.

Definition 12 (Evaluation Restart)**.** *If we evaluate $g(\boldsymbol{x}, z)$ using an intermediate state $g(\boldsymbol{x}, z')$ where $z < z'$, we call this evaluation "evaluation restart".*

2.3. Optimization Using Zero-Cost Benchmark

In this section, we explain how optimizations using a zero-cost benchmark work. Especially, we focus on how to save experiment runtimes with a zero-cost benchmark. Note that parallel setup is classified into either synchronous or asynchronous setup.

2.3.1. Non-Parallel Setup

In general, an optimization procedure for a single worker works as follows:

1. A set of observations is initialized to $\mathcal{D}_0 = \emptyset$ and we set $N = 0$,
2. An optimizer samples an HP configuration $\boldsymbol{x}_{N+1}$ from its policy $\pi(\boldsymbol{x}|\mathcal{D}_N)$,
3. A worker evaluates $\tilde{f}_{N+1}$ and appends $(\boldsymbol{x}_{N+1}, \tilde{f}_{N+1})$ to $\mathcal{D}_N$,
4. N is incremented to $N + 1$, and
5. An optimizer repeats 2. – 4. until its budget runs out.

The stochastic part of this procedure lies in Process 2 where we sample an HP configuration from the policy $\pi(\boldsymbol{x}|\mathcal{D}_N)$. As the policy π is conditioned on $\mathcal{D}_N$, the optimization results of both using and not using a zero-cost benchmark match as long as $\mathcal{D}_N$ at each iteration is identical. In non-parallel setup, we evaluate each HP configuration one by one, so the observation order is same as the sampling order and this is why we can simply replace the actual evaluation of $\tilde{f}$ with a zero-cost benchmark.

2.3.2. Synchronous Setup

When an optimization method requires G HP configurations to update its policy $\pi(\boldsymbol{x}|\mathcal{D}_N)$, we call this setup *synchronous setup*. In this setup, an optimization method works as follows:

1. A set of observations is initialized to $\mathcal{D}_0 = \emptyset$ and we set $i = 0$,

2. An optimizer samples G HP configurations $\{\boldsymbol{x}_{iG+j}\}_{j=1}^{G}$, i.e. the i-th generation, from its policy $\pi(\boldsymbol{x}|\mathcal{D}_{iG})$,

3. Each HP configuration is assigned to one of the available workers or waits until a worker becomes available if no worker is free,

4. An optimizer waits until all the evaluations $\tilde{f}_{iG+1}, \ldots, \tilde{f}_{iG+G}$ come and updates the observations to $\mathcal{D}_{(i+1)G} = \mathcal{D}_{iG} \cup \{(\boldsymbol{x}_{iG+j}, \tilde{f}_{iG+j})\}_{j=1}^{G}$,

5. i is incremented to $i+1$, and

6. An optimizer repeats 2. – 5. until its budget runs out.

Similarly to the non-parallel setup, synchronous setup is also straightforward because the observation order in the i-th generation does not affect the stochasticity of the sampling policy and all the workers simply need to wait until the last observation in the i-th generation comes. Although it is not trivial to appropriately allocate each HP configuration to workers in an online manner, we can perfectly reproduce the exact observation order and the simulated runtime $T_p^{(n)}$ for an arbitrary pair of $(n, p) \in [N] \times [P]$ in a post-hoc way as long as we store the optimization results $\{(\boldsymbol{x}_n, \tilde{f}_n, \tilde{\tau}_n)\}_{n=1}^{N}$. Since it is possible to reproduce the simulated runtime $T_p^{(n)}$ at any point for an arbitrary $p \in [P]$, we can simply run an experiment with a single worker and calculate simulated times for any $p \in [P]$ later. Although we provide the functionality to reproduce synchronous setup in our wrapper, we do not discuss the details in this thesis.

2.3.3. Asynchronous Setup

When an optimizer updates its policy every time it receives a result and there are multiple workers, we call this setup *asynchronous setup*. In this setup, an optimizer works as follows:

1. A set of observations is initialized to $\mathcal{D}_0 = \emptyset$ and we set $N = 0$,

2. An optimizer samples an HP configuration $\boldsymbol{x}^{(N+1)}$, i.e. the $(N+1)$-th sample, from its policy $\pi(\boldsymbol{x}|\mathcal{D}_n)$ where $N - P + 1 \leq n \leq N$ and n mostly depends on the sampling time at each iteration,

3. the p-th worker where $p = \text{argmin}_{p' \in [P]} T_{p'}^{(N)}$ evaluates $\tilde{f}^{(N+1)}$ and appends $(\boldsymbol{x}^{(N+1)}, \tilde{f}^{(N+1)})$ to $\mathcal{D}_{N'}$ [4],

4. N' is incremented to $N' + 1$, and

5. An optimizer repeats 2. – 4. until its budget runs out.

Unlike the other setups, the exact observation order and the exact set of observations for asynchronous optimization cannot be reproduced in a post-hoc manner because the policy π gives us a wrong result unless the set of observations at each sampling timing is exact. Therefore, the exact observation order must be preserved when using a zero-cost benchmark. The most naïve way to achieve this is the following:

Definition 13 (Naïve Simulation)**.** *For each query $\boldsymbol{x}^{(n)}$, the allocated worker sleeps for $\tilde{\tau}^{(n)}$ seconds.*

With the naïve simulation (see also Figure 2 (**Left**)), we can simply reproduce the exact observation order. However, the naïve simulation cannot benefit in terms of the runtime reduction, and thus we need to invent a wrapper that lets each worker sleep only for a small amount of time while maintaining the exact observation order.

2.4. Parallel Processing in Hyperparameter Optimization Libraries

In this section, we describe two different designs of hyperparameter optimization libraries and their asynchronous parallel processing mechanism. Throughout this thesis, we assume that a sampler runs on the main process, which means multiple HP configurations cannot be sampled simultaneously, and distributes each HP configuration to one of the workers created in a child process or a child thread. Note that although we do not discuss parallel sampling in this thesis, our wrapper also supports parallel samplers.

[4] Since the evaluation of $\tilde{f}^{(N+1)}$ requires $\tilde{\tau}^{(N+1)}$ seconds, N becomes $N' \geq N$ while we wait for the evaluation.

2.4.1. User Perspective of Hyperparameter Optimization Libraries

Since HPO libraries optimize an objective function $f(\boldsymbol{x})$ given its search space $\mathcal{X}$ (and possibly its fidelity space $\mathcal{Z}$ as well), libraries require users to provide: (1) an objective function or its wrapper that has the compatible signatures for each library, (2) its search space $\mathcal{X}$, (3) its fidelity space $\mathcal{Z}$, and (4) optimization setup parameters. Examples of optimization setup parameters are such as the optimization direction, i.e. minimization or maximization, the number of objectives M, whether to use MFO, and an optimization budget. Most libraries first take all four elements with compatible signatures or formats. Then once libraries trigger an optimization, they internally repeat a suggestion of an HP configuration and its evaluation until the specified budget runs out [5].

In the optimization loop, an optimizer repeats (1) sampling of an HP configuration $\boldsymbol{x}^{(N+1)}$ from its policy $\pi(\boldsymbol{x}|\mathcal{D}_n)$, (2) allocation of the HP configuration $\boldsymbol{x}^{(N+1)}$ to a free worker, and (3) update of the policy after receiving available observations $\{(\boldsymbol{x}_i, \tilde{f}_i)\}_{i=n+1}^{N'}$. Most importantly, the objective function is the only component that has opportunities to interact with libraries after the initialization because the objective function has to take an HP configuration from the sampler, evaluate it, and return the output. As mentioned earlier, the optimization loop is not exposed to users for asynchronous optimization, so we need to rely on the internal process of the objective function to let each worker communicate with each other. To enable communication, we must be able to know which evaluation is processed by which worker out of P different workers and we need to understand the mechanisms of worker spawning functions in HPO libraries. Therefore, we explain two different user interfaces in the next two subsections.

2.4.2. Creating Workers in User Side

This type of parallel processing employs typically multiple main processes or allocates an instantiated objective function to each worker on the user side. The concrete examples of such parallel processing are `mpi4py`, file-based synchronization, and server-based synchronization. For example, `mpi4py` executes a Python script P times to run P parallel workers. Then each run, i.e. each worker, receives a unique worker ID and

[5] To the best of our knowledge, asynchronous optimization does not accept the ask-and-tell interface because users need to implement the asynchronous part in this case; see Def. 9 for the definition of the ask-and-tell interface.

they use the worker ID when they send data to or receive data from a specific worker. The basic mechanism is the same for file-based or server-based synchronization as well. They send or receive data via file system or a built server. Talking about major HPO libraries, NePS [6] uses file-based synchronization and BOHB (HPBandSter) [10] uses server-based synchronization. These libraries instantiate the provided objective function for each worker and keep them alive in each worker until the end of an optimization.

2.4.3. Creating Workers in Application Side

We first assume that there are only P unique thread IDs or process IDs, one of them is assigned to each evaluation, and none of the concurrent evaluations shares its ID with the other concurrent evaluations [7]. This type of parallel processing employs only one main process during a run and this main process creates child processes or child threads to distribute multiple jobs (one job for each child process or thread at most) to them in the run and each of them processes the assigned jobs concurrently. The concrete examples of such parallel processing are `multiprocessing`, `threading`, `dask`, and `concurrent` in Python. Talking about major HPO libraries, Optuna [15] uses `concurrent` and SMAC3 [14] uses `dask`. These parallel processing modules call the provided objective function every time an HP configuration is allocated to a worker and all the workers share the objective function. Hence, if we would like to let each worker communicate with each other, the objective function must somehow recognize which worker is calling the objective function for each evaluation. We discuss further details in the next chapter, but process ID or thread ID is used to achieve this in a nutshell.

[6] https://github.com/automl/neps/

[7] It is very common for parallel processing to have consistent thread IDs or process IDs until one run finishes. Without this assumption, it is harder to resolve the racing condition between workers.

3. Related Work

3.1. Hyperparameter Optimization Benchmarks and Simulation

Although there have been many HPO benchmarks invented for MFO such as HPOBench [4], NASLib [18], and JAHS-Bench-201 [6], none of them provides a module to allow researchers to simulate runtime internally. Other than HPO benchmarks, many HPO frameworks handling MFO have also been developed so far such as Optuna [15], SMAC3 [14], Dragonfly [19], and RayTune [20]. However, no framework above considers the simulation of runtime. Although HyperTune [21] and Syne-Tune [22] are internally simulating the runtime, we cannot simulate optimizers of interest if the optimizers are not introduced in the packages. It implies that researchers cannot immediately simulate their own new methods unless they incorporate their methods in these packages and limits experiments and fair comparisons. Furthermore, their simulation backend assumes that optimizers take the ask-and-tell interface (see Def. 9) and it requires the reimplementation of optimizers of interest in their codebase. Since reimplementation is time-consuming and does not guarantee its correctness without tests, it is helpful to have an easy-to-use Python wrapper for the simulation that does not require any reimplementation.

3.2. Multi-Fidelity Optimization Methods

MFO has been already used in various fields [23, 24, 25] and many papers have been published so far. The terminology "Multi-fidelity optimization (MFO)" was first brought by Kandasamy *et al.* [7] although some papers already made some effort to exploit lower-fidelity functions before [26, 27, 28, 29]. One of the most basic MFO methods is successive halving (SH) [8]. It creates a population of HP configurations and early-stops poor-performing HP configurations after using predetermined amounts

of a budget. A control parameter determines the amount of budgets at which each HP configuration is checked. HyperBand (HB) [9] removes the control parameter by trying several different budget choices in successive halving (SH). Due to the easy extensibility of HB, it has been combined with various methods such as differential evolution [11, 30] and Bayesian optimization [10, 31, 32, 14]. Not only these combinations, HB itself has also been modified to improve the performance [33, 34, 35, 36, 37, 38, 39]. For example, ASHA [33] promotes evaluations, which wait for the other ongoing evaluations, to the next fidelity without waiting for all the evaluations in the population to reduce the overhead caused by waiting times, PASHA [37] dynamically determines a budget choice based on rank consistency and HyperJump [38] skips evaluations of HP configurations that are likely to be early-stopped by HB. Along with HB and SH, the median stopping rule [40] is also practically used.

While HB and SH prune evaluations based on some rules in principle, some papers tackle this problem using machine learning algorithms. One of the approaches is learning curve prediction [29, 41, 42, 43]. Learning curve prediction tries to learn the time series of performance metrics over training epochs to judge if we should stop some evaluations early. While these papers train a machine learning algorithm along with a surrogate model for an optimization method, freeze-thaw Bayesian optimization [28] and DyHPO [44] train a surrogate model that takes both HP configuration $\boldsymbol{x}$ and a training epoch z. They once stop evaluations and restart pending evaluations if they judge it is worth evaluating with more training epochs. Another approach is cost-aware optimization, which considers the cost-effectiveness of evaluations with respect to information gain. This approach first trains a joint model $\hat{g}(\boldsymbol{x}, \boldsymbol{z})$ over HP configuration $\boldsymbol{x}$ and a fidelity vector $\boldsymbol{z}$. Then it either optimizes a cost-aware acquisition function, which is typically information gain per cost [27, 45, 46, 47, 48, 49, 50], or narrows down the next candidates so that the next evaluation surely reduces the uncertainty over the high-fidelity search space [7, 19, 51]. Although most methods train a joint model using multi-task Gaussian process [52], some methods use a linear combination of surrogate models on different fidelities [53, 32, 54]. The joint training in these papers relies on basic meta-learning methods in HPO [55, 56, 57].

While the aforementioned methods fix an HP configuration and progressively increase the fidelity, DAC [58, 59] dynamically changes an HP configuration $\boldsymbol{x}$ during training. For example, population-based training [60, 61, 62, 63, 64] dynamically changes neural architecture during training. To the best of our knowledge, population-based

training has not been supported by zero-cost benchmarks. Therefore, we do not support the problems considered in DAC in this thesis. On the other hand, our wrapper supports some other problem setups such as multi-objective optimization and constrained optimization as existing zero-cost benchmarks assume these setups. In fact, some multi-objective MFO methods already exist [65, 35, 30] and constrained optimization [66, 67, 68, 69, 70] could also be potentially extended to the MFO setup.

4. Asynchronous Optimization Wrapper for Zero-Cost Benchmarks

Before delving into the details, we will first define the following terminologies:

Definition 14 (Multi-Core Simulation (MCS))**.** *Given an HPO library that can run multiple workers concurrently, if our wrapper simulates an asynchronous optimization using multiple workers concurrently, we generally call such a simulation "multi-core simulation (MCS)".*

Definition 15 (Single-Core Simulation (SCS))**.** *Given an HPO library that has the ask-and-tell interface, if our wrapper simulates an asynchronous optimization for multiple workers using only a single worker, we call such a simulation "single-core simulation (SCS)".*

Although we discuss the details later, we do not have to use multiple workers to simulate an asynchronous optimization on a zero-cost benchmark if an HPO library has the ask-and-tell interface. In other words, SCS is a multithreading- and a multiprocessing-free simulation. Furthermore, we also use the following terminology:

Definition 16 (One-to-One Exchange)**.** *During each sampling, only one worker is free. More formally,* $\tilde{w}^{(N)} = 0$ *holds for an arbitrary* $N \in \mathbb{Z}_+ \setminus [P]$.

Note that this terminology stems from the fact that one result, but not multiple results, is exchanged with one HP configuration at each query.

Given the terminologies, we first describe the wrapper algorithms for MCS to answer to **RQ1** and show the validity of the algorithms. After the descriptions, we discuss the limitations of MCS and introduce SCS which is faster and free from multi-core-related issues to answer to **RQ3**. To validate the algorithms, we assume the following:

Assumption 1. *During an experiment, the overheads except for a sampling take* 0 *seconds.*

More specifically, we ignore the overheads caused by communication between workers and a query from zero-cost benchmarks. Although this is not practically correct, these overheads take milliseconds and they are negligible compared to an actual runtime $\tilde{\tau}$ and a sampling time $\tilde{t}$. In the next chapter, we empirically test our algorithm to guarantee the behavior of our algorithms. Furthermore, we assume the following:

Assumption 2. *Given a measure space* $(\mathbb{R}_+, \mathcal{B}_+, \mu)$ *where* $\mathcal{B}_+$ *is the Borel body over* $\mathbb{R}_+$ *and* $\mu : \mathcal{B}_+ \to \mathbb{R}_{\geq 0}$ *is the Lebegue measure, the probability measure of runtime* $\tilde{\tau}$ *is* 0 μ*-a.e. for an arbitrary HP configuration* $\boldsymbol{x} \in \mathcal{X}$*. More specifically,* $\mathbb{P}[\tilde{\tau} = c] = 0$ μ*-a.e. for an arbitrary choice of* $(\boldsymbol{x}, c) \in \mathcal{X} \times \mathbb{R}_+$*.*

Roughly speaking, this assumption implies that when we run a provided objective function with a specific HP configuration $\boldsymbol{x}$ multiple times, none of them exhibits exactly the same runtime.

4.1. Algorithm for Cheap Optimizer

The algorithm in this section relies on the formulation in Section 2.2.1 and this formulation assumes that a sampling time $\tilde{t}$ is almost zero. We first show the following theorem:

Corollary 1. *For an arbitrary pair* $(p, p') \in [P] \times [P]$ $(p \neq p')$ *and a positive integer* $N \in \mathbb{Z}_+ \setminus [P-1]$*,* $T_p^{(N)} \neq T_{p'}^{(N)}$ *holds almost surely.*

Corollary 1 states that there is only one free worker at the moment of each sampling once we finish the initialization. In other words, one-to-one exchange (see Def. 16) holds for cheap optimizers. Note that if $N < P$, some of $T_p^{(N)}$ are obviously zero.

Proof. First of all, if $P = 1$, the statement is obviously true because there exists no pair $(p, p') \in [1] \times [1]$ such that $p \neq p'$, so we check the statement for $P > 1$. To

prove the corollary, we focus on the simulated runtime of the p-th worker and the p'-th worker. For simplicity, we define the following for the proof:

$$\begin{aligned} \mathcal{I}_p^{(N)} &\coloneqq \{i_p^{(1)}, i_p^{(2)}, \dots\}, \text{ and} \\ \mathcal{I}_{p'}^{(N)} &\coloneqq \{i_{p'}^{(1)}, i_{p'}^{(2)}, \dots\} \end{aligned} \tag{4}$$

where $1 \leq i_p^{(1)} < i_p^{(2)} < \cdots \leq N$ and $1 \leq i_{p'}^{(1)} < i_{p'}^{(2)} < \cdots \leq N$. From Assumption 2, $\mathbb{P}[\tilde{\tau}^{(i_p^{(1)})} \neq \tilde{\tau}^{(i_{p'}^{(1)})}] = 0$ μ-$a.e.$ holds. Assume $\mathbb{P}[\sum_{j=1}^{k} \tilde{\tau}^{(i_p^{(j)})} \neq \tilde{\tau}^{(i_{p'}^{(1)})}] = 0$ μ-$a.e.$ holds for a positive integer k. Obviously, $\tilde{\tau}^{(i_{p'}^{(1)})} - \sum_{j=1}^{k} \tilde{\tau}^{(i_p^{(j)})}$ is a constant real number, and hence the same equation holds for $k+1$ as well. Due to the mathematical induction and Assumption 2, $\mathbb{P}[\sum_{j=1}^{k} \tilde{\tau}^{(i_p^{(j)})} \neq \tilde{\tau}^{(i_{p'}^{(1)})}] = 0$ μ-$a.e.$ holds for an arbitrary positive integer k.

Now, we see k as a fixed integer and assume $\mathbb{P}[\sum_{j=1}^{k} \tilde{\tau}^{(i_p^{(j)})} \neq \sum_{j=1}^{k'} \tilde{\tau}^{(i_{p'}^{(j)})}] = 0$ μ-$a.e.$ holds for a positive integer k'. In the same vein, $\sum_{j=1}^{k'} \tilde{\tau}^{(i_{p'}^{(j)})} - \sum_{j=1}^{k} \tilde{\tau}^{(i_p^{(j)})}$ is obviously a constant real number. Due to the mathematical induction and Assumption 2, $\mathbb{P}[\sum_{j=1}^{k} \tilde{\tau}^{(i_p^{(j)})} \neq \sum_{j=1}^{k'} \tilde{\tau}^{(i_{p'}^{(j)})}] = 0$ μ-$a.e.$ holds for an arbitrary positive integer k'. This completes the proof. □

Using Corollary 1, we can guarantee the following theorem:

Theorem 1. *Given P parallel workers and a positive integer $N \in \mathbb{Z}_+ \setminus [P]$, the size of a set of observations is almost surely $N - P$ when we sample the N-th sample.*

This theorem states that we can assume one-to-one exchange setup for cheap optimizers.

Proof. Due to the assumption of the sampling time $\tilde{t}^{(N)}$ being zero, the size of a set of observations is zero if $N \leq P$. When $N = P + 1$, at least one worker must be available, so the size of the set of observations is at least 1. Since the $(P+1)$-th sampling happens at $T_{\min}^{(P)} \coloneqq \min_{p \in [P]} T_p^{(P)}$, which is a constant value, the set size of $\operatorname{argmin}_{p \in [P]} T_p^{(P)}$ is almost surely 1 based on Corollary 1, so the size of the set of observations is almost surely 1. Assume the size of the set of observations is almost surely $k - P$ when we sample the k-th sample for $k > P$. Since the set size of $\operatorname{argmin}_{p \in [P]} T_p^{(k+1)}$ is almost surely 1 based on Corollary 1 and the size of the set

of observations is almost surely $k - P$, the size of the set of observations is almost surely $k - P + 1$ for $N = k + 1$. This completes the proof. □

Using Theorem 1, the p-th worker needs to:

1. wait till $p = \mathrm{argmin}_{p' \in [P]} T_{p'}^{(N)}$ satisfies,
2. receive $\boldsymbol{x}^{(N+1)}$ and fetch the corresponding result $(\tilde{f}^{(N+1)}, \tilde{\tau}^{(N+1)})$ from a zero-cost benchmark [1],
3. add $\tilde{\tau}^{(N+1)}$ to its own cumulative runtime $T_p^{(N)}$ and store $\boldsymbol{a}^{(N+1)}$ if it is the continual setup,
4. return the result $(\boldsymbol{x}^{(N+1)}, \tilde{f}^{(N+1)}, \tilde{\tau}^{(N+1)})$ to the optimizer once $p = \mathrm{argmin}_{p' \in [P]} T_{p'}^{(N')}$ is satisfied, and
5. replace N with N' and go back to 2.

Each worker needs to know $T_p^{(N)}$ from the other worker and available intermediate states from the past in this scenario, so this information must be shared somehow. If libraries spawn workers in the application side, we do not have access to the host module of child processes or child threads. Therefore, the information share must rely on either file system or a server. In our wrapper, we use file system because of its simplicity. Algorithm 1 shows the pseudocode of our wrapper in this scenario. Although our theory assumes that a sampling time $\tilde{t}^{(N)}$ is zero, this is not practically correct and we consider the sampling time. Under the assumption of the one-to-one exchange setup, the $(N+1)$-th waiting time is computed as follows:

$$\tilde{w}^{(N+1)} = \max\left\{0, \underbrace{\min_{p' \in [P]} T_{p'}^{(N-1)} + \tilde{w}^{(N)} + \tilde{t}^{(N)}}_{\text{The end of the latest sampling}} - \min_{p' \in [P]} T_{p'}^{(N)}\right\}. \tag{5}$$

If an optimizer is really cheap, $\tilde{w}^{(1)} = 0$ and $\tilde{t}^{(1)} = 0$ hold, so $\tilde{w}^{(N)}$ is always zero. Suppose $p = \mathrm{argmin}_{p' \in [P]} T_{p'}^{(N)}$ holds, then the simulated runtime for the p-th worker before the sampling will be:

$$T_p^{(N)} + \tilde{w}^{(N+1)} = \max\left\{T_p^{(N)}, \min_{p' \in [P]} T_{p'}^{(N-1)} + \tilde{w}^{(N)} + \tilde{t}^{(N)}\right\}. \tag{6}$$

[1] If it is an evaluation restart, we use a loaded intermediate state to calibrate the exact runtime.

Hence, the end of the next sampling will be:

$$\begin{aligned} T_{\text{now}}^{\text{new}} &\coloneqq T_p^{(N)} + \tilde{w}^{(N+1)} + \tilde{t}^{(N+1)} \\ &= \max\left\{ T_p^{(N)}, \min_{p' \in [P]} T_{p'}^{(N-1)} + \tilde{w}^{(N)} + \tilde{t}^{(N)} \right\} + \tilde{t}^{(N+1)} \\ &= \max\{T_p^{(N)}, T_{\text{now}}^{\text{old}}\} + \tilde{t}^{(N+1)}. \end{aligned} \tag{7}$$

Note that the following theorem upper-bounds the waiting time:

Theorem 2. *Suppose the sampling time is upper-bounded by $\tilde{t}_{\max}$, i.e. $\forall N \in \mathbb{Z}_+, \tilde{t}^{(N)} \leq \tilde{t}_{\max}$, then the waiting time satisfies $\tilde{w}^{(N)} \leq (P-1)\tilde{t}_{\max}$ for an arbitrary $N \in \mathbb{Z}_+$.*

Proof. In the worst case, a worker needs to wait for at most $P-1$ samplings after the worker was appended to the sampling waiting list because the sampling waiting list is first-in-first-out and the sampling waiting list could have the list length of at most P. Therefore, the waiting time is at most $(P-1)\tilde{t}_{\max}$. This completes the proof. □

This theorem states that if $\tilde{t}^{(N)}$ is really small, $\tilde{w}^{(N)}$ is also small, and one-to-one exchange is practically acceptable in terms of the simulated runtime reproduction. Furthermore, the following theorem holds:

Theorem 3. *If $\tilde{w}^{(n)} = 0$ for an arbitrary $n \in [N]$, the p-th worker does not have to know sampling times for the other workers to precisely compute $T_p^{(N)}$.*

Proof. From the assumption of $\tilde{w}^{(n)} = 0$ for an arbitrary n, the following is always true:

$$\min_{p' \in [P]} T_{p'}^{(n-1)} + \tilde{t}^{(n)} - \min_{p' \in [P]} T_{p'}^{(n)} \leq 0. \tag{8}$$

It implies that the update in Eq. (7) for the p-th worker requires only $T_p^{(n)}$ and $\tilde{t}^{(n+1)}$. Furthermore, the update from $T_p^{(n)}$ to $T_p^{(n+1)}$ only requires $T_p^{(n)}$, $\tilde{t}^{(n+1)}$, and $\tilde{\tau}^{(n+1)}$ if $n+1 \in \mathcal{I}_p^{(N)}$ and if $n+1 \notin \mathcal{I}_p^{(N)}$, $T_p^{(n+1)} = T_p^{(n)}$. Since the update of $T_p^{(n)}$ does not require $\tilde{t}^{(n)}$ for all $n \notin \mathcal{I}_p^{(N)}$, the p-th worker does not have to know sampling times for the other workers and this completes the proof. □

Algorithm 1 Automatic Waiting Time Scheduling Wrapper (see Figure 3 as well)

function WORKER($\boldsymbol{x}^{(N+1)}, \boldsymbol{a}^{(N+1)}$)
  Get intermediate state $s^{(N+1)} \coloneqq (\tilde{\tau}, T, \boldsymbol{a}) = \mathcal{S}.\texttt{get}(\boldsymbol{x}^{(N+1)}, (0, 0, \boldsymbol{a}^{(N+1)}))$
  if $s^{(N+1)}$ is invalid for $(\boldsymbol{x}^{(N+1)}, \boldsymbol{a}^{(N+1)})$ and $T_p^{(N)} + \tilde{t}^{(N+1)}$ **then**
    ▷ Condition 1: $s^{(\cdot)}$ must be $T \leq T_p^{(N)} + \tilde{t}^{(N+1)}$
    ▷ Condition 2: The new fidelity input has higher fidelity
    $s^{(N+1)} \leftarrow (0, 0, \boldsymbol{a}^{(N+1)})$
  Query the result $(\tilde{f}^{(N+1)}, \tilde{\tau}^{(N+1)})$
  Calibrate runtime $\tilde{\tau}^{(N+1)} \leftarrow \tilde{\tau}^{(N+1)} - \tilde{\tau}$
  ▷ T_{now} already takes the waiting time $\tilde{w}^{(N+1)}$ into account
  $T_{\text{now}} \leftarrow \max(T_{\text{now}}, T_p^{(N)}) + \tilde{t}^{(N+1)}$
  $T_p^{(N+1)} \leftarrow T_{\text{now}} + \tilde{\tau}^{(N+1)}, T_{p'}^{(N+1)} \leftarrow T_{p'}^{(N+1)}$ $(p' \neq p)$
  ▷ k is the number of results from the other workers appended during the wait
  ▷ $\tilde{t}^{(N+k+2)}$ is available only if $T_p^{(N+1)} \neq \min_{p' \in [P]} T_{p'}^{(N+k+1)}$
  Wait till $T_p^{(N+1)} \leq \min_{p' \in [P]} T_{p'}^{(N+k+1)} + \tilde{t}^{(N+k+2)}$ satisfies
  Record the intermediate state $\mathcal{S}[\boldsymbol{x}^{(N+1)}] = (\tilde{\tau}^{(N+1)}, T_p^{(N+1)}, \boldsymbol{a}^{(N+1)})$
  return $\tilde{f}^{(N+1)}$

π (an optimizer policy), `get_n_results` (a function that returns the number of recognized results by our wrapper. The results include the ones that have not been recognized by the optimizer yet.).
$\mathcal{D} \leftarrow \emptyset, T_p^{(0)} \leftarrow 0, T_{\text{now}} \leftarrow 0, \mathcal{S} \leftarrow \texttt{dict()}$
while the budget is left **do**
  ▷ This codeblock is run by P different workers in parallel
  $N \leftarrow \texttt{get\_n\_results()}$
  Get $\boldsymbol{x}^{(N+1)} \sim \pi(\cdot|\mathcal{D})$ and $\boldsymbol{a}^{(N+1)}$ with $\tilde{t}^{(N+1)}$ seconds
  $\tilde{f}^{(N+1)} \leftarrow \texttt{worker}(\boldsymbol{x}^{(N+1)}, \boldsymbol{a}^{(N+1)})$
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\boldsymbol{x}^{(N+1)}, \tilde{f}^{(N+1)})\}$

This theorem leads to an important benefit of the one-to-one exchange setup, which enables correct simulations even with a benchmark that causes a large query overhead because each worker can recognize overheads in its own process. Note that a query overhead must be negligible for an expensive optimizer as discussed in Theorem 4. However, if any one of $\tilde{w}^{(N)}$ takes a positive value, the size of the set of observations $\mathcal{D}$ at each query will not be correct and it affects the optimization behavior. Therefore, we discuss the implementation for expensive optimizers, i.e. $\tilde{t}^{(N)} \gg 0$ in the next section.

4.2. Algorithm for Expensive Optimizer

The algorithm in this section relies on the formulation in Section 2.2.2 and this formulation assumes that a sampling time $\tilde{t}$ is non-zero. As mentioned in Section 2.2.2, $\min_{p\in[P]} T_p^{(N)} + \tilde{t}^{(N+1)}$ could be larger than $T_{p'}^{(N)}$ even if $\min_{p\in[P]} T_p^{(N)} < T_{p'}^{(N)}$. Hence, the size of a set of observations at the $(N+1)$-th sampling becomes:

$$N - P + \sum_{p=1}^{P} \mathbb{I}\left[\min_{p'\in[P]} T_{p'}^{(N)} + \tilde{w}^{(N+1)} \geq T_p^{(N)}\right]. \tag{9}$$

Unlike the algorithm for cheap optimizers, the p-th worker may need to release its result without waiting for becoming $p = \operatorname{argmin}_{p'\in[P]} T_{p'}^{(N)}$ because $\min_{p'\in[P]} T_{p'}^{(N)} + \tilde{w}^{(N+1)} \geq T_p^{(N)}$ may hold while waiting for another sampling. It implies that we cannot assume the one-to-one exchange setup. As we release the p-th worker once $\min_{p'\in[P]} T_{p'}^{(N)} + \tilde{w}^{(N+1)} \geq T_p^{(N)}$ holds, each sampling will actually wait for the exact waiting time and we do not have to calculate the waiting time. Therefore, we can simply calculate the end of the sampling time by:

$$T_{\text{now}} = T_p^{(N)} + \underbrace{\tilde{w}^{(N+1)} + \tilde{t}^{(N+1)}}_{\text{Actual waiting time since the worker release}}. \tag{10}$$

Our wrapper in this scenario works identically as Algorithm 1 except Line 13 in Algorithm 1 is replaced with Algorithm 2. While the waiting time calculation becomes simpler, the waiting algorithm for each worker becomes slightly more complicated. Furthermore, this algorithm incurs some problems: (1) the racing condition that makes our wrapper incapable of handling a benchmark with a large query overhead and (2) more frequent access to the file system. Problem 1 leads to the following theorem:

Theorem 4. *Suppose we use Algorithm 2, precise simulation cannot be guaranteed if a benchmark query is non negligible.*

Proof. Suppose the number of parallel workers takes $P = 3$, a sampling time is always $\tilde{t}^{(n)} = 10$, and a query overhead $\mathcal{T}$ is 25. Assume we will fetch the k-th sample $\boldsymbol{x}^{(k)}$, the k-th runtime is $\tilde{\tau}^{(k)} = 3$, and all three workers finished their latest evaluations at the simulated runtime of T. In other words, $T_p^{(k-1)} = T$ for all $p \in [3]$.

1. The k-th sampling for Worker 1 starts at T,

2. The sampling for Worker 1 finishes at $T_{\text{now}} \leftarrow T + 10$,
3. The query for Worker 1 starts at T_{now},
4. The $(k+1)$-th sampling for Worker 2 starts at T_{now},
5. The sampling for Worker 2 finishes at $T_{\text{now}} \leftarrow T + 20$,
6. The $(k+2)$-th sampling for Worker 3 starts at T_{now}, and
7. The query for Worker 1 comes at $T + 25$.

However, Worker 1 should have received the k-th result at the simulated runtime of $T + 13$, and hence the k-th result must have been considered for the $(k+2)$-th sampling for Worker 3. Although we could avoid this contradiction if Algorithm 2 has the control to let each sampling wait until the latest query comes, the only control our wrapper has is the result release timing and sampling timing cannot be controlled by Algorithm 2. Therefore, this example is the counterexample of precise simulation and we cannot surely guarantee the correctness of simulations when we use a benchmark with a non-negligible query overhead. This completes the proof. □

This theorem is the reasoning behind Problem 1. More specifically, this problem comes from `get_n_results` and `get_p_now` in Algorithm 2 which are the functions that require communication between workers. Our wrapper first knows that sampling for a specific worker finishes, i.e. a change in p_{now}, and then receives a result, i.e. a change in N after a benchmark query. As the timer for sampling is still running, we need to register for the sampling finish immediately without waiting for the registration for the queried result. However, this algorithm causes a racing condition if a benchmark query overhead is large.

Problem 2 comes from the separated information registrations by the two functions. Each worker needs to check file system twice at each iteration in Algorithm 2 while Algorithm 1 checks only once at each iteration.

4.3. Limitations of Multi-Core Simulation

As discussed in Section 4.1 and Section 4.2, we provided two types of MCS wrapper algorithms:

Algorithm 2 Waiting Algorithm for Non One-to-One Exchange Setup

Δt (sleeping time till the next check.), `get_n_results` (a function that returns the number of recognized results by our wrapper. The results include the ones that have not been recognized by the optimizer yet.), `get_p_now` (a function that returns the index of the worker that will receive the latest sampling. In principle, the index of the free worker with the smallest $T_p^{(N)}$.).

1: ▷ t is used to measure sampling duration for the p-th worker at each iteration
2: ▷ $p_{\text{now}}^{\text{new}}$ may not be $\text{argmin}_{p' \in [P]} T_{p'}^{(N)}$ depending on benchmark query overhead
3: $N \leftarrow$ `get_n_results()`, $p_{\text{now}}^{\text{new}} \leftarrow$ `get_p_now()`, $t \leftarrow 0$
4: ▷ t_{now} is the current waiting time for the $p_{\text{now}}^{\text{new}}$-th worker at this moment
5: $T_{\text{now}} \leftarrow T_{p_{\text{now}}^{\text{new}}}^{(N)} + t_{\text{now}}$
6: **while** $T_p^{(N)} \neq T_{\text{min}}^{(N)} (:= \min_{p' \in [P]} T_{p'}^{(N)})$ **do**
7: ▷ $p_{\text{now}}^{\text{new}}$-th worker is waiting for the next sampling ($p \neq p_{\text{now}}^{\text{new}}$)
8: **if** $T_p^{(N)} \leq T_{p_{\text{now}}^{\text{new}}}^{(N)} + t$ **then**
9: **break**
10: $N_{\text{new}} \leftarrow$ `get_n_results()`
11: **if** $N = N_{\text{new}}$ and $p_{\text{now}}^{\text{old}} = p_{\text{now}}^{\text{new}}$ **then**
12: ▷ The latest sampling is in progress
13: $t \leftarrow t + \Delta t$
14: **else**
15: $T_{\text{now}} \leftarrow \max\{T_{\text{now}} + t, T_{p_{\text{now}}^{\text{new}}}^{(N_{\text{new}})}\}, N \leftarrow N_{\text{new}}, p_{\text{now}}^{\text{old}} \leftarrow p_{\text{now}}^{\text{new}}, t \leftarrow 0$
16: `time.sleep(`Δt`)`
17: **return** ▷ The p-th worker finishes the wait

- **Cheap Optimizer**: the algorithm with the assumption of one-to-one exchange, i.e. $\forall N \in \mathbb{Z}_+, \tilde{w}^{(N)} = 0$, shown in Algorithm 1, and

- **Expensive Optimizer**: Algorithm 1 which uses Algorithm 2 in Line 13.

While both algorithms support a direct usage of multi-core HPO and they only require minimal coding to use (see Appendix B for the actual usage), there are some drawbacks due to the parallel processing nature. The first algorithm is guaranteed to theoretically work perfectly and it works even with a large benchmark query overhead as long as a waiting time $\tilde{w}^{(N)}$ is always zero. However, some optimizers obviously cause a waiting time. The second algorithm mitigates the issue in the first algorithm, but a large query overhead causes the racing condition and simulations fail if we use such a benchmark. For example, we observed `dask.distributed.Client.scatter`, which allows us to keep a shared dataset in memory, causes a hang when using JAHS-Bench-201 [6] with 4 workers. In this case, users need to load surrogate models

at every query and it caused a 20-second overhead in our environment. Furthermore, there are some common issues that we did not discuss earlier:

- related to a large query overhead, individual query results for a tabular benchmark should be stored separately rather than stored as a big file [2] so that we can avoid incompatibility due to the large query and sampling overheads,
- these algorithms are not available for Windows OS as `fcntl` module, which is used for the file lock to prevent contamination, does not support Windows OS,
- simulation fails to reproduce the exact result if the initial sampling cost for P workers is more expensive than $\min_{p \in [P]} T_p^{(P)}$ [3],
- in general, a multi-core optimization is more likely to fail than a single-core optimization and MCS could suffer from HPO libraries-related vulnerabilities, e.g. scalability w.r.t. the number of workers P, and
- users must specify the temporary directory when using on a computer cluster to avoid slow down due to high I/O usage.

Our wrapper has another algorithm to address these issues in exchange for another constraint and we discuss the detail in the next section.

4.4. Algorithm for Ask-and-Tell Interface

As discussed in the previous section, while MCS directly allows users to apply existing HPO libraries, there are various issues. These problems can be addressed when we use SCS (see Def. 15). SCS enforces the ask-and-tell interface to HPO libraries, but as long as HPO libraries take the interface, we can simulate asynchronous optimization in a safer and more stable way. Additionally, HPO libraries do not even have to have a multi-core implementation to run simulations. For example, although HEBO [13] does not have its multi-core implementation, we could simulate its results as HEBO has the ask-and-tell interface. Algorithm 3 shows the algorithm for SCS. The key

[2] For example, if a tabular benchmark has $\{(\boldsymbol{x}_n, \tilde{f}_n, \tilde{\tau}_n)\}_{n=1}^{N_{\text{all}}}$, these results should not be stored as one big file, but rather stored as N_{all} files each with a result $\{(\tilde{f}_n, \tilde{\tau}_n)\}$. If we need to load the whole data at every query, the loading will cause a significant overhead, but it can be avoided if we store each result individually as we need to load only one small file at every query.

[3] To be fair, it does not make sense to use P workers in this case as this situation implies that we cannot fill out all the workers and it will be a waste of resources.

Algorithm 3 Single-Core Simulation (SCS) Using the Ask-and-Tell Interface

`opt` (an optimizer instance that has methods `opt.ask` and `opt.tell`.), `opt.ask` (a sampling method that returns an HP configuration $\boldsymbol{x}$ and optionally a set of arguments $\boldsymbol{a}$. We only show non-continual setup for simplicity.), `opt.tell` (a method to report a result $(\boldsymbol{x}, \tilde{f})$) to `opt`.

1: $T_{\text{now}} \leftarrow 0, \mathcal{W} \leftarrow [\texttt{None}] * P$
2: **for** $N = 1, 2, \ldots$ **do**
3: ▷ This codeblock is run by only 1 worker unlike Algorithm 1
4: $p \leftarrow \text{argmin}_{p' \in [P]} T_p^{(N)}$
5: $\boldsymbol{x}^{(N+1)} \leftarrow$`opt.ask()` ▷ The sampling time is $\tilde{t}^{(N+1)}$
6: Query $\tilde{f}^{(N+1)}, \tilde{\tau}^{(N+1)}$ using $\boldsymbol{x}^{(N+1)}$ and record to $\mathcal{W}$`[p]` $\leftarrow (\boldsymbol{x}^{(N+1)}, \tilde{f}^{(N+1)})$
7: ▷ Note that $\tilde{w}^{(N+1)} = \max\{0, T_{\text{now}} - T_p^{(N)}\}$
8: $T_{\text{now}} \leftarrow \max\{T_p^{(N)}, T_{\text{now}}\} + \tilde{t}^{(N+1)}, T_p^{(N+1)} \leftarrow T_{\text{now}} + \tilde{\tau}^{(N+1)}$
9: ▷ Skip if $\mathcal{W}$`[p]` is `None`
10: `opt.tell(`$\mathcal{W}$`[p])` and $\mathcal{W}$`[p]` $\leftarrow$ `None` for all p' that satisfies $T_{p'}^{(N+1)} \leq T_{\text{now}}$

point of the SCS is in Line 10, which the ask-and-tell interface makes possible. As Algorithm 3 allows us to schedule the return timing of each result to each worker only with a single core, experiments will be free from any parallel processing-related issues. The downsides of this algorithm are (1) it enforces HPO libraries to implement the ask-and-tell interface, (2) it may fail to reproduce the stochasticity caused by random seeds as discussed in Section 5.3, and (3) it may miss computational bottlenecks caused by high memory consumption due to expensive parallel evaluations of HP configurations. In Table 1, we summarize the advantages and disadvantages of each wrapper algorithm.

Table 1.: The advantages and disadvantages of each wrapper algorithm. Note that ◯ implies that it is an advantage and × implies that it is a disadvantage. "Actual Multi-Core Handling" means that we can directly run multi-core optimization natively supported in an HPO library.

	Algorithm 1		Algorithm 3
	One-to-One	Non One-to-One	
Runtime (File System Access)	×	×	◯
Query Overhead Handling	◯	×	◯
Sampling Overhead Handling	×	◯	◯
Random Seed Effect	◯	◯	×
Windows Support	×	×	◯
Scalability w.r.t. P	×	×	◯
Actual Multi-Core Handling	◯	◯	×
Actual Multi-Core Coding	×	×	◯
Any Optimizer Interface	◯	◯	×

5. Empirical Algorithm Validation on Test Cases

In this chapter, we discuss the validity of our algorithms discussed in Chapter 4 using several test cases to answer to **RQ2**. Throughout this chapter, we use the number of workers $P = 4$. We also note that our wrapper behavior depends only on returned runtime at each iteration in a non-continual setup and it is sufficient to consider only runtime $\tilde{\tau}^{(n)}$ and sampling time $\tilde{t}^{(n)} + \tilde{w}^{(n)}$ at each iteration. Therefore, we use a so-called fixed-configuration sampler, which defines a sequence of HP configurations and their corresponding runtimes at the beginning and samples from the fixed sequence iteratively. More formally, assume we would like to evaluate N HP configurations, then the sampler first generates $\{\tilde{\tau}^{(n)}\}_{n=1}^{N}$ and one of the free workers receives an HP configuration at the n-th sampling that leads to the runtime of $\tilde{\tau}^{(n)}$. Furthermore, we use two different strategies during sampling to simulate the sampling cost:

1. **Expensive Optimizer**: that intentionally sleeps for $c(|\mathcal{D}| + 1)$ seconds before giving $\tilde{\tau}^{(n)}$ to a worker where $|\mathcal{D}|$ is the size of a set of observations and $c \in \mathbb{R}_+$ is a proportionality constant, and
2. **Cheap Optimizer**: that gives $\tilde{\tau}^{(n)}$ to a worker immediately without sleeping.

In principle, the results of each test case are uniquely determined by a pair of an optimizer and a sequence of runtimes. Hence, we define such pairs at the beginning of each section. Note that we use the one-to-one exchange setup for "Cheap Optimizer" and non one-to-one exchange setup for "Expensive Optimizer".

5.1. Visual Verification on Small Handcrafted Test Cases

In this section, we will check our wrapper on small handcrafted cases. To test our wrapper, we use the following three test cases:

1. the cheap optimizer with $\{\tilde{\tau}^{(n)}\} = \{100, 40, 30, 20, 20, 30, 40, 20, 20, 30, 20, 40, 30, 20, 30, 20, 30, 40, 30, 10\}$,

2. the expensive optimizer with $\{\tilde{\tau}^{(n)}\} = \{40, 60, 60, 50, 50, 30, 30, 30\}$, and

3. the expensive optimizer with $\{\tilde{\tau}^{(n)}\} = \{50, 130, 80, 160, 130, 70, 20, 30\}$.

Note that the expensive optimizer uses $c = 10$. The first setup is the most basic setup and it should work as follows:

1. Worker 1 gets $\tilde{\tau}^{(1)} = 100$, Worker 2 gets $\tilde{\tau}^{(2)} = 40$, Worker 3 gets $\tilde{\tau}^{(3)} = 30$, and Worker 4 gets $\tilde{\tau}^{(4)} = 20$ after the initial sampling ($T_:^{(4)} = \{100, 40, 30, 20\}$),

2. Worker 4 returns its result and receives $\tilde{\tau}^{(5)} = 20$ ($T_:^{(5)} = \{100, 40, 30, 40\}$),

3. Worker 3 returns its result and receives $\tilde{\tau}^{(6)} = 30$ ($T_:^{(6)} = \{100, 40, 60, 40\}$),

4. Worker 2 returns its result and receives $\tilde{\tau}^{(7)} = 40$ ($T_:^{(7)} = \{100, 80, 60, 40\}$),

5. Worker 4 returns its result and receives $\tilde{\tau}^{(8)} = 20$ ($T_:^{(8)} = \{100, 80, 60, 60\}$), and

6. We repeat till the end and we expect to have the simulated minimum runtime of $\{20, 30, 40, 40, 60, 60, 80, 80, 90, 100, 100, 120, 120, 120, 130, 140, 150, 150, 160, 160\}$ at each iteration, which was accurately obtained in Figure 4 (**Left**).

The second setup requires our wrapper to wait for samplings for the other workers if necessary and it should work as follows:

1. Worker 1 gets $\tilde{\tau}^{(1)} = 40$ after $\tilde{t}^{(1)} = 10$, Worker 2 gets $\tilde{\tau}^{(2)} = 60$ after $\tilde{w}^{(2)} = 10$ and $\tilde{t}^{(2)} = 10$, Worker 3 gets $\tilde{\tau}^{(3)} = 60$ after $\tilde{w}^{(3)} = 20$ and $\tilde{t}^{(3)} = 10$, Worker 4 gets $\tilde{\tau}^{(4)} = 50$ after $\tilde{w}^{(4)} = 30$ and $\tilde{t}^{(4)} = 10$, and update to $T_{\text{now}} \leftarrow \tilde{w}^{(4)} + \tilde{t}^{(4)} = 40$ ($T_:^{(4)} = \{50, 80, 90, 90\}$),

2. Worker 1 returns its result ($|\mathcal{D}| = 1$), receives $\tilde{\tau}^{(5)} = 50$ after $\tilde{w}^{(5)} = 0$ and $\tilde{t}^{(5)} = 20$, and update to $T_{\text{now}} \leftarrow T_1^{(4)} + \tilde{t}^{(5)} = 70$ ($T_:^{(5)} = \{120, 80, 90, 90\}$),

3. Worker 2 returns its result ($|\mathcal{D}| = 2$), receives $\tilde{\tau}^{(6)} = 30$ after $\tilde{w}^{(6)} = 0$ ($\because T_2^{(5)} = 80 > T_{\text{now}}$) and $\tilde{t}^{(6)} = 30$, and update to $T_{\text{now}} \leftarrow T_2^{(5)} + \tilde{t}^{(6)} = 110$ ($T_:^{(6)} = \{120, 140, 90, 90\}$),

4. Worker 3 and Worker 4 return their results ($|\mathcal{D}| = 4$) and wait till the end of the latest sampling ($T_{\text{now}} = 110$),

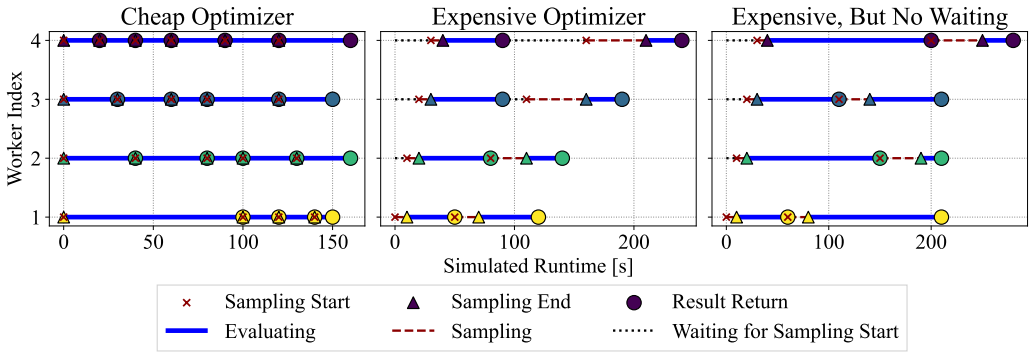


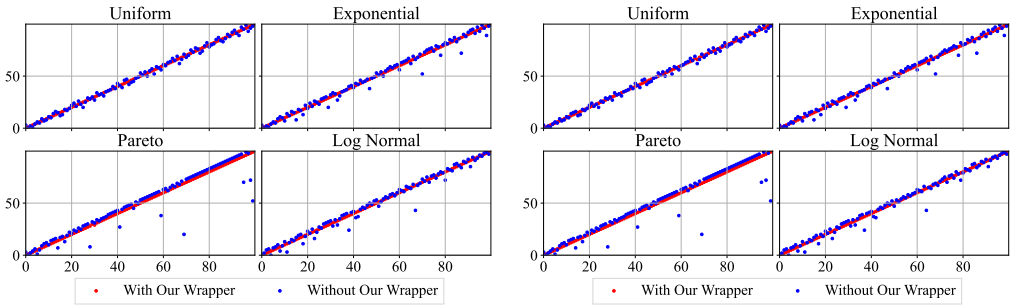
Figure 4.: The results of the experiments on small handcrafted cases obtained by our wrapper. The x-axis shows the simulated runtime of each experiment and the y-axis has four horizontal lines, which represent the timelines for each worker. We can see that our wrapper gives the expected timelines for each worker. **Left**: the test case with the cheap optimizer. As discussed in the text, the order of each HP configuration is correct. **Center**: the test case with the expensive optimizer and a sequence of HP configurations that causes waiting before some samplings. As expected, each sampling waits for the end of the previous sampling. **Right**: the test case with the expensive optimizer and a sequence of HP configurations that does not cause any waiting. As expected, each sampling starts immediately after each evaluation.

5. Worker 3 receives $\tilde{\tau}^{(7)} = 30$ at $T_{\text{now}} + \tilde{t}^{(7)} = 160$ where $\tilde{t}^{(7)} = 50$ because of $|\mathcal{D}| = 4$ and update to $T_{\text{now}} \leftarrow T_{\text{now}} + \tilde{t}^{(7)} = 160$ ($T_{:}^{(7)} = \{120, 140, 190, 90\}$),
6. Worker 1 and Worker 2 return their results during the sampling above ($|\mathcal{D}| = 6$),
7. Worker 4 receives $\tilde{\tau}^{(8)} = 30$ at $T_{\text{now}} + \tilde{t}^{(8)} = 230$ where $\tilde{t}^{(8)} = 70$ because of $|\mathcal{D}| = 6$ ($T_{:}^{(8)} = \{120, 140, 190, 260\}$).

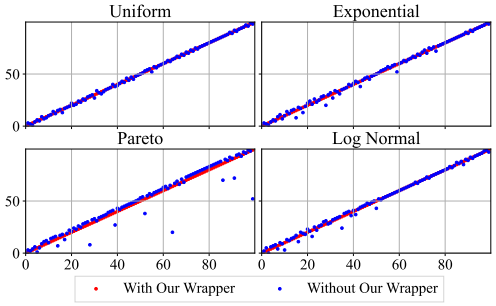
This result matches with Figure 4 (**Center**). Finally, the last setup also uses the expensive sampler while each worker fortunately does not have to wait for samplings for the other workers at all. This example works similarly to the other two setups and we obtained the expected result in Figure 4 (**Right**) by our wrapper.

5.2. Quantitative Verification on Random Test Cases

We tested our wrapper on small test cases and visually verified our algorithms in the previous section. In this section, we test our algorithms quantitatively and also



(a) Cheap optimizer **(b)** Expensive optimizer with $c = 5 \times 10^{-3}$



(c) Expensive optimizer with $c = 5 \times 10^{-2}$

Figure 5.: The return order verification results. When we use our wrapper, the red dots are obtained. If all the dots are aligned on $y = x$, it implies that the return order in a simulation with our wrapper and that in its naïve simulation perfectly match. As expected, the red dots completely overlap with $y = x$. See the text for the plot details.

empirically validate Theorem 1 and Theorem 3. For the tests, we generated test cases $\{\tilde{\tau}^{(n)}\}_{n=1}^{N}$ where $N = 100$ from the following distributions:

1. (**Uniform**): $\frac{T}{c} \sim \mathcal{U}(0, 2)$,
2. (**Exponential**): $\frac{T}{c} \sim \mathrm{Exp}(1)$,
3. (**Pareto**): $\frac{T+1}{c} \sim \mathcal{P}(\alpha = 1)$, and
4. (**LogNormal**): $\ln \frac{\sqrt{e}T}{c} \sim \mathcal{N}(0, 1)$,

where T is the probability variable of the runtime τ and we used $c = 5$. Each distribution uses the default setups of `numpy.random` and we calibrated each distribution except for the Pareto distribution so that the expectation becomes 5. Furthermore, we used (1) the cheap optimizer, (2) the expensive optimizer with $c = 5 \times 10^{-3}$, and (3) the expensive optimizer with $c = 5 \times 10^{-2}$. As $N = 100$, the worst sampling

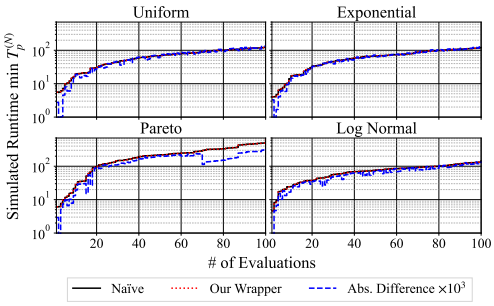
duration for Optimizers 2 and 3 will be 0.5 and 5 seconds. Since the expected runtime for each evaluation is 5 seconds, Optimizer 2 is a slightly expensive optimizer and Optimizer 3 is a very expensive optimizer. As we can expect a moderate amount of waiting time for Optimizer 2 and a lot of waiting time for Optimizer 3, it is more challenging to yield the precise return order and the precise simulated runtime for both setups. Hence, we would like to verify if our wrapper yields the precise return order and the precise simulated runtime using these setups.

First, we check return orders. We performed the following procedures to check whether the obtained return orders are correct:

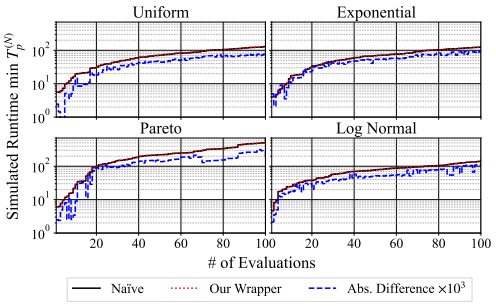
1. Run an optimization using naïve simulation (see Def. 13) and obtain $\{\tau_n^{\mathrm{NS}}\}_{n=1}^N$,
2. Run an optimization using our wrapper (MCS) and obtain $\{\tau_n^{\mathrm{MCS}}\}_{n=1}^N$,
3. Calculate the permutation $\{i_n\}_{n=1}^N$ of $[N]$ such that $\{\tau_n^{\mathrm{NS}}\}_{n=1}^N = \{\tau_{i_n}^{\mathrm{MCS}}\}_{n=1}^N$ holds, and
4. Plot $\{(n, i_n)\}_{n=1}^N$ (see Figure 5).

If the simulated return order is correct, the plot $\{(n, i_n)\}_{n=1}^N$ will look like $y = x$, i.e. (n, n) for all $n \in [N]$, and we expect to have such plots for all the experiments. Ultimately, our wrapper is unnecessary if we yield the exact order without our wrapper, so we also ran optimizations without our wrapper for the comparison.

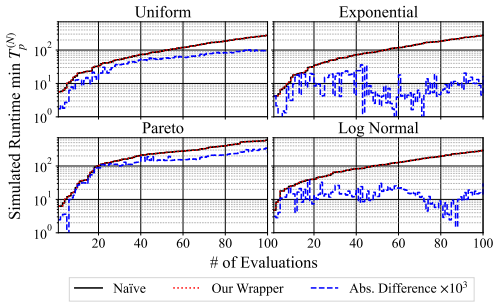
Figure 5 shows the results and the red dots represent the results using our wrapper. According to the figures, our wrapper successfully got $y = x$ for all the setups while we did not get $y = x$ without our wrapper. Although we can see the trend such that return orders become accurate at the late stage of an optimization except for the Pareto distribution even without our wrapper, our wrapper is essential to precisely simulate results. An interesting finding lies in the results for the Pareto distribution. As the Pareto distribution has a heavy tail, it often generates relatively large values, e.g. most samples exhibited around 5 seconds, but we could observe some samples had 500 seconds. Such samples locate far under the red lines in each figure for the Pareto distribution. Since optimizers without our wrapper will be immediately notified of the results for such samples, most blue dots locate slightly above the red lines and it completely confuses simulations. For example, HPOBench [4] also has this property and it is worth noting that our wrapper can overcome the edge case.



(a) Cheap optimizer



(b) Expensive optimizer with $c = 5 \times 10^{-2}$

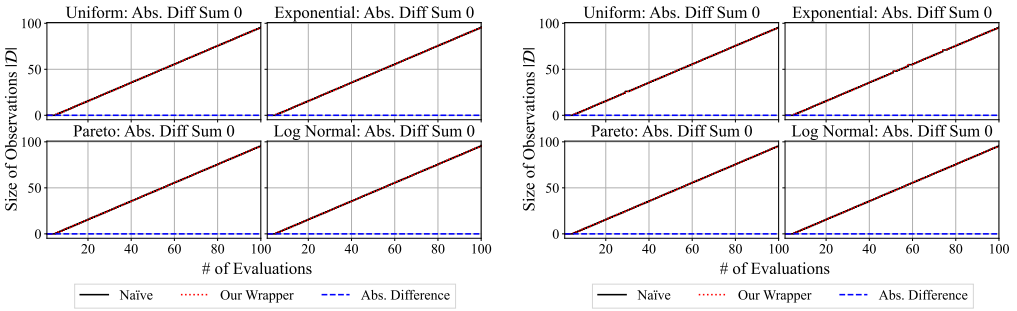


(c) Expensive optimizer with $c = 5 \times 10^{-3}$

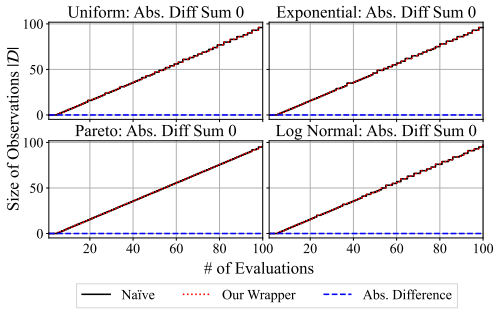
Figure 6.: The verification of the simulated runtime. The red dotted lines show the simulated runtime of our wrapper and the black solid lines show the actual runtime of the naïve simulation. The blue dotted lines show the absolute difference between the simulated runtime of our wrapper and the actual runtime of the naïve simulation multiplied by 1000. The red dotted lines and the black solid lines are expected to completely overlap and the blue lines should exhibit zero ideally.

Next, we check whether the simulated runtimes at each iteration were correctly calculated using the same setups. Figure 6 presents the simulated runtimes for each setup. As can be seen in the figures, our wrapper got a relative error of $1.0 \times 10^{-5} \sim 1.0 \times 10^{-3}$. Since the expectation of runtime is 5.0 seconds except for the Pareto distribution, the error was approximately $0.05 \sim 5.0$ milliseconds and this value comes from the communication overhead in our wrapper. This result empirically supports Theorem 3. Although the error is already sufficiently small, the relative error becomes much smaller when we use more expensive benchmarks that will give a large runtime $\tilde{\tau}^{(n)}$.

Finally, we check the size of the set of observations $|\mathcal{D}|$ at each iteration using the same setups. This experiment is also important because many optimizers such as Bayesian optimization use the set of observations $\mathcal{D}$ to train a surrogate model at



(a) Cheap optimizer **(b)** Expensive optimizer with $c = 5 \times 10^{-3}$



(c) Expensive optimizer with $c = 5 \times 10^{-2}$

Figure 7.: The size of the set of observations over time. The x-axis shows the number of HP config evaluations and the y-axis shows the size of the set observations $|\mathcal{D}|$. The red dotted lines are for the results obtained by our wrapper and the black solid lines are for the results obtained by the naïve simulation. The blue dashed lines show the absolute difference between these two values. We added the summation of the absolute difference in the titles of each subfigure and the value was zero for all the cases.

each iteration and a wrong set of observations may underestimate the performance of Bayesian optimization. Figure 7 presents the results for each setup. As can be seen, our wrapper got perfectly identical results to the naïve simulation and we could verify our implementation. Especially, the results for the cheap optimizer empirically support Theorem 1.

5.3. Performance Verification on Actual Runtime Reduction

In the previous sections, we verified the correctness of our algorithms and we could successfully validate our algorithms and their implementations. In this section, we

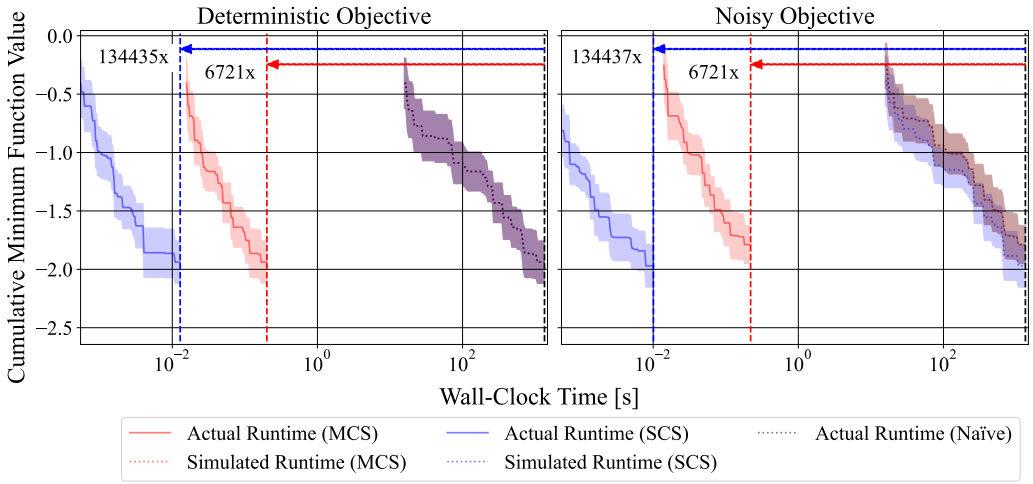


Figure 8.: The verification of actual runtime reduction. The x-axis shows the wall-clock time and the y-axis shows the cumulative minimum objective value during optimizations. Naïve simulation (black dotted line) serves the correct result and the simulated results (red/blue dotted lines) for each algorithm should ideally match the result of the naïve simulation. Actual runtime (red/blue solid lines) show the runtime reduction compared to the simulated results and it is better if we get the final result as quick as possible. **Left**: optimization of a deterministic multi-fidelity 6D Hartmann function. The simulated results for both MCS and SCS coincide with the correct result while both of them showed significant speedups. **Right**: optimization of a noisy multi-fidelity 6D Hartmann function. While the simulated result for MCS coincide with the correct result, SCS did not yield the same result. MCS could reproduce the result because MCS still uses the same parallel processing procedure and the only change is to wrap the objective function.

demonstrate the runtime reduction effect by our wrapper. To test the runtime reduction, we optimized multi-fidelity 6D Hartmann [1] [7] using random search with $P = 4$ workers over 10 different random seeds. In the noisy case, we added a random noise to the objective function. We used both MCS and SCS in this experiment and the naïve simulation. Figure 8 shows the result. In Figure 8 (**Left**), both MCS and SCS reproduced the results by the naïve simulation perfectly while they finished the experiments 6.7×10^3 times and 1.3×10^5 times quicker, respectively. Note that it is hard to see, but the right side of Figure 8 (**Left**) has the three lines: (1) Simulated Runtime (MCS), (2) Simulated Runtime (SCS), and (3) Actual Runtime

[1] We set the runtime function so that the maximum runtime for one evaluation becomes 1 hour. More precisely, we used $10 \times r(\boldsymbol{z})$ instead of $r(\boldsymbol{z})$ in Appendix A.2.

(Naïve), and they completely overlap with each other. SCS is much quicker than MCS because it does not require communication between each worker via the file system. Although MCS could reproduce the results by the naïve simulation even for the noisy case, SCS failed to reproduce the results because the naïve simulation relies on multi-core optimization while SCS does not use multi-core optimization and this difference affects the random seed effect to the optimizations. However, since SCS still reproduces the results for the deterministic case, it verifies our implementation of SCS. From the results, we can conclude that while SCS is generally quicker because it does not require communication via the file system, it may fail to reproduce the random seed effect due to the fact that it wraps an optimizer by relying on the ask-and-tell interface instead of using the multi-core implementation provided by the optimizer.

6. Real-World Experiments Using Zero-Cost Benchmarks

In this chapter, we perform MFO using various major HPO libraries. First, we show the average rank performance of each method over time and answer to **RQ4** using a statistical test. Last but not least, while our experiments would have taken 5.8×10^{10} seconds without our wrapper, our experiments took 4.3×10^{7} seconds in total; see Table 2 and Appendix C.2 for more details. Note that our wrapper usage is described in Appendix B.

6.1. Experiment Setup

In the experiments, we used the following benchmark problems:

1. the multi-fidelity version of Branin, 3D Hartmann, 6D Hartmann functions [19] with the runtime function of $3600 \times r(\boldsymbol{z})$,
2. MLP benchmarks from HPOBench [4],
3. HPOlib [16],
4. JAHS-Bench-201 [6], and
5. LCBench [71] from YAHPOBench [72].

Note that the details are available in Appendix A. For optimizers, we used the following:

1. random search [73] in Optuna [15] with 10 times more budgets,
2. (**MFO**) HyperBand [9] in HpBandSter [10] with 10 times more budgets,
3. tree-structured Parzen estimator (TPE) [74, 75] in Optuna,

Table 2.: Actual and simulated runtimes for total experiment runtimes. **Act.** is the total actual runtime and **Sim.** is the total simulated runtime. × **Fast** shows how many times the total actual runtime was faster.

$P = 1$			$P = 2$			$P = 4$			$P = 8$		
Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
9.2e+06/	3.0e+10/	3.3e+03	1.1e+07/	1.5e+10/	1.5e+03	1.1e+07/	7.7e+09/	6.9e+02	1.2e+07/	3.9e+09/	3.2e+02

4. (**MFO**) BOHB (a combination of TPE and HyperBand) in HpBandSter [10],
5. HEBO (a state-of-the-art Bayesian optimization method) [13],
6. (**MFO**) DEHB (a combination of differential evolution and HyperBand) [11],
7. (**MFO**) NePS (a variant of HyperBand by default) [1], and
8. (**MFO**) SMAC3 (a combination of random forest-based Bayesian optimization and HyperBand) [14].

We marked MFO methods with "(**MFO**)" above.

Each optimizer was run on each benchmark problem over 30 different random seeds with different numbers of workers $P \in \{1, 2, 4, 8\}$. Since each MFO method uses HyperBand internally, we consistently used $\eta = 3$, i.e. the default value of a control parameter of HyperBand that determines the proportion of HP configurations discarded in each round of successive halving [8]. The budget for each optimization was fixed to 200 full evaluations and this leads to 450 function calls for HyperBand-based methods with $\eta = 3$. Each optimizer except for HEBO was run using MCS, but HEBO was run using SCS as HEBO does not support multi-core optimization, but it supports the ask-and-tell interface. Note that SMAC3 could not be run on LCBench and JAHS-Bench-201 due to a dependency issue. All the computations for this experiment were performed on bwForCluster NEMO, which has 20 cores of Intel(R) Xeon(R) CPU E5-2630 v4 on each computational node, and we used 1 core of the CPU and 15GB RAM per worker. For HEBO, we used 1 core of the CPU with 32GB RAM for all the setups. As seen in Table 2, the whole experiment could be finished 1.3×10^3 times shorter runtime. Note that since the simulated runtime obviously becomes quicker given the same amount of budget as the number of workers

[1] https://github.com/automl/neps/

P increases, $\times$ Fast apparently worsens, but the total actual runtime became only slightly slower (about 1.3 times) as the number of workers P increases.

We used average rank for the visualization of the performance, so we describe the calculation method below:

1. Collect the results $\{\{\{\{(\boldsymbol{x}_{k,l,s,n}, \tilde{f}_{k,l,s,n}, T_{k,l,s,n})\}_{n=1}^{N_l}\}_{s=1}^{30}\}_{l=1}^{L}\}_{k=1}^{K}$ where K is the number of setups, L is the number of optimizers, s is the seed specifier, N_l is the number of function calls for the l-th optimizer, and $T_{k,l,s,n}$ is the simulated runtime of the k-th setup using the l-th optimizer with the s-th seed up to the n-th observations,

2. For each $l \in [L]$ and $k \in [K]$, compute $T_{k,l}^{\min} = \min_{s\in[30]} T_{k,l,s,1}$ and $T_{k,l}^{\max} = \max_{s\in[30]} T_{k,l,s,N_l}$ and log-scale time step grids $\{t_{k,l,i}\}_{i=1}^{100}$ where $t_{k,l,i} \coloneqq \exp[\log T_{k,l}^{\min} + \frac{i-1}{99}(\log T_{k,l}^{\max} - \log T_{k,l}^{\min})]$,

3. For each $i \in [100]$ and $s \in [30]$, compute n_i such that $T_{k,l,s,n_i} \leq t_{k,l,i} < T_{k,l,s,n_i+1}$ and gather as $\{(\bar{f}_{k,l,i}, t_{k,l,i})\}_{i=1}^{100} \coloneqq \{(\mathrm{med}_{s\in[30]}\ \tilde{f}_{k,l,s,n_i}, t_{k,l,i})\}_{i=1}^{100}$ where med calculates the median of a given set,

4. For each $k \in [K]$, compute $T_k^{\max} = \max_{l\in[L]} T_{k,l}^{\max}$, $T_k^{\min} = T_k^{\max} \times 10^{-5}$, and log-scale time step grids $\{t_{k,i}\}_{i=1}^{200}$ where $t_{k,i} \coloneqq \exp[\log T_k^{\min} + \frac{i-1}{199}(\log T_k^{\max} - \log T_k^{\min})]$,

5. For each $i \in [200]$ and $l \in [L]$, compute n_i such that $t_{k,l,n_i} \leq t_{k,i} < t_{k,l,n_i+1}$ and gather as $\{r_{k,l,i}\}_{i=1}^{200} \coloneqq \{\mathrm{rank}(\bar{f}_{k,l,n_i}, \{\bar{f}_{k,l',n_i}\}_{l'=1}^{L})\}_{i=1}^{200}$ where $\mathrm{rank}(a, \mathcal{S})$ calculates the ranking of $a \in \mathcal{S}$ in the set $\mathcal{S}$, and

6. For each $l \in [L]$, compute the average rank $\{\bar{r}_{l,i}\}_{i=1}^{200} \coloneqq \{\mathrm{mean}_{k\in[K]} r_{k,l,i}\}_{i=1}^{200}$.

Then we plot the trajectories $\{(b_i, \bar{r}_{l,i})\}_{i=1}^{200}$ for each optimizer $l \in [L]$ where $b_i \coloneqq \exp[\log 10^{-5} + \frac{i-1}{199}(\log 10^{0} - \log 10^{-5})] = \frac{200-i}{199} \log 10^{-5}$ is the budget fraction used by an optimizer. Note that the module is publicly available [2]. Furthermore, we performed the Friedman rank test for each $P \in \{1, 2, 4, 8\}$ and compared the rankings over each number of workers to answer to **RQ4**.

[2] See `get_average_rank` in `http://github.com/nabenabe0928/mfhpo-simulator`.

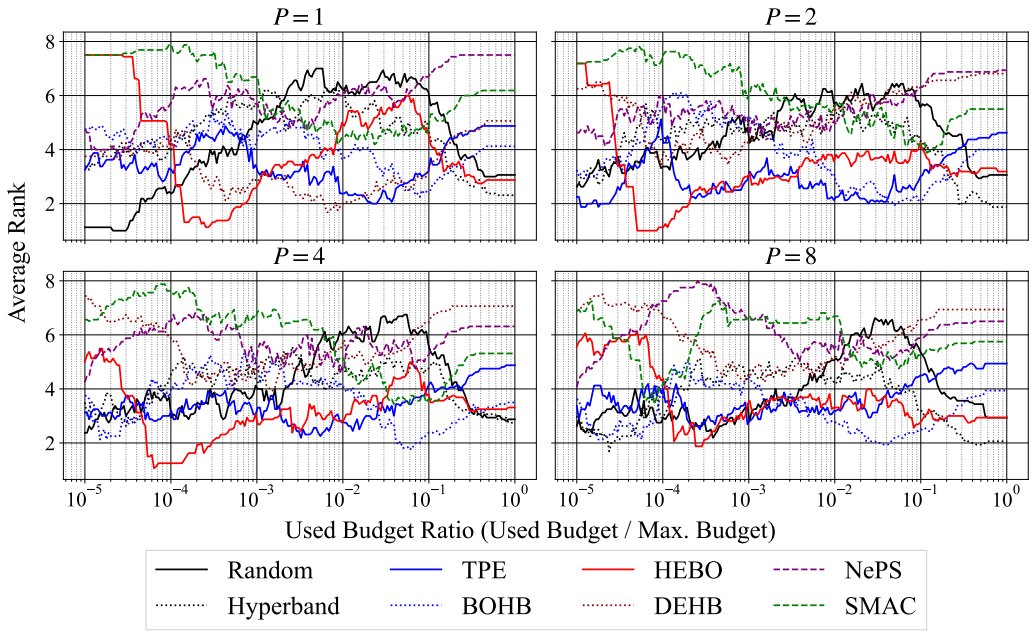


Figure 9.: The average rank on HPOBench.

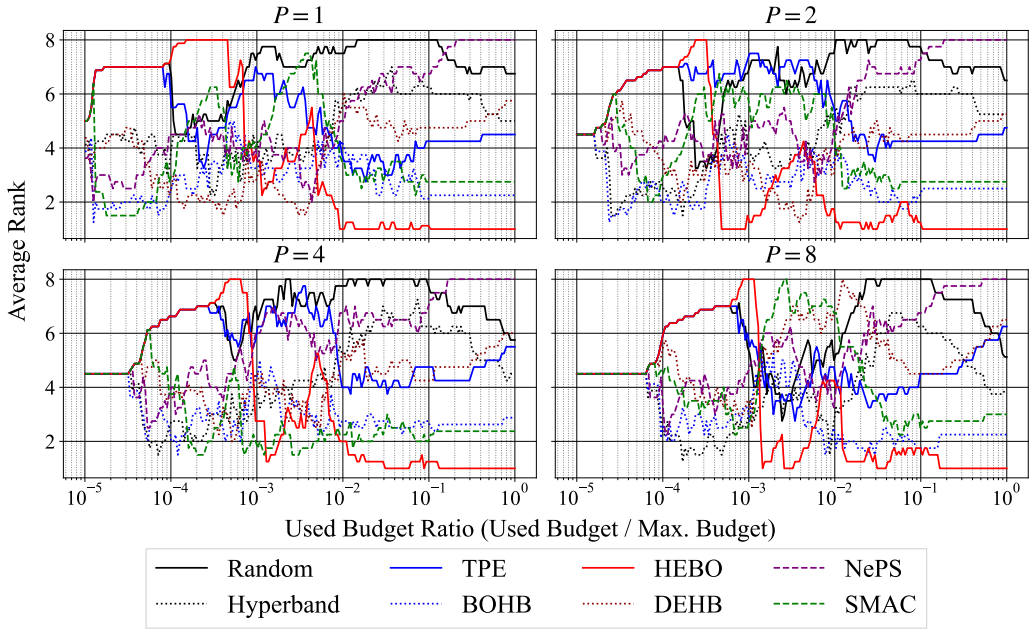


Figure 10.: The average rank on HPOlib.

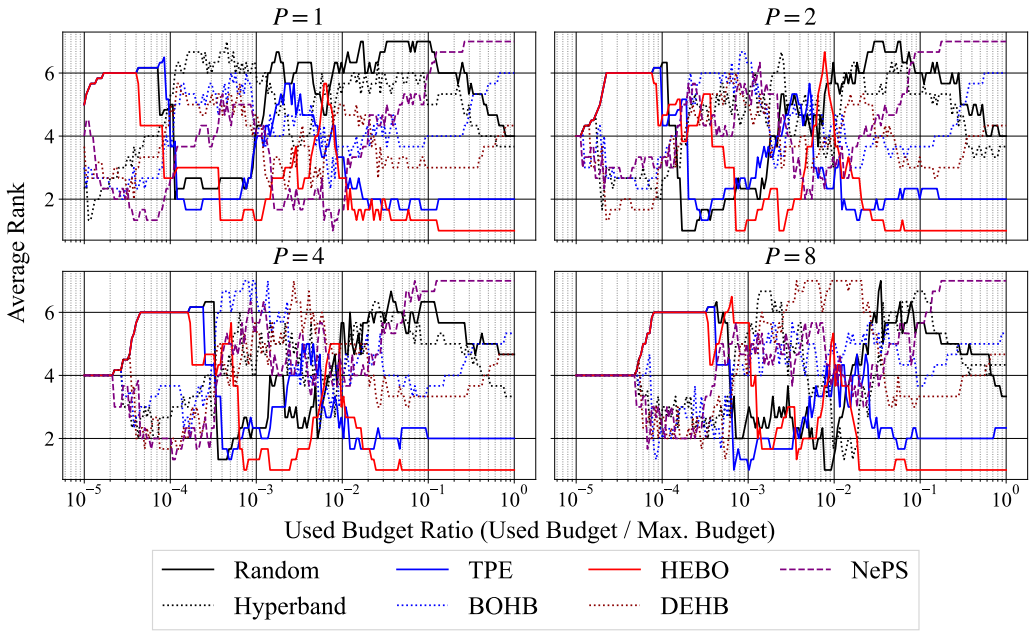


Figure 11.: The average rank on JAHS-Bench-201.

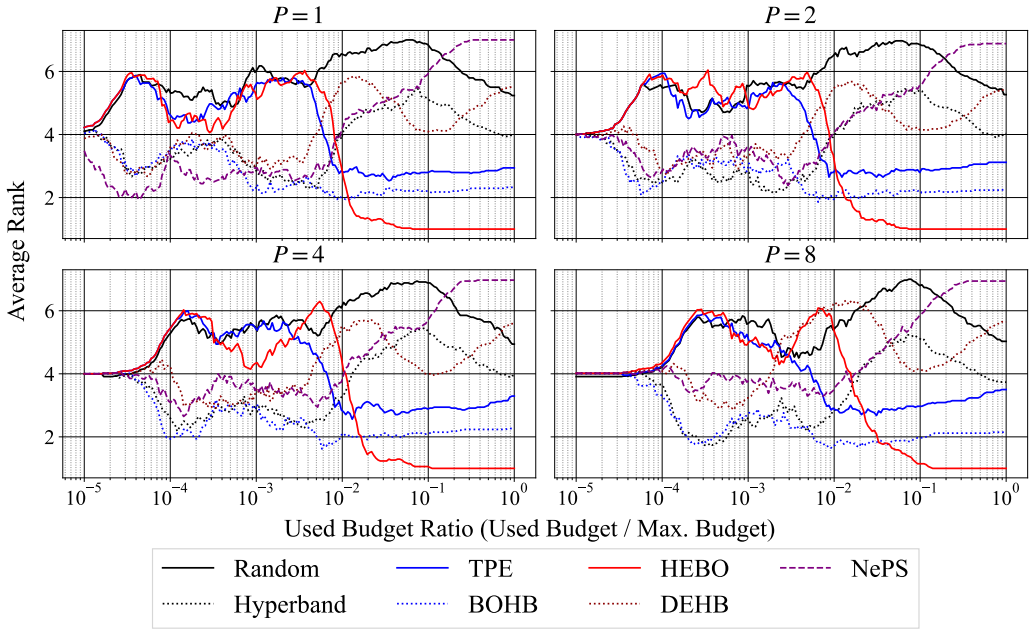


Figure 12.: The average rank on LCBench.

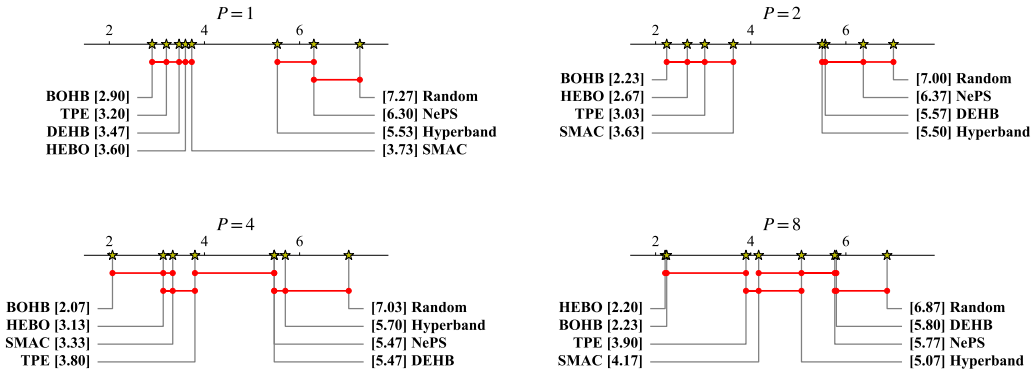


Figure 13.: The critical difference diagrams with $1/2^4$ of the runtime budget for random search. "[x.xx]" shows the average rank of each optimizer after using $1/2^4$ of the runtime budget for random search. For example, "BOHB [2.90]" means that BOHB achieved the average rank of 2.90 among all the optimizers after running the specified amount of budget. The title of each figure shows the number of workers used in the visualization and the red bars connect all the optimizers that show no significant performance difference. Note that we used all the results except for JAHS-Bench-201 and LCBench due to the incompatibility between SMAC and JAHS-Bench-201 and LCBench.

6.2. Results

Figures 9–12 present the average rank performance curves for each optimizer on each benchmark. As the average rank plots lose the information about the scale in the objective function value, we provided the objective function value over time in Appendix C.1. Although rough trajectories of average rank plots do not change largely for some cases, ranks between some optimizers change depending on the number of workers. To statistically test this and answer to **RQ4**, we performed the Friedman test and its post-hoc analysis, i.e. critical difference diagram. The result is shown in Figure 13. To visualize the figure, we used the results of each optimizer after using $1/2^4$ of the runtime budget for random search. Note that random search uses ten times more budget compared to the other methods except for HyperBand, so $1/2^4$ of the maximum runtime budget $T_k^{\max}$ implies that all the optimizers including random search and HyperBand consumed the same amount of runtime. The critical difference diagrams for different budget sizes are available in Appendix C.3. According to Figure 13, while some optimizer pairs such as BOHB and HEBO, and random search and NePS show the same performance statistically

over the four different numbers of workers $P \in \{1, 2, 4, 8\}$, DEHB exhibited different performance significance depending on the number of workers. For example, DEHB belongs to the top group with BOHB, TPE, and HEBO for $P = 1$, but it belongs to the bottom group with random search and NePS for $P = 8$. As shown by the red bars, we see statistically significant performance differences between the top groups and the bottom groups. Therefore, this directly answers to **RQ4** and we need to consider the effect caused by the number of workers P for practical application.

7. Conclusions

In this thesis, we presented the algorithms to reduce experiment runtimes of asynchronous MFO and answered the four research questions. In Chapter 4, we provided the algorithms to sort out the return order and correctly calculate the simulated runtime at each iteration to answer to **RQ1**. Although we showed that these algorithms work theoretically, the implementation will still suffer from the vulnerabilities in multi-core processing, e.g. racing conditions and latency caused by communication between workers. To address this problem, we also provided the algorithm to simulate identically only with a single core and this answered to **RQ3**. In Chapter 5, we empirically validated our algorithms and implementation using various runtime distributions and various optimizers with different sampling costs. The results exhibited that our implementation correctly yields the correct return orders and this answered to **RQ3**. Additionally, the simulated runtimes match the actual runtimes with a relative error of 0.1% while finishing the simulations 6.7×10^3 times quicker for MCS and 1.3×10^5 times quicker for SCS. Finally, in Chapter 6, we performed experiments using major HPO libraries for parallel setups. The whole experiments were over 1.3×10^3 times quicker than the naïve simulations would have taken. The statistical test on the data obtained by the experiments showed that experiments only with non-parallel setup $P = 1$ are insufficient to demonstrate the performance of HPO methods on parallel setups and this answered to **RQ4**. Our wrapper can be installed via `pip install mfhpo-simulator` and it will positively impact the MFO research using parallel setups.

Bibliography

[1] Y. Chen, A. Huang, Z. Wang, I. Antonoglou, J. Schrittwieser, D. Silver, and N. de Freitas, "Bayesian optimization in AlphaGo," *arXiv:1812.06855*, 2018.

[2] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *AAAI Conference on Artificial Intelligence*, 2018.

[3] K. Eggensperger, F. Hutter, H. Hoos, and K. Leyton-Brown, "Efficient benchmarking of hyperparameter optimizers via surrogates," in *AAAI Conference on Artificial Intelligence*, 2015.

[4] K. Eggensperger, P. Müller, N. Mallik, M. Feurer, R. Sass, A. Klein, N. Awad, M. Lindauer, and F. Hutter, "HPOBench: A collection of reproducible multi-fidelity benchmark problems for HPO," *arXiv:2109.06716*, 2021.

[5] S. Arango, H. Jomaa, M. Wistuba, and J. Grabocka, "HPO-B: A large-scale reproducible benchmark for black-box HPO based on OpenML," *arXiv:2106.06257*, 2021.

[6] A. Bansal, D. Stoll, M. Janowski, A. Zela, and F. Hutter, "JAHS-Bench-201: A foundation for research on joint architecture and hyperparameter search," in *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*, 2022.

[7] K. Kandasamy, G. Dasarathy, J. Schneider, and B. Póczos, "Multi-fidelity Bayesian optimisation with continuous approximations," in *International Conference on Machine Learning*, 2017.

[8] K. Jamieson and A. Talwalkar, "Non-stochastic best arm identification and hyperparameter optimization," in *International Conference on Artificial Intelligence and Statistics*, 2016.

[9] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "HyperBand: A novel bandit-based approach to hyperparameter optimization," *Journal of Machine Learning Research*, vol. 18, 2017.

[10] S. Falkner, A. Klein, and F. Hutter, "BOHB: Robust and efficient hyperparameter optimization at scale," in *International Conference on Machine Learning*, 2018.

[11] N. Awad, N. Mallik, and F. Hutter, "DEHB: Evolutionary HyperBand for scalable, robust and efficient hyperparameter optimization," *arXiv:2105.09821*, 2021.

[12] J. Snoek, H. Larochelle, and R. Adams, "Practical Bayesian optimization of machine learning algorithms," *Advances in Neural Information Processing Systems*, 2012.

[13] A. Cowen-Rivers, W. Lyu, R. Tutunov, Z. Wang, A. Grosnit, R. Griffiths, A. Maraval, H. Jianye, J. Wang, J. Peters, *et al.*, "HEBO: pushing the limits of sample-efficient hyper-parameter optimisation," *Journal of Artificial Intelligence Research*, vol. 74, 2022.

[14] M. Lindauer, K. Eggensperger, M. Feurer, A. Biedenkapp, D. Deng, C. Benjamins, T. Ruhkopf, R. Sass, and F. Hutter, "SMAC3: A versatile Bayesian optimization package for hyperparameter optimization," *Journal of Machine Learning Research*, vol. 23, 2022.

[15] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *International Conference on Knowledge Discovery & Data Mining*, 2019.

[16] A. Klein and F. Hutter, "Tabular benchmarks for joint architecture and hyperparameter optimization," *arXiv:1905.04970*, 2019.

[17] X. Dong and Y. Yang, "NAS-Bench-201: Extending the scope of reproducible neural architecture search," *arXiv:2001.00326*, 2020.

[18] Y. Mehta, C. White, A. Zela, A. Krishnakumar, G. Zabergja, S. Moradian, M. Safari, K. Yu, and F. Hutter, "NAS-Bench-Suite: NAS evaluation is (now) surprisingly easy," *arXiv:2201.13396*, 2022.

[19] K. Kandasamy, K. Vysyaraju, W. Neiswanger, B. Paria, C. Collins, J. Schneider, B. Poczos, and E. Xing, "Tuning hyperparameters without grad students: Scalable

and robust Bayesian optimisation with Dragonfly," *Journal of Machine Learning Research*, vol. 21, 2020.

[20] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training," *arXiv:1807.05118*, 2018.

[21] Y. Li, Y. Shen, H. Jiang, W. Zhang, J. Li, J. Liu, C. Zhang, and B. Cui, "Hyper-Tune: towards efficient hyper-parameter tuning at scale," *arXiv:2201.06834*, 2022.

[22] D. Salinas, M. Seeger, A. Klein, V. Perrone, M. Wistuba, and C. Archambeau, "Syne Tune: A library for large scale hyperparameter tuning and reproducible research," in *International Conference on Automated Machine Learning*, 2022.

[23] J. Song, Y. Tokpanov, Y. Chen, D. Fleischman, K. Fountaine, H. Atwater, and Y. Yue, "Optimizing photonic nanostructures via multi-fidelity Gaussian processes," *arXiv:1811.07707*, 2018.

[24] A. Palizhati, M. Aykol, S. Suram, J. Hummelshøj, and J. Montoya, "Multi-fidelity sequential learning for accelerated materials discovery," 2021.

[25] P. Thodoroff, M. Kaiser, R. Williams, R. Arthern, S. Hosking, N. Lawrence, J. Byrne, and I. Kazlauskaite, "Multi-fidelity experimental design for ice-sheet simulation," *arXiv:2307.08449*, 2023.

[26] F. Hutter, H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Learning and Intelligent Optimization*, 2011.

[27] K. Swersky, J. Snoek, and R. Adams, "Multi-task Bayesian optimization," *Advances in Neural Information Processing Systems*, 2013.

[28] K. Swersky, J. Snoek, and R. Adams, "Freeze-thaw Bayesian optimization," *arXiv:1406.3896*, 2014.

[29] T. Domhan, J. Springenberg, and F. Hutter, "Speeding up automatic hyper-parameter optimization of deep neural networks by extrapolation of learning curves," in *International Joint Conference on Artificial Intelligence*, 2015.

[30] N. Awad, A. Sharma, and F. Hutter, "MO-DEHB: Evolutionary-based HyperBand for multi-objective optimization," *arXiv:2305.04502*, 2023.

[31] A. Klein, L. Tiao, T. Lienart, C. Archambeau, and M. Seeger, "Model-based asynchronous hyperparameter and neural architecture search," *arXiv:2003.10865*, 2020.

[32] B. Yu, H. Shen, P. Huai, Q. Xu, and W. He, "FedTLBOHB: Efficient HyperBand with transfer learning for vertical federated learning," in *International Conference on Computer and Communications*, 2022.

[33] L. Li, K. Jamieson, A. Rostamizadeh, E. Gonina, J. Ben-Tzur, M. Hardt, B. Recht, and A. Talwalkar, "A system for massively parallel hyperparameter tuning," *Proceedings of Machine Learning and Systems*, vol. 2, 2020.

[34] G. Zhu and R. Zhu, "Accelerating hyperparameter optimization of deep neural network via progressive multi-fidelity evaluation," in *Advances in Knowledge Discovery and Data Mining*, 2020.

[35] R. Schmucker, M. Donini, M. Zafar, D. Salinas, and C. Archambeau, "Multi-objective asynchronous successive halving," *arXiv:2106.12639*, 2021.

[36] G. Zappella, D. Salinas, and C. Archambeau, "A resource-efficient method for repeated HPO and NAS problems," *arXiv:2103.16111*, 2021.

[37] O. Bohdal, L. Balles, B. Ermis, C. Archambeau, and G. Zappella, "PASHA: Efficient HPO with progressive resource allocation," *arXiv:2207.06940*, 2022.

[38] P. Mendes, M. Casimiro, P. Romano, and D. Garlan, "HyperJump: accelerating HyperBand via risk modelling," in *AAAI Conference on Artificial Intelligence*, vol. 37, 2023.

[39] H. Lee, G. Lee, J. Kim, S. Cho, D. Kim, and D. Yoo, "Improving multi-fidelity optimization with a recurring learning rate for hyperparameter tuning," in *Winter Conference on Applications of Computer Vision*, 2023.

[40] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google Vizier: A service for black-box optimization," in *International Conference on Knowledge Discovery & Data Mining*, 2017.

[41] A. Klein, S. Falkner, J. Springenberg, and F. Hutter, "Learning curve prediction with Bayesian neural networks," 2016.

[42] A. Chandrashekaran and I. Lane, "Speeding up hyper-parameter optimization by extrapolation of learning curves using previous builds," in *European Conference on Machine Learning and Knowledge Discovery in Databases*, 2017.

[43] S. Adriaensen, H. Rakotoarison, S. Müller, and F. Hutter, "Efficient Bayesian learning curve extrapolation using prior-data fitted networks," in *Meta-Learning Workshop at Advances in Neural Information Processing Systems*, 2022.

[44] M. Wistuba, A. Kadra, and J. Grabocka, "Supervising the multi-fidelity race of hyperparameter configurations," *Advances in Neural Information Processing Systems*, 2022.

[45] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter, "Fast Bayesian optimization of machine learning hyperparameters on large datasets," in *International Conference on Artificial Intelligence and Statistics*, 2017.

[46] M. Poloczek, J. Wang, and P. Frazier, "Multi-information source optimization," *Advances in Neural Information Processing Systems*, 2017.

[47] S. Takeno, H. Fukuoka, Y. Tsukada, T. Koyama, M. Shiga, I. Takeuchi, and N. Karasuyama, "Multi-fidelity Bayesian optimization with max-value entropy search and its parallelization," in *International Conference on Machine Learning*, 2020.

[48] J. Wu, S. Toscano-Palmerin, P. Frazier, and A. Wilson, "Practical multi-fidelity Bayesian optimization for hyperparameter tuning," in *Uncertainty in Artificial Intelligence*, 2020.

[49] S. Li, R. Kirby, and S. Zhe, "Batch multi-fidelity Bayesian optimization with deep auto-regressive networks," *Advances in Neural Information Processing Systems*, 2021.

[50] S. Belakaria, J. Doppa, N. Fusi, and R. Sheth, "Bayesian optimization over iterative learners with structured responses: A budget-aware planning approach," in *International Conference on Artificial Intelligence and Statistics*, 2023.

[51] Y. Yang, K. Deng, and M. Zhu, "Multi-level training and Bayesian optimization for economical hyperparameter optimization," *arXiv:2007.09953*, 2020.

[52] E. Bonilla, K. Chai, and C. Williams, "Multi-task Gaussian process prediction," *Advances in Neural Information Processing Systems*, 2007.

[53] Y. Li, Y. Shen, J. Jiang, J. Gao, C. Zhang, and B. Cui, "MFES-HB: Efficient HyperBand with multi-fidelity quality measurements," in *AAAI Conference on Artificial Intelligence*, 2021.

[54] J. Zhao, X. Ning, E. Liu, B. Ru, Z. Zhou, T. Zhao, C. Chen, J. Zhang, Q. Liao, and Y. Wang, "Dynamic ensemble of low-fidelity experts: Mitigating NAS cold-start," *arXiv:2302.00932*, 2023.

[55] M. Feurer, B. Letham, F. Hutter, and E. Bakshy, "Practical transfer learning for Bayesian optimization," *arXiv:1802.02219*, 2018.

[56] S. Watanabe, N. Awad, M. Onishi, and F. Hutter, "Multi-objective tree-structured Parzen estimator meets meta-learning," *arXiv:2212.06751*, 2022.

[57] S. Watanabe, N. Awad, M. Onishi, and F. Hutter, "Speeding up multi-objective hyperparameter optimization by task similarity-based meta-learning for the tree-structured Parzen estimator," *arXiv:2212.06751*, 2023.

[58] A. Biedenkapp, H. Bozkurt, T. Eimer, F. Hutter, and M. Lindauer, "Dynamic algorithm configuration: foundation of a new meta-algorithmic framework," in *European Conference on Artificial Intelligence*, 2020.

[59] S. Adriaensen, A. Biedenkapp, G. Shala, N. Awad, T. Eimer, M. Lindauer, and F. Hutter, "Automated dynamic algorithm configuration," *Journal of Artificial Intelligence Research*, vol. 75, 2022.

[60] M. Jaderberg, V. Dalibard, S. Osindero, W. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, *et al.*, "Population based training of neural networks," *arXiv:1711.09846*, 2017.

[61] J. Parker-Holder, V. Nguyen, and S. Roberts, "Provably efficient online hyperparameter optimization with population-based bandits," *arXiv:2002.02518*, 2020.

[62] B. Zhang, R. Rajan, L. Pineda, N. Lambert, A. Biedenkapp, K. Chua, F. Hutter, and R. Calandra, "On the importance of hyperparameter optimization for model-based reinforcement learning," in *International Conference on Artificial Intelligence and Statistics*, 2021.

[63] J. Liang, S. Gonzalez, H. Shahrzad, and R. Miikkulainen, "Regularized evolutionary population-based training," in *Genetic and Evolutionary Computation Conference*, 2021.

[64] A. Dushatskiy, A. Chebykin, T. Alderliesten, and P. Bosman, "Multi-objective population based training," in *International Conference on Machine Learning*, 2023.

[65] S. Belakaria, A. Deshwal, and J. Doppa, "Multi-fidelity multi-objective Bayesian optimization: An output space entropy search approach," in *AAAI Conference on artificial intelligence*, 2020.

[66] M. Gelbart, J. Snoek, and R. Adams, "Bayesian optimization with unknown constraints," *arXiv:1403.5607*, 2014.

[67] J. Gardner, M. Kusner, Z. Xu, K. Weinberger, and J. Cunningham, "Bayesian optimization with inequality constraints," in *International Conference on Machine Learning*, 2014.

[68] B. Letham, B. Karrer, G. Ottoni, and E. Bakshy, "Constrained Bayesian optimization with noisy experiments," *Bayesian Analysis*, vol. 14, 2019.

[69] S. Watanabe and F. Hutter, "c-TPE: Generalizing tree-structured Parzen estimator with inequality constraints for continuous and categorical hyperparameter optimization," *arXiv:2211.14411*, 2022.

[70] S. Watanabe and F. Hutter, "c-TPE: Tree-structured Parzen estimator with inequality constraints for expensive hyperparameter optimization," *arXiv:2211.14411*, 2023.

[71] L. Zimmer, M. Lindauer, and F. Hutter, "Auto-PyTorch: Multi-fidelity metalearning for efficient and robust AutoDL," *Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[72] F. Pfisterer, L. Schneider, J. Moosbauer, M. Binder, and B. Bischl, "YAHPO Gym – an efficient multi-objective multi-fidelity benchmark for hyperparameter optimization," in *International Conference on Automated Machine Learning*, 2022.

[73] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, 2012.

[74] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," *Advances in Neural Information Processing Systems*, 2011.

[75] S. Watanabe, "Tree-structured Parzen estimator: Understanding its algorithm components and their roles for better empirical performance," *arXiv:2304.11127*, 2023.

[76] S. Müller and F. Hutter, "TrivialAugment: Tuning-free yet state-of-the-art data augmentation," in *International Conference on Computer Vision*, 2021.

A. Benchmark Problems

We first note that since the Branin and the Hartmann functions must be minimized, our functions have different signs from the prior literature that aims to maximize objective functions and when $\boldsymbol{z} = [z_1, z_2, \ldots, z_K] \in \mathbb{R}^K$, our examples take $\boldsymbol{z} = [z, z, \ldots, z] \in \mathbb{R}^K$. However, if users wish, users can specify $\boldsymbol{z}$ as $\boldsymbol{z} = [z_1, z_2, \ldots, z_K]$ from `fidel_dim`.

A.1. Branin Function

The Branin function is the following $2D$ function that has 3 global minimizers and no local minimizer:

$$f(x_1, x_2) = a(x_2 - bx_1^2 + cx_1 - r)^2 + s(1-t)\cos x_1 + s \tag{11}$$

where $\boldsymbol{x} \in [-5, 10] \times [0, 15]$, $a = 1$, $b = 5.1/(4\pi^2)$, $c = 5/\pi$, $r = 6$, $s = 10$, and $t = 1/(8\pi)$. The multi-fidelity Branin function was invented by Kandasamy *et al.* [19] and it replaces b, c, t with the following $b_{\boldsymbol{z}}, c_{\boldsymbol{z}}, t_{\boldsymbol{z}}$:

$$\begin{aligned} b_{\boldsymbol{z}} &\coloneqq b - \delta_b(1 - z_1), \\ c_{\boldsymbol{z}} &\coloneqq c - \delta_c(1 - z_2), \text{ and} \\ t_{\boldsymbol{z}} &\coloneqq t + \delta_t(1 - z_3), \end{aligned} \tag{12}$$

where $\boldsymbol{z} \in [0, 1]^3$, $\delta_b = 10^{-2}$, $\delta_c = 10^{-1}$, and $\delta_t = 5 \times 10^{-3}$. $\delta.$ controls the rank correlation between low- and high-fidelities and higher $\delta.$ yields less correlation. The runtime function for the multi-fidelity Branin function is computed as [1]:

$$\tau(\boldsymbol{z}) \coloneqq C(0.05 + 0.95 z_1^{3/2}) \tag{13}$$

[1] See the implementation of Kandasamy *et al.* [19]: `branin_mf.py` at `https://github.com/dragonfly/dragonfly/`.

where $C \in \mathbb{R}_+$ defines the maximum runtime to evaluate f.

A.2. Hartmann Function

The following Hartmann function has 4 local minimizers for the $3D$ case and 6 local minimizers for the $6D$ case:

$$f(\boldsymbol{x}) \coloneqq -\sum_{i=1}^{4} \alpha_i \exp\left[-\sum_{j=1}^{3} A_{i,j}(x_j - P_{i,j})^2\right] \tag{14}$$

where $\boldsymbol{\alpha} = [1.0, 1.2, 3.0, 3.2]^\top$, $\boldsymbol{x} \in [0,1]^D$, A for the $3D$ case is

$$A = \begin{bmatrix} 3 & 10 & 30 \\ 0.1 & 10 & 35 \\ 3 & 10 & 30 \\ 0.1 & 10 & 35 \end{bmatrix}, \tag{15}$$

A for the $6D$ case is

$$A = \begin{bmatrix} 10 & 3 & 17 & 3.5 & 1.7 & 8 \\ 0.05 & 10 & 17 & 0.1 & 8 & 14 \\ 3 & 3.5 & 1.7 & 10 & 17 & 8 \\ 17 & 8 & 0.05 & 10 & 0.1 & 14 \end{bmatrix}, \tag{16}$$

P for the $3D$ case is

$$P = 10^{-4} \times \begin{bmatrix} 3689 & 1170 & 2673 \\ 4699 & 4387 & 7470 \\ 1091 & 8732 & 5547 \\ 381 & 5743 & 8828 \end{bmatrix}, \tag{17}$$

and P for the $6D$ case is

$$P = 10^{-4} \times \begin{bmatrix} 1312 & 1696 & 5569 & 124 & 8283 & 5886 \\ 2329 & 4135 & 8307 & 3736 & 1004 & 9991 \\ 2348 & 1451 & 3522 & 2883 & 3047 & 6650 \\ 4047 & 8828 & 8732 & 5743 & 1091 & 381 \end{bmatrix}. \tag{18}$$

The multi-fidelity Hartmann function was invented by Kandasamy *et al.* [19] and it replaces $\boldsymbol{\alpha}$ with the following $\boldsymbol{\alpha}_{\boldsymbol{z}}$:

$$\boldsymbol{\alpha}_{\boldsymbol{z}} \coloneqq \delta(1 - \boldsymbol{z}) \tag{19}$$

where $\boldsymbol{z} \in [0, 1]^4$ and $\delta = 0.1$ is the factor that controls the rank correlation between low- and high-fidelities. Higher δ yields less correlation. The runtime function of the multi-fidelity Hartmann function is computed as [2]:

$$\tau(\boldsymbol{z}) = \frac{1}{10} + \frac{9}{10}\frac{z_1 + z_2^3 + z_3 z_4}{3} \tag{20}$$

for the $3D$ case and

$$\tau(\boldsymbol{z}) = \frac{1}{10} + \frac{9}{10}\frac{z_1 + z_2^2 + z_3 + z_4^3}{4} \tag{21}$$

for the $6D$ case where $C \in \mathbb{R}_+$ defines the maximum runtime to evaluate f.

A.3. Tabular & Surrogate Benchmarks

In this thesis, we used the MLP benchmark in Table 6 of HPOBench [4], HPOlib [16], JAHS-Bench-201 [6], and LCBench [71] in YAHPOBench [72].

HPOBench is a collection of tabular, surrogate, and raw benchmarks. In our example, we have the MLP (multi-layer perceptron) benchmark, which is a tabular benchmark, in Table 6 of the HPOBench paper [4]. This benchmark has 8 classification tasks and provides the validation accuracy, runtime, F1 score, and precision for each configuration at epochs of $\{3, 9, 27, 81, 243\}$. The search space of MLP benchmark in HPOBench is provided in Table 3.

HPOlib is a tabular benchmark for neural networks on regression tasks (Slice Localization, Naval Propulsion, Protein Structure, and Parkinsons Telemonitoring). This benchmark has 4 regression tasks and provides the number of parameters, runtime, and training and validation mean squared error (MSE) for each configuration at each epoch. The search space of HPOlib is provided in Table 4.

[2] See the implementation of Kandasamy *et al.* [19]: `hartmann3_2_mf.py` for the $3D$ case and `hartmann6_4_mf.py` for the $6D$ case at `https://github.com/dragonfly/dragonfly/`.

Table 3.: The search space of the MLP benchmark in HPOBench (5 discrete + 1 fidelity parameters). Note that we have 2 fidelity parameters only for the raw benchmark. Each benchmark has performance metrics of 30000 possible configurations with 5 random seeds.

Hyperparameter	Choices
L2 regularization	$[10^{-8}, 1.0]$ with 10 evenly distributed grids
Batch size	$[4, 256]$ with 10 evenly distributed grids
Initial learning rate	$[10^{-5}, 1.0]$ with 10 evenly distributed grids
Width	$[16, 1024]$ with 10 evenly distributed grids
Depth	$\{1, 2, 3\}$
Epoch (**Fidelity**)	$\{3, 9, 27, 81, 243\}$

JAHS-Bench-201 is an XGBoost surrogate benchmark for neural networks on image classification tasks (CIFAR10, Fashion-MNIST, and Colorectal Histology). This benchmark has 3 image classification tasks and provides FLOPS, latency, runtime, architecture size in megabytes, test accuracy, training accuracy, and validation accuracy for each configuration with two fidelity parameters: image resolution and epoch. The search space of JAHS-Bench-201 is provided in Table 5.

LCBench is a random-forest surrogate benchmark for neural networks on OpenML datasets. This benchmark has 34 tasks and provides training/test/validation accuracy, losses, balanced accuracy, and runtime at each epoch. The search space of HPOlib is provided in Table 6.

Note that the task IDs for each benchmark dataset are listed in Table 7.

Table 4.: The search space of HPOlib (6 discrete + 3 categorical + 1 fidelity parameters). Each benchmark has performance metrics of 62208 possible configurations with 4 random seeds.

Hyperparameter	Choices
Batch size	$\{2^3, 2^4, 2^5, 2^6\}$
Initial learning rate	$\{5 \times 10^{-4}, 10^{-3}, 5 \times 10^{-3}, 10^{-2}, 5 \times 10^{-2}, 10^{-1}\}$
Number of units {1,2}	$\{2^4, 2^5, 2^6, 2^7, 2^8, 2^9\}$
Dropout rate {1,2}	$\{0.0, 0.3, 0.6\}$
Learning rate scheduler	{`cosine`, `constant`}
Activation function {1,2}	{`relu`, `tanh`}
Epoch (**Fidelity**)	$[1, 100]$

Table 5.: The search space of JAHS-Bench-201 (2 continuous + 2 discrete + 8 categorical + 2 fidelity parameters). JAHS-Bench-201 is an XGBoost surrogate benchmark and the outputs are deterministic.

Hyperparameter	Range or choices
Learning rate	$[10^{-3}, 1]$
L2 regularization	$[10^{-5}, 10^{-2}]$
Activation function	{`ReLU`, `Hardswish`, `Mish`}
Trivial augment [76]	{`True`, `False`}
Depth multiplier	$\{1, 2, 3\}$
Width multiplier	$\{2^2, 2^3, 2^4\}$
Cell search space (NAS-Bench-201 [17], Edge 1 – 6)	{`none`, `avg-pool-3x3`, `bn-conv-1x1`, `bn-conv-3x3`, `skip-connection`}
Epoch (**Fidelity**)	$[1, 200]$
Resolution (**Fidelity**)	$[0.0, 1.0]$

Table 6.: The search space of LCBench (3 discrete + 4 continuous + 1 fidelity parameters). Although the original LCBench is a collection of 2000 random configurations, YAHPOBench created random-forest surrogates over the 2000 observations. Users can choose deterministic or non-deterministic outputs.

Hyperparameter	Choices
Batch size	$[2^4, 2^9]$
Max number of units	$[2^6, 2^{10}]$
Number of layers	$[1, 5]$
Initial learning rate	$[10^{-4}, 10^{-1}]$
L2 regularization	$[10^{-5}, 10^{-1}]$
Max dropout rate	$[0.0, 1.0]$
Momentum	$[0.1, 0.99]$
Epoch (**Fidelity**)	$[1, 52]$

Table 7.: The correspondance of task IDs and their tasks. As HPOBench and LCBench use OpenML tasks, we show the OpenML task ID for them.

Task ID	HPOBench	HPOlib	JAHS-Bench-201	LCBench
1	167104	Slice Localization	CIFAR10	3945
2	167184	Protein Structure	Fashion-MNIST	7593
3	189905	Naval Propulsion	Colorectal Histology	34539
4	167161	Parkinsons Telemonitoring	–	126025
5	167181	–	–	126026
6	167190	–	–	126029
7	189906	–	–	146212
8	167168	–	–	167104
9	–	–	–	167149
10	–	–	–	167152
11	–	–	–	167161
12	–	–	–	167168
13	–	–	–	167181
14	–	–	–	167184
15	–	–	–	167185
16	–	–	–	167190
17	–	–	–	167200
18	–	–	–	167201
19	–	–	–	168329
20	–	–	–	168330
21	–	–	–	168331
22	–	–	–	168335
23	–	–	–	168868
24	–	–	–	168908
25	–	–	–	168910
26	–	–	–	189354
27	–	–	–	189862
28	–	–	–	189865
29	–	–	–	189866
30	–	–	–	189873
31	–	–	–	189905
32	–	–	–	189906
33	–	–	–	189908
34	–	–	–	189909

B. Tool Usage

In this chapter, we describe more details on our wrapper.

B.1. Wrapper Object (`ObjectiveFuncWrapper`) Arguments

The arguments of `ObjectiveFuncWrapper` object are as follows:

- `obj_func`: the objective function that takes (`eval_config`, `fidels`, `seed`, `**data_to_scatter`) as arguments and returns $f(\boldsymbol{x}|\boldsymbol{a})$ and $\tau(\boldsymbol{x}|\boldsymbol{a})$ where `eval_config` is `dict[str, Any]`,
- `launch_multiple_wrappers_from_user_side (bool)`: whether to instantiate multiple wrappers from user side as in Section 2.4.2 or application side as in Section 2.4.3,
- `ask_and_tell (bool)`: whether to use SCS (`True`) or not,
- `save_dir_name (str | None)`: the results and the required information will be stored in `mfhpo-simulator-info/<save_dir_name>/`,
- `n_workers (int)`: the number of parallel workers P,
- `n_evals (int)`: the number of evaluations to get,
- `n_actual_evals_in_opt (int)`: the number of HP configurations to be evaluated in an optimizer which is used only for checking if no hang happens (should take `n_evals + n_workers`),
- `continual_max_fidel (int | None)`: when users would like to restart an evaluation from an intermediate state, the maximum fidelity value for the target fidelity must be provided. Note that the restart is allowed only if there is only one fidelity parameter, i.e. $K = 1$. If `None`, no restart happens,

- `runtime_key (str)`: the key of the runtime in the returned value of `obj_func`,
- `obj_keys (list[str])`: the keys of the objective and constraint names in the returned value of `obj_func` and the values of the specified keys will be stored in the result file,
- `fidel_keys (list[str] | None)`: the keys of the fidelity parameters in input `fidels`,
- `seed (int | None)`: the random seed to be used in the wrapper,
- `max_waiting_time (float)`: the maximum waiting time for each worker and if each worker did not get any update for this amount of time, it will return `inf`,
- `store_config (bool)`: whether to store HP configurations, fidelities, and seed used for each evaluation,
- `store_actual_cumtime (bool)`: whether to store actual runtimes at each iteration,
- `check_interval_time (float)`: how often each worker should check whether a new job can be assigned to it.
- `allow_parallel_sampling (bool)`: whether samplings can happen in parallel and the default is `False`,
- `config_tracking (bool)`: whether to validate `config_id` provided from the user side. It slows down down the simulation when `n_evals` is large (> 3000), but it is recommended to avoid unexpected bugs when we use the continual setup,
- `max_total_eval_time (float)`: the maximum total evaluation time for the optimization. For example, if `max_total_eval_time = 3600` is specified and the simulation has not finished `n_evals` evaluations in simulated runtime of 3600 seconds, the simulation will be terminated. It is useful to combine with a large `n_evals`,
- `expensive_sampler (bool)`: whether to use MCS with Algorithm 2 or not. If a query overhead is large, it should be set `False`,

- `tmp_dir (str | None)`: the temporary directory useful for cluster usage. By using this argument, the root directory of the data storage path will be `<tmp_dir>`. More specifically, `<tmp_dir>/mfhpo-simulator-info/<save_dir_name>` will be used, and

- `worker_index (int | None)`: the explicit worker index specifier useful for HPO libraries discussed in Section 2.4.2. If not specified, the index for each worker will be automatically allocated, but it could be unstable. For example, automatic index allocation failed with 0.01% of the probability for `n_workers = 8` in our environment.

Note that `data_to_scatter` is especially important when an optimizer uses multi-processing packages such as `dask`, which deserialize `obj_func` every time we call. By passing large-size data via `data_to_scatter`, the time for (de)serialization will be negligible if optimizers use `dask.scatter` or something similar internally. We kindly ask readers to check any updates to the arguments at `https://github.com/nabenabe0928/mfhpo-simulator/`.

B.2. Wrapper Interface

In our package, `ObjectiveFuncWrapper` is the main module and it provides three different instantiation options: (1) function wrapper for HPO libraries discussed in Section 2.4.3 (e.g. DEHB and SMAC3), (2) function wrapper for HPO libraries discussed in Section 2.4.2 (e.g. NePS, BOHB, and MPI-based optimizers), and (3) function and optimizer wrapper for the ask-and-tell interface. In this thesis, we discuss the usage of Options 1 and 2, and we kindly ask users to refer to our repository [1] for Option 3.

An instance of `ObjectiveFuncWrapper` serves as an objective function and we just need to pass it to an optimizer as shown in Figure 2 (**Right**). When optimizers take a different interface, we can easily modify the interface via inheritance:

[1] `https://github.com/nabenabe0928/mfhpo-simulator/tree/main/examples/ask_and_tell`

Listing B.1: An example of inheritance for a different interface.

```
class MyObjectiveFuncWrapper(ObjectiveFuncWrapper):
    def __call__(self, config, budget):
        # modify config into dict[str, Any]
        return super().__call__(
            eval_config=config,
            fidels={self.fidel_keys[0]: budget}
        )
```

In Listing B.1, the optimizer requires the objective function to have arguments named `config` instead of `eval_config` and `budget` instead of `fidels`. Then we need to somehow modify `config` into the format of `eval_config` (`dict[str, Any]`) if `config` is not `dict[str, Any]`. We can also easily deactivate the MFO setting if `fidel_keys=None` is specified:

Listing B.2: An example of inheritance for non-MFO setup.

```
class MyObjectiveFuncWrapper(ObjectiveFuncWrapper):
    def __call__(self, eval_config):
        # fidel_keys must be None; otherwise, get an error
        return super().__call__(eval_config=eval_config)
```

See our repository [2] for more examples.

[2] `https://github.com/nabenabe0928/mfhpo-simulator/tree/main/examples`

C. Additional Results

In this chapter, we show additional results that we did not present in the main part.

C.1. Performance over Time for Each Task

Figures 14–65 present the performance over time for each setup. To plot the figures, we used `benchmark_simulator.utils.get_performance_over_time_from_paths` provided in our package and the weak-color bands were calculated by standard error over 30 random seeds. The y-axis shows the minimum cumulative objective value and we consistently minimize the objective function for all the setups.

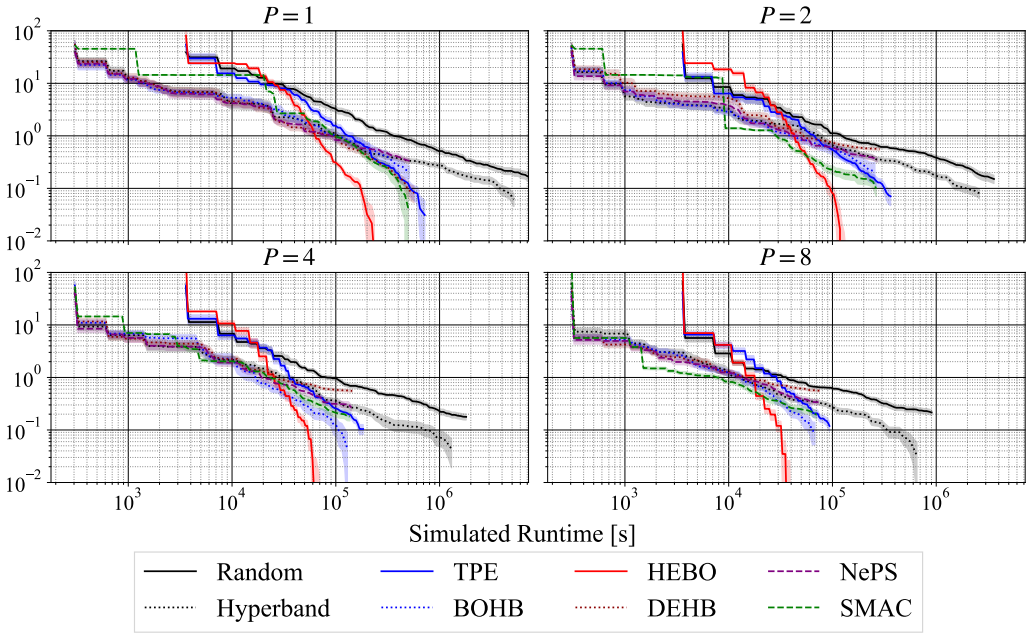


Figure 14.: The performance over time on the Branin function.

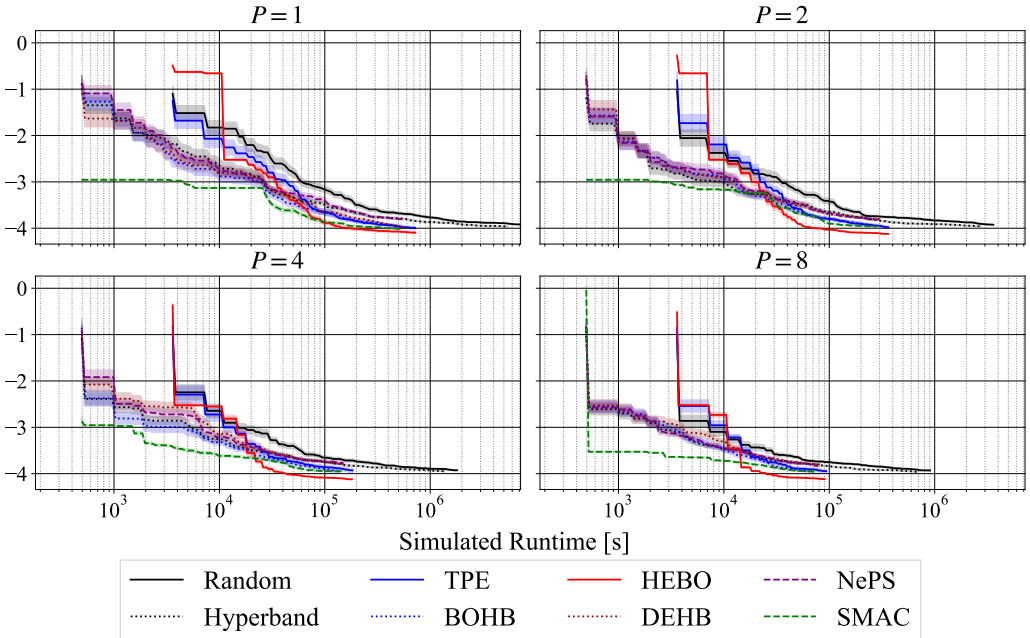


Figure 15.: The performance over time on the 3D Hartmann function.

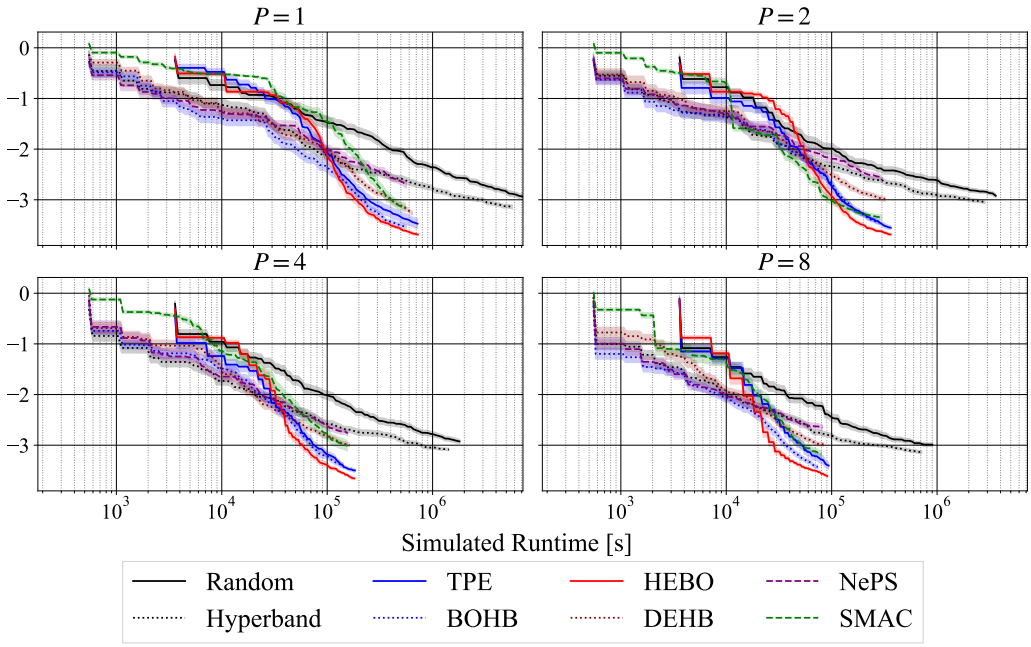


Figure 16.: The performance over time on the 6D Hartmann function.

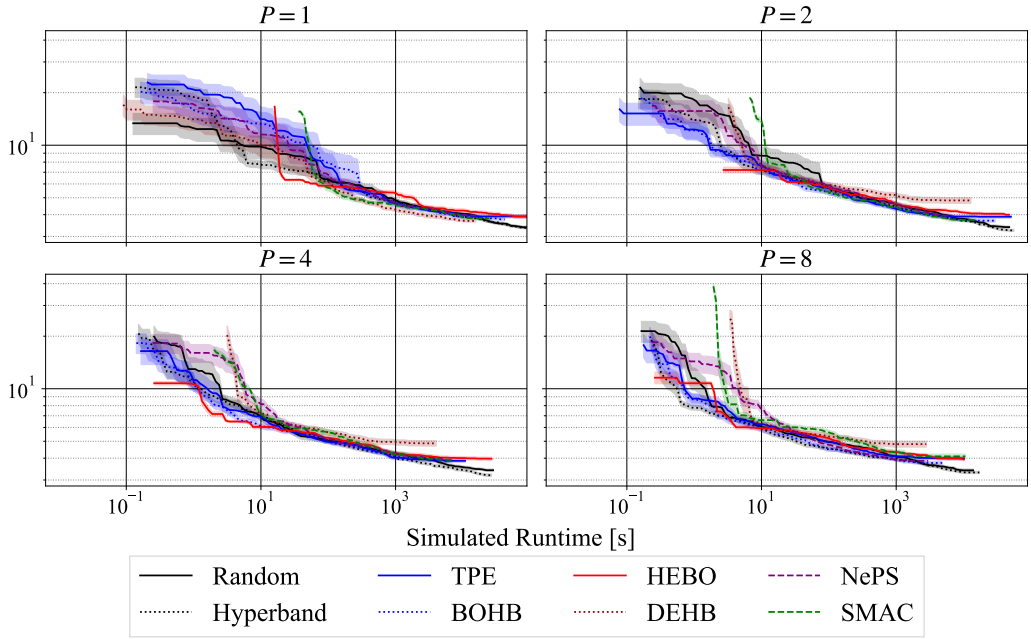


Figure 17.: The performance over time on OpenML ID 167104 from HPOBench.

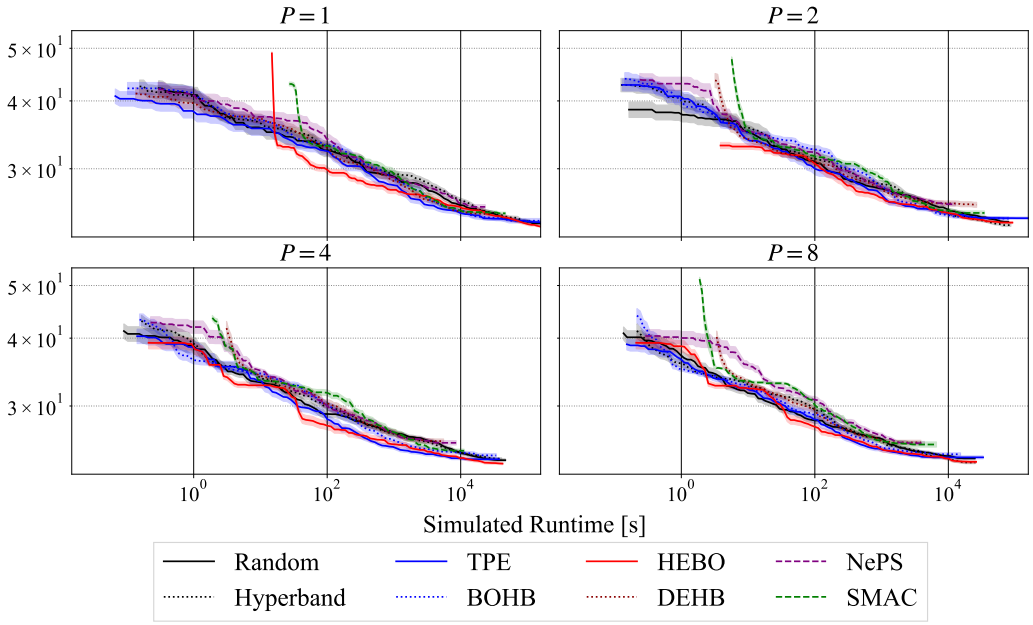


Figure 18.: The performance over time on OpenML ID 167184 from HPOBench.

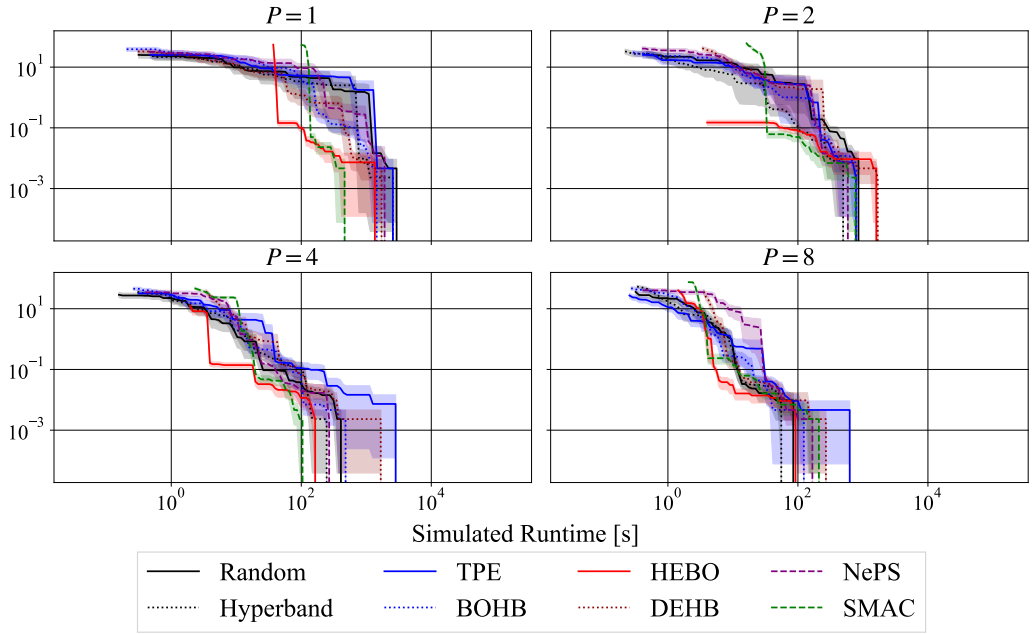


Figure 19.: The performance over time on OpenML ID 189905 from HPOBench.

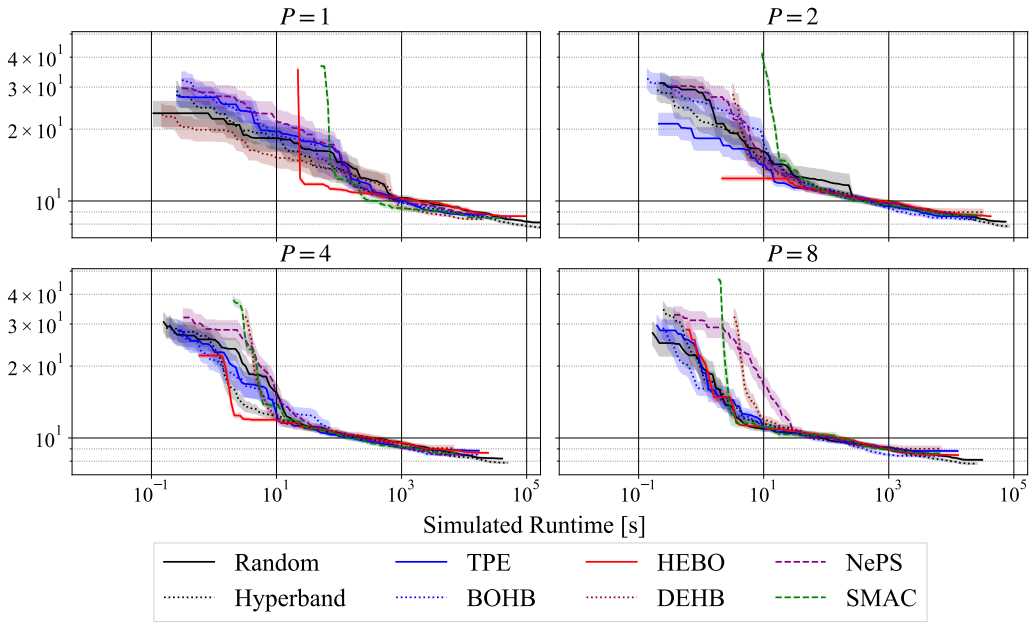


Figure 20.: The performance over time on OpenML ID 167161 from HPOBench.

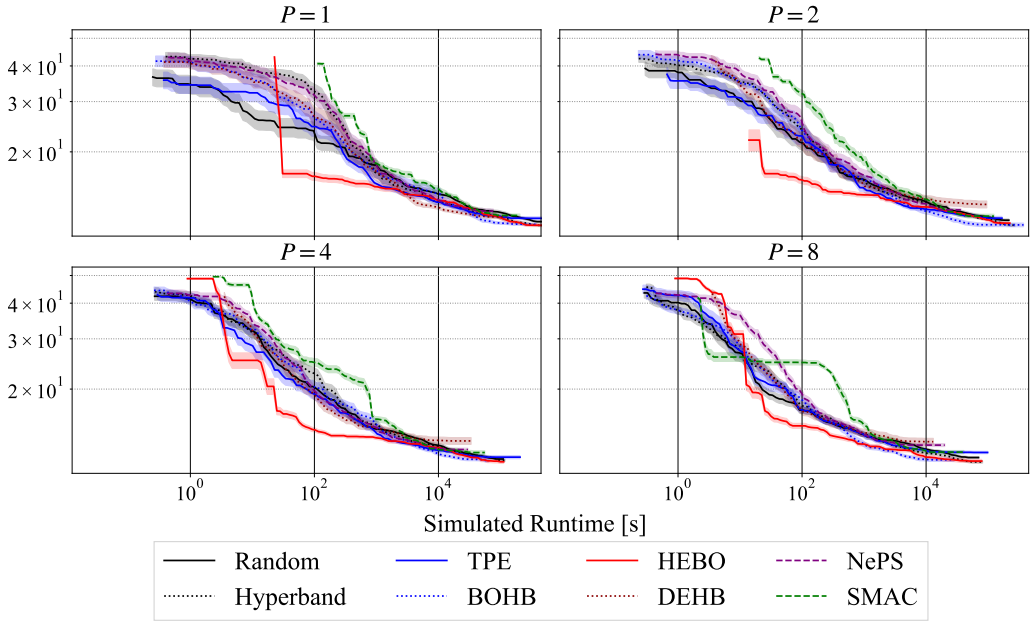


Figure 21.: The performance over time on OpenML ID 167181 from HPOBench.

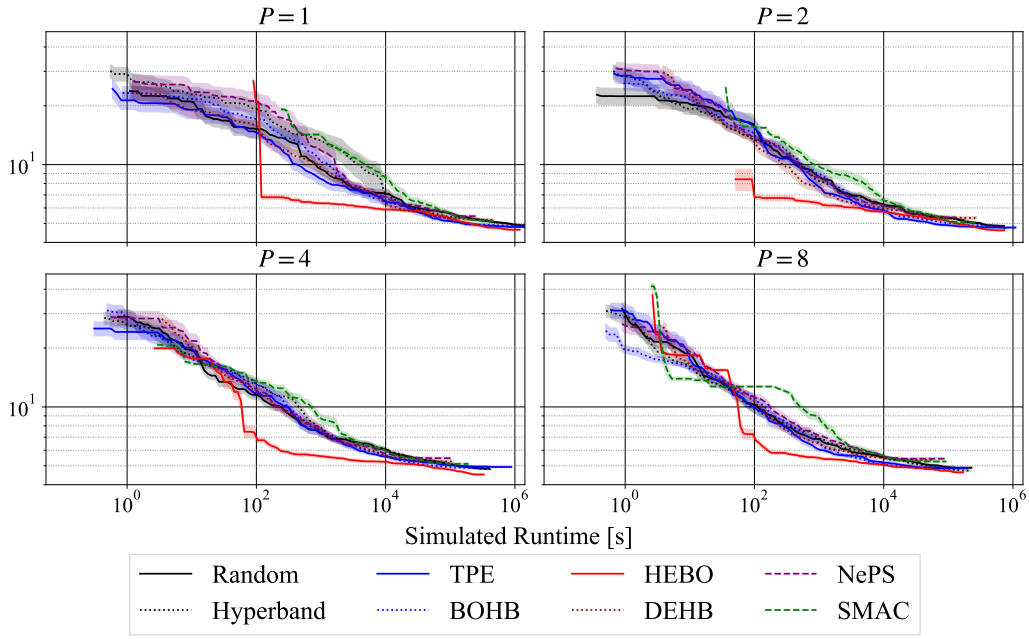


Figure 22.: The performance over time on OpenML ID 167190 from HPOBench.

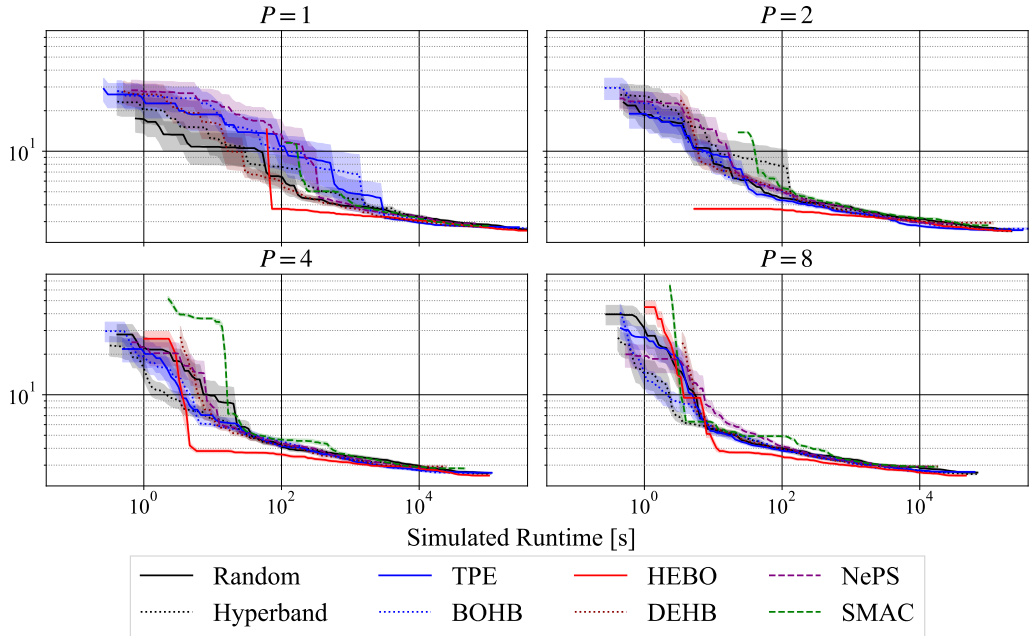


Figure 23.: The performance over time on OpenML ID 189906 from HPOBench.

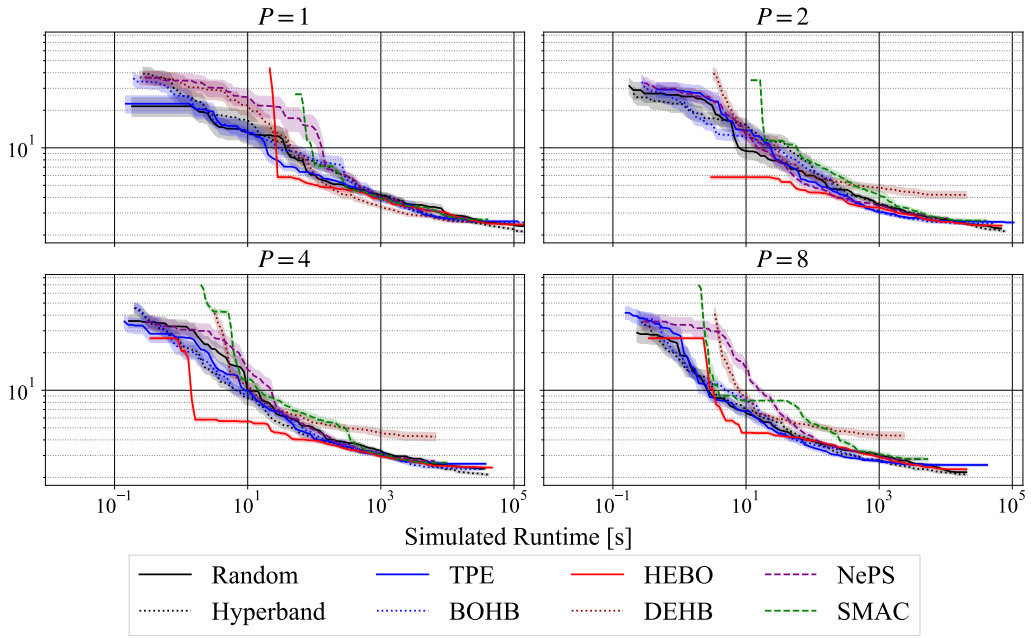


Figure 24.: The performance over time on OpenML ID 167168 from HPOBench.

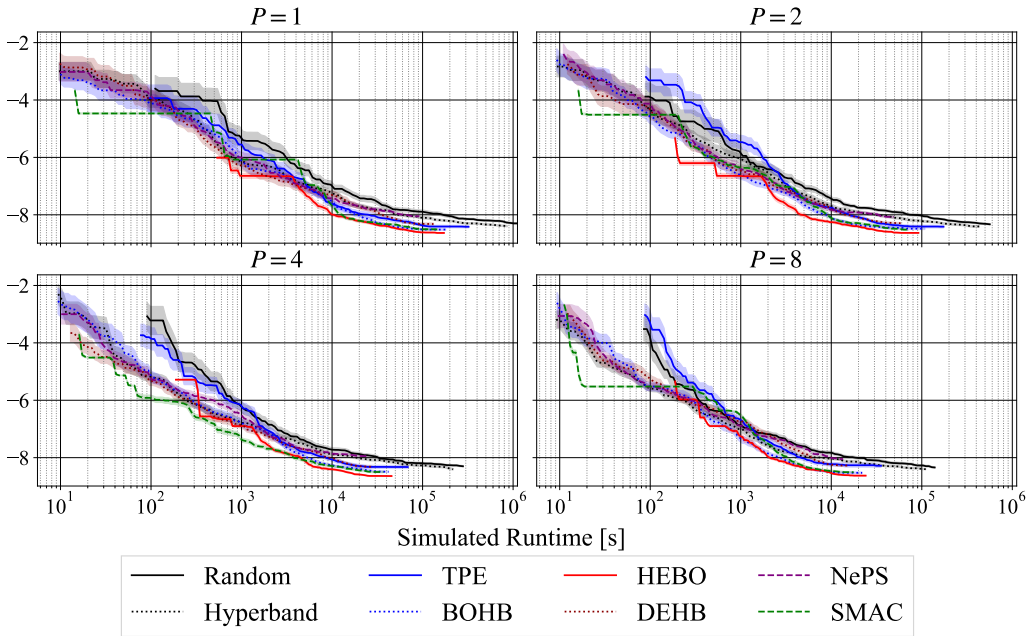


Figure 25.: The performance over time on Slice Localization of HPOlib.

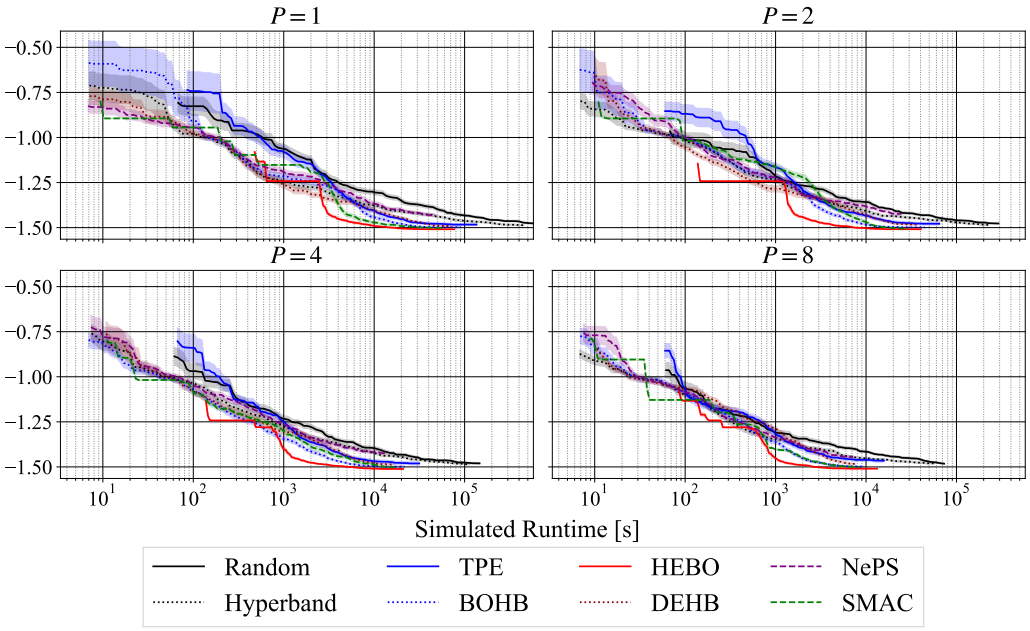


Figure 26.: The performance over time on Protein Structure of HPOlib.

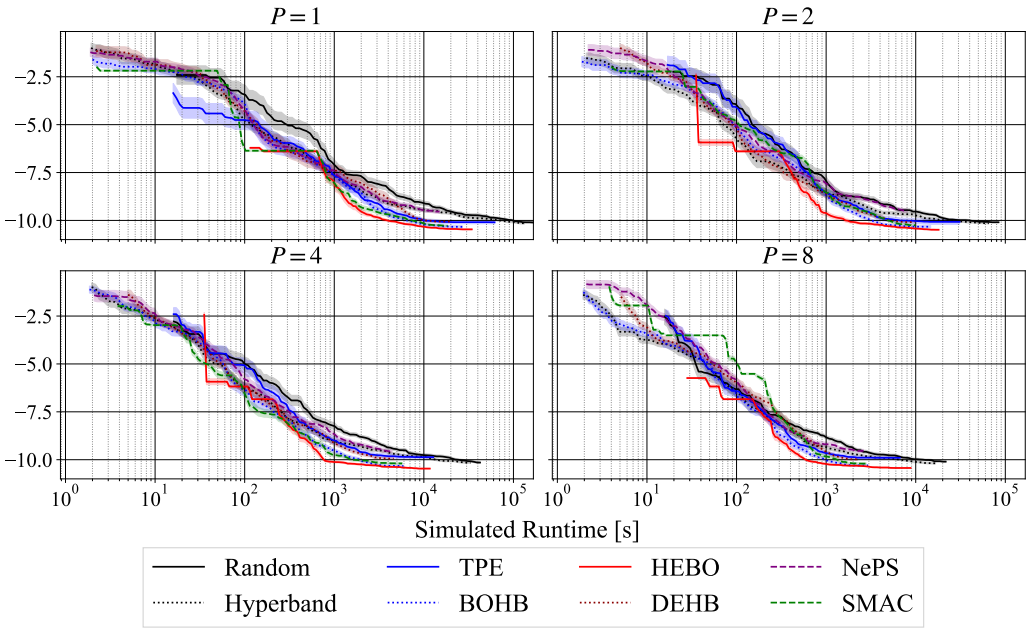


Figure 27.: The performance over time on Naval Propulsion of HPOlib.

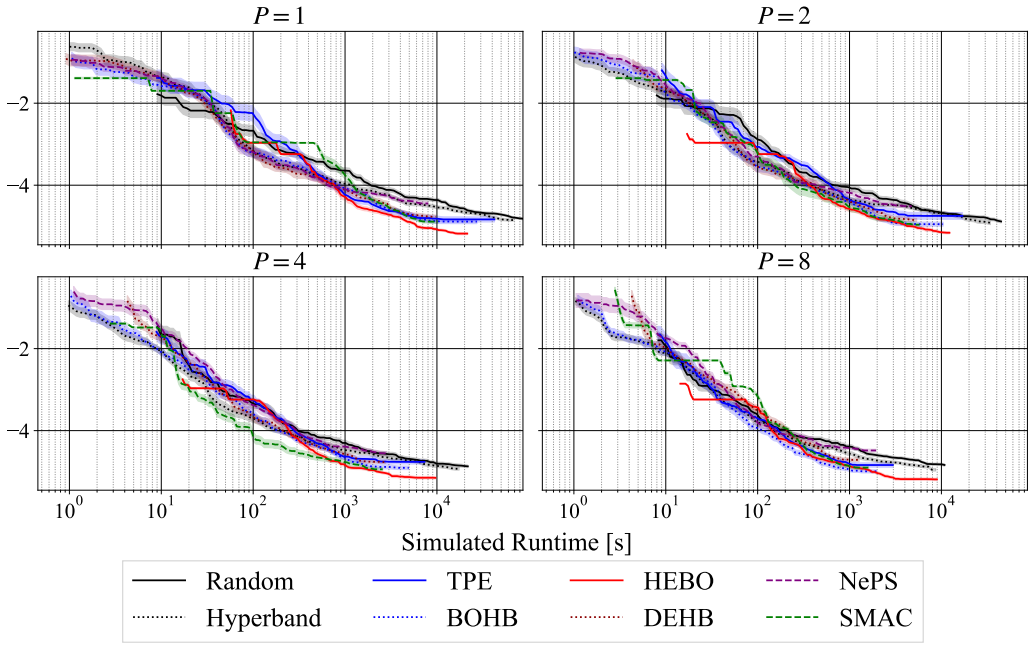


Figure 28.: The performance over time on Parkinsons Telemonitoring of HPOlib.

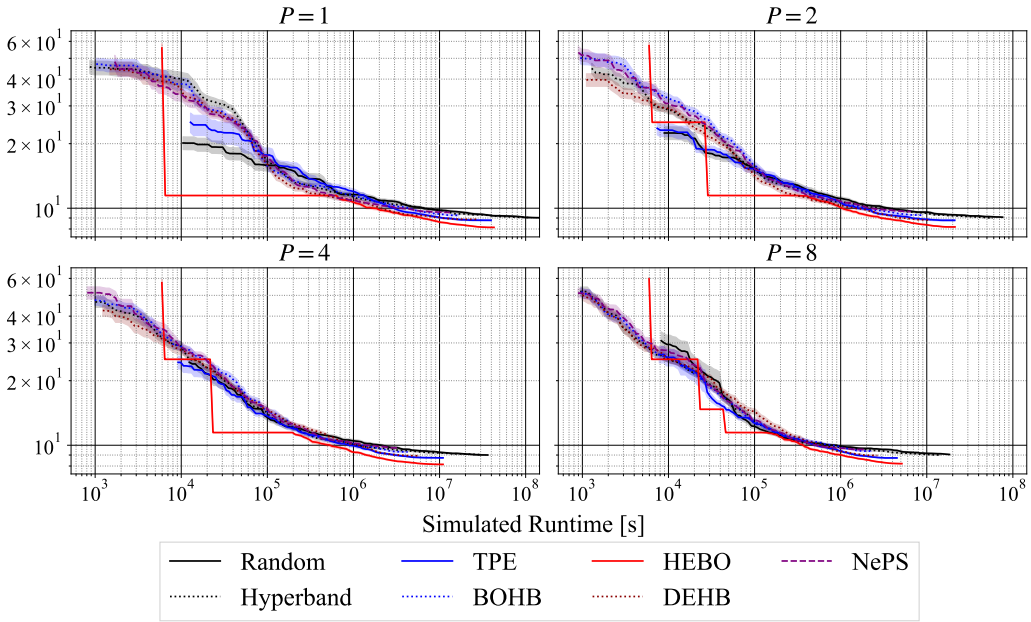


Figure 29.: The performance over time on CIFAR10 of JAHS-Bench-201.

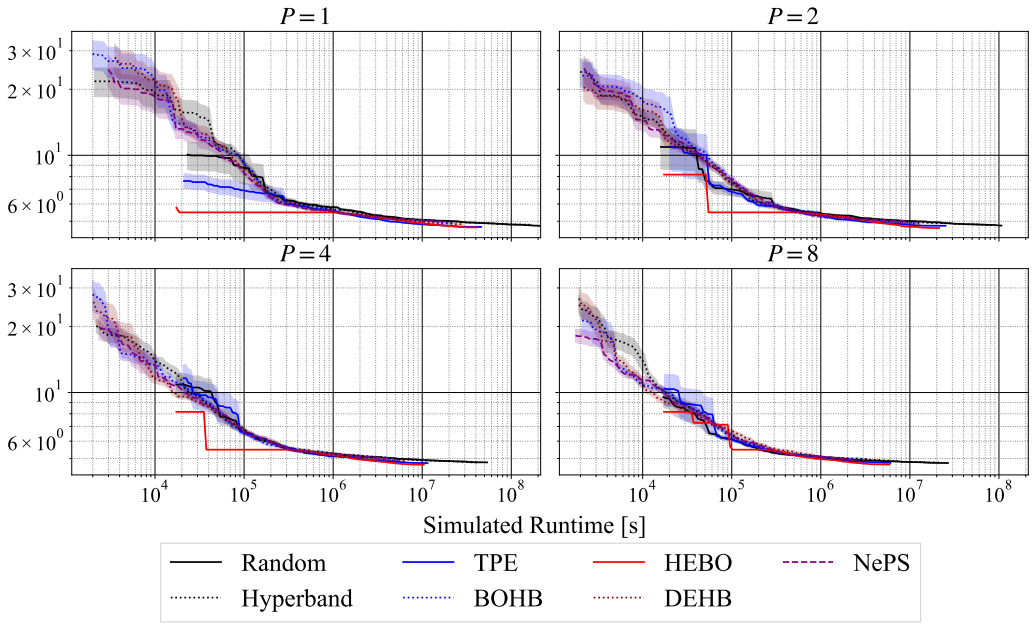


Figure 30.: The performance over time on Fashion-MNIST of JAHS-Bench-201.

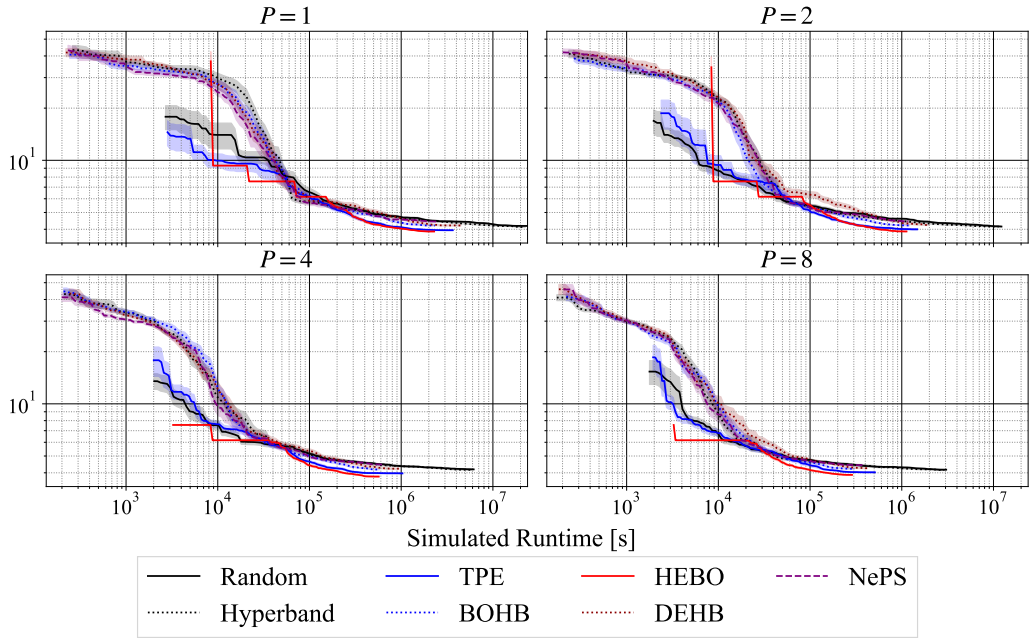


Figure 31.: The performance over time on Colorectal Histology of JAHS-Bench-201.

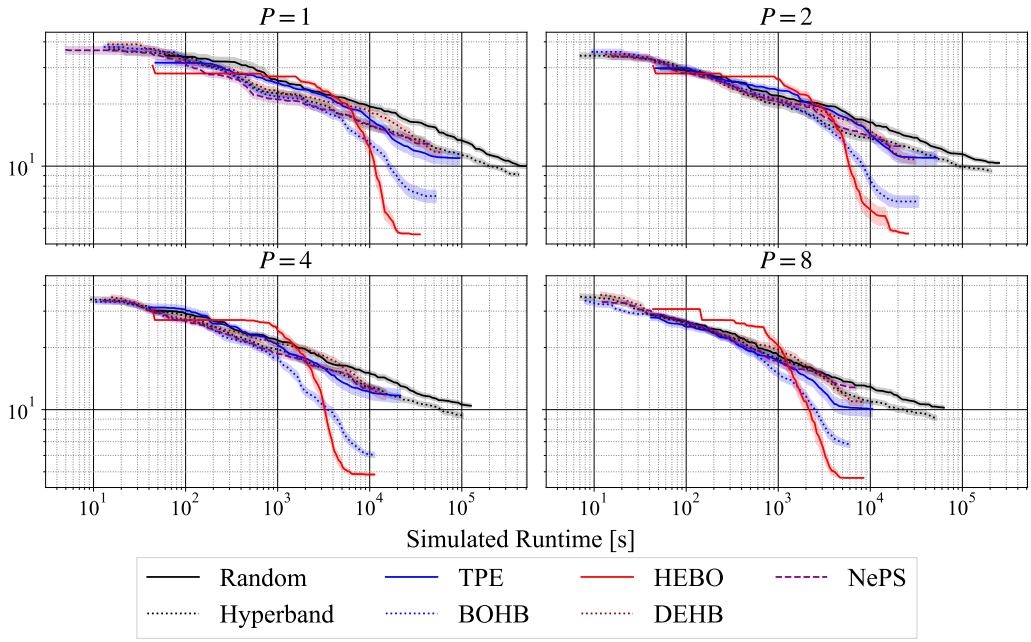


Figure 32.: The performance over time on OpenML ID 3945 from LCBench.

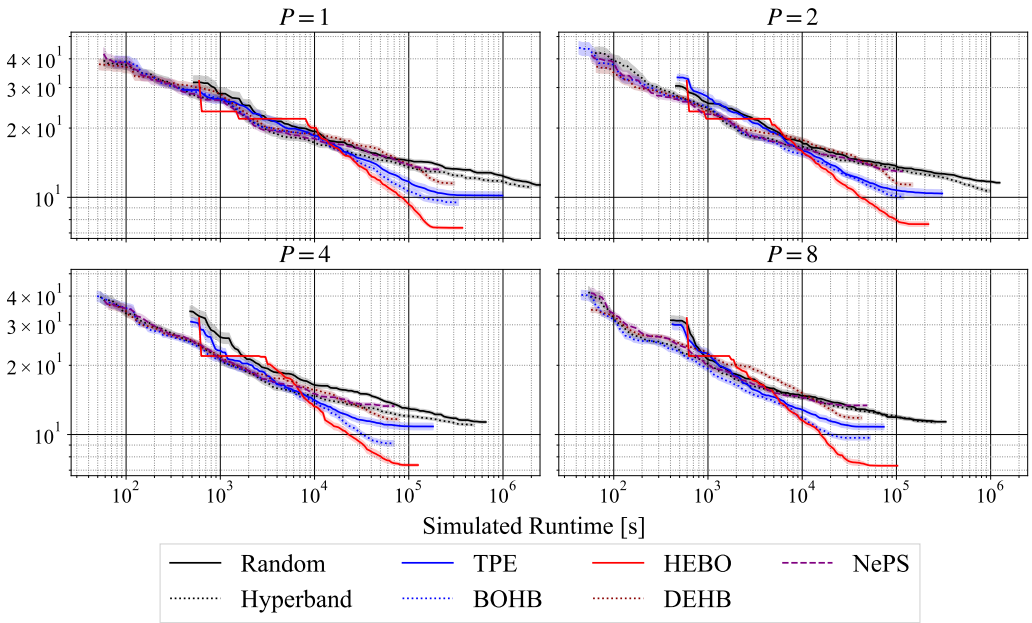


Figure 33.: The performance over time on OpenML ID 7593 from LCBench.

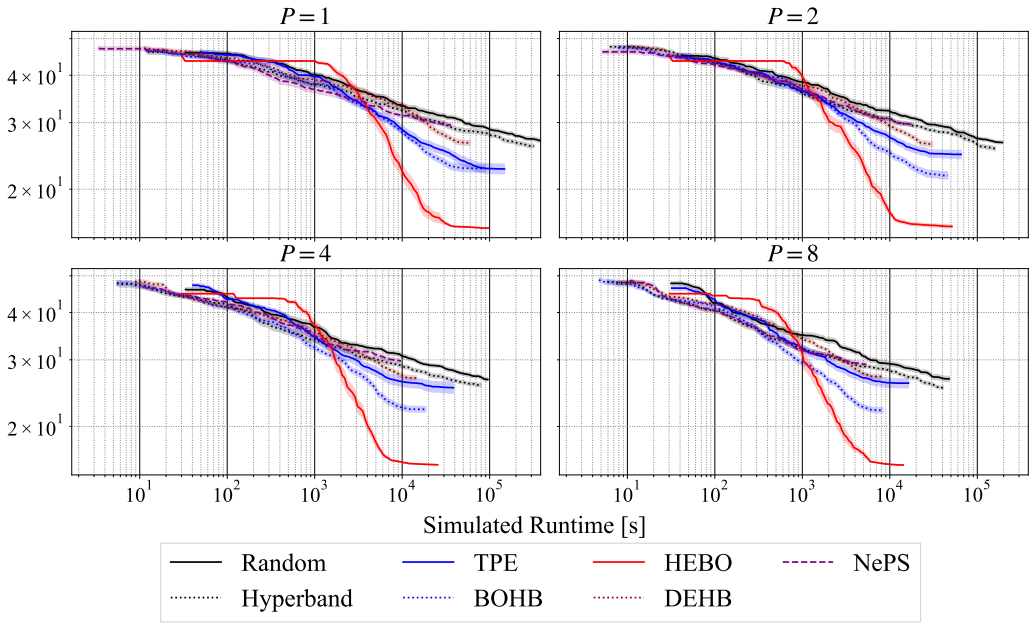


Figure 34.: The performance over time on OpenML ID 34539 from LCBench.

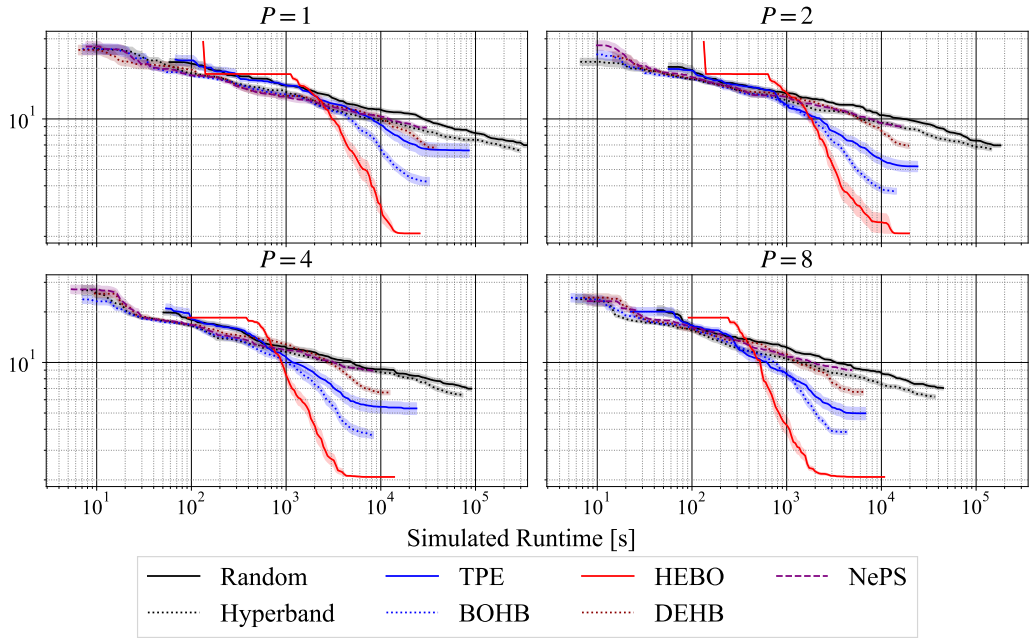


Figure 35.: The performance over time on OpenML ID 126025 from LCBench.

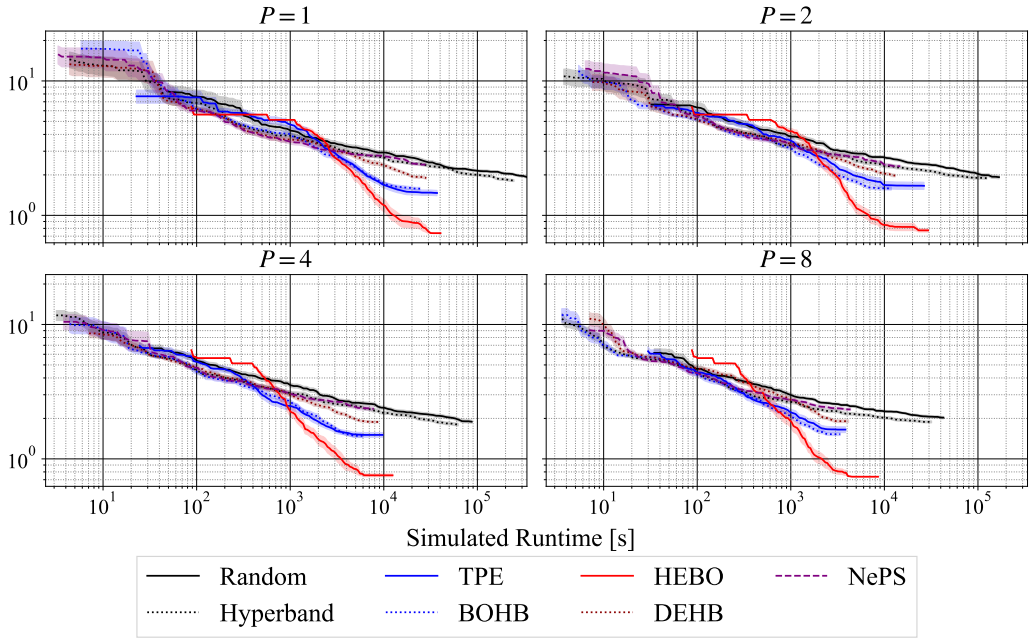


Figure 36.: The performance over time on OpenML ID 126026 from LCBench.

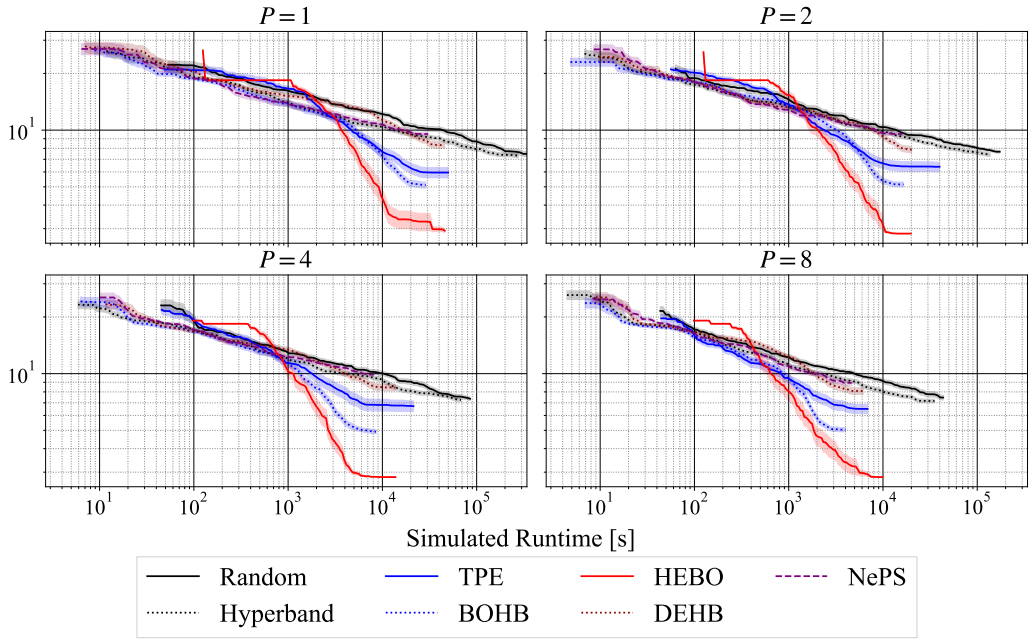


Figure 37.: The performance over time on OpenML ID 126029 from LCBench.

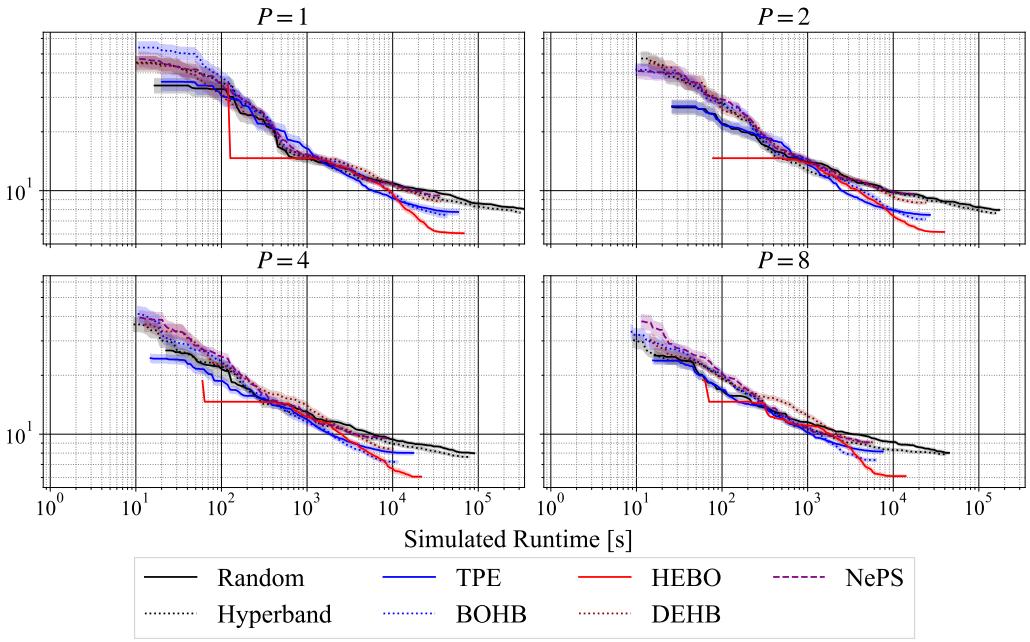


Figure 38.: The performance over time on OpenML ID 146212 from LCBench.

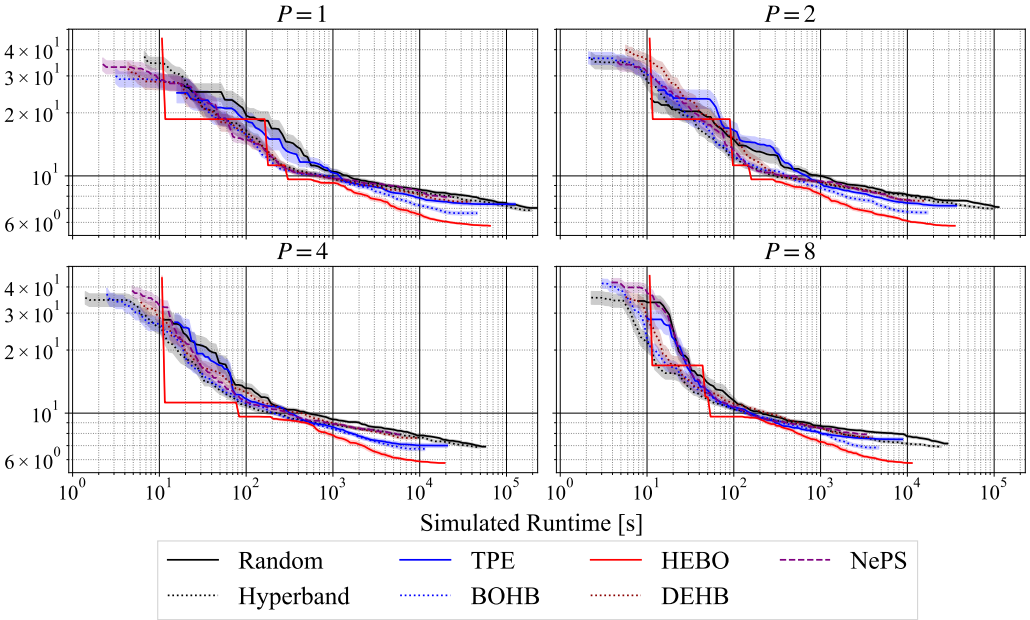


Figure 39.: The performance over time on OpenML ID 167104 from LCBench.

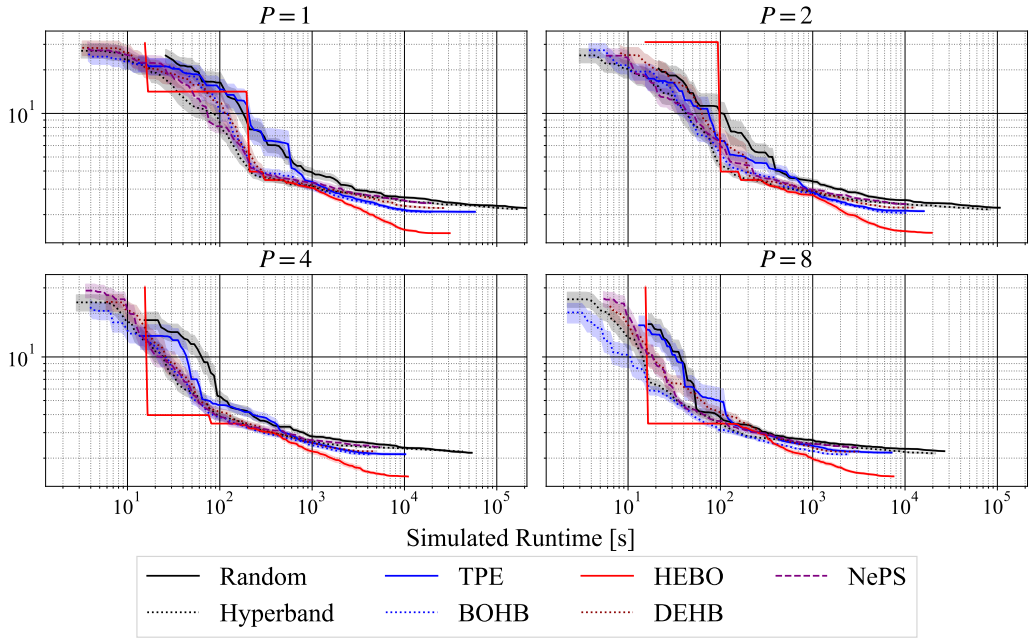


Figure 40.: The performance over time on OpenML ID 167149 from LCBench.

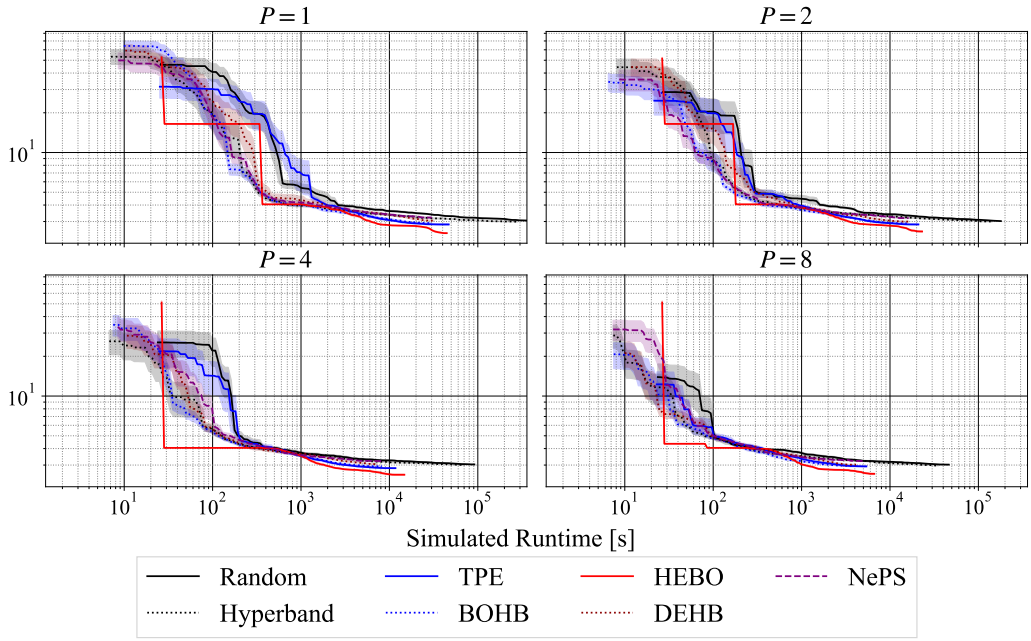


Figure 41.: The performance over time on OpenML ID 167152 from LCBench.

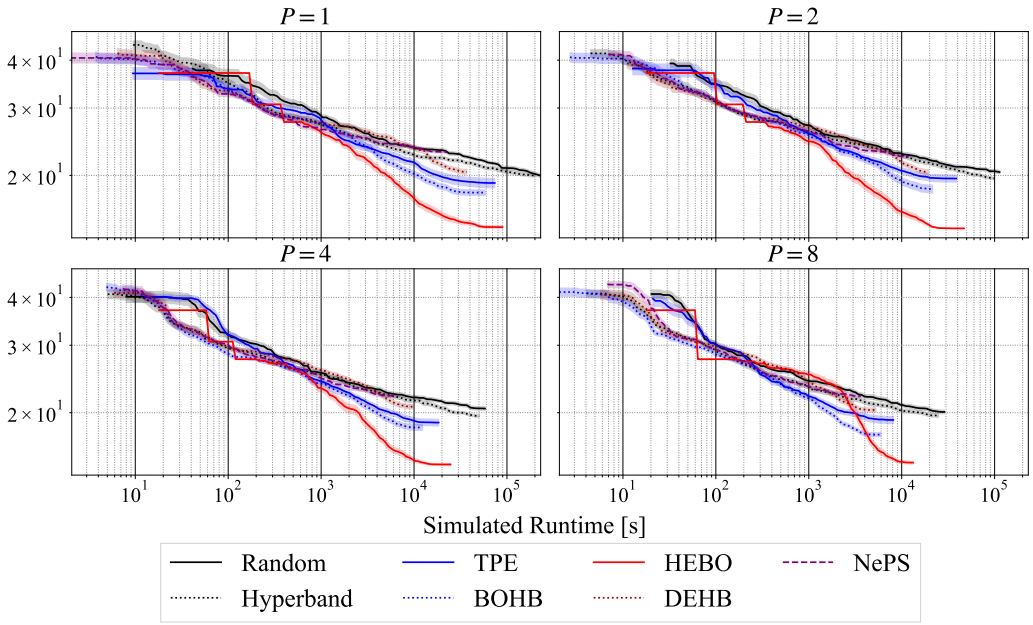


Figure 42.: The performance over time on OpenML ID 167161 from LCBench.

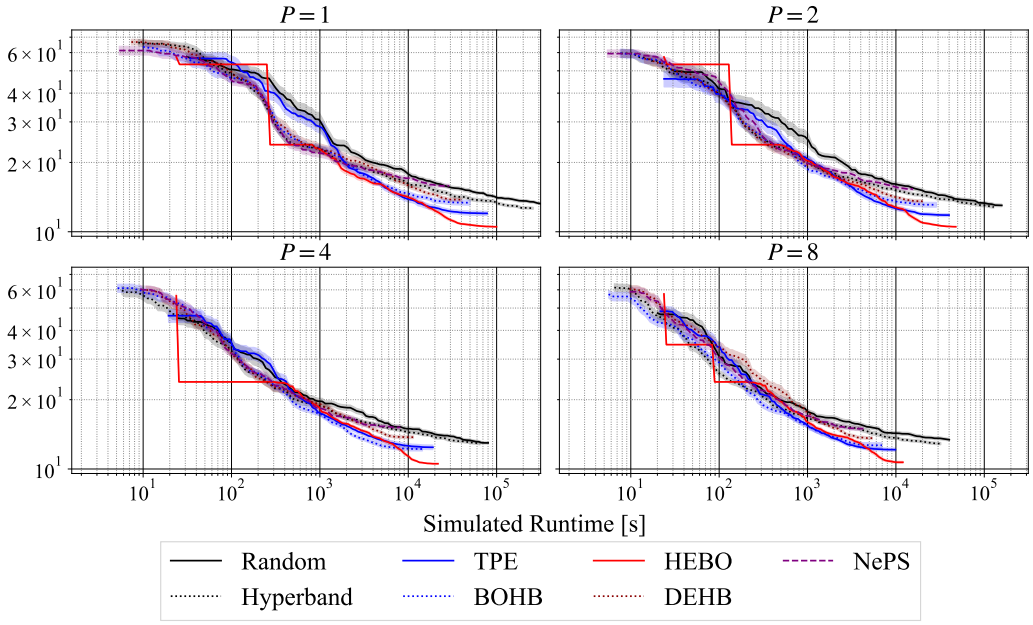


Figure 43.: The performance over time on OpenML ID 167168 from LCBench.

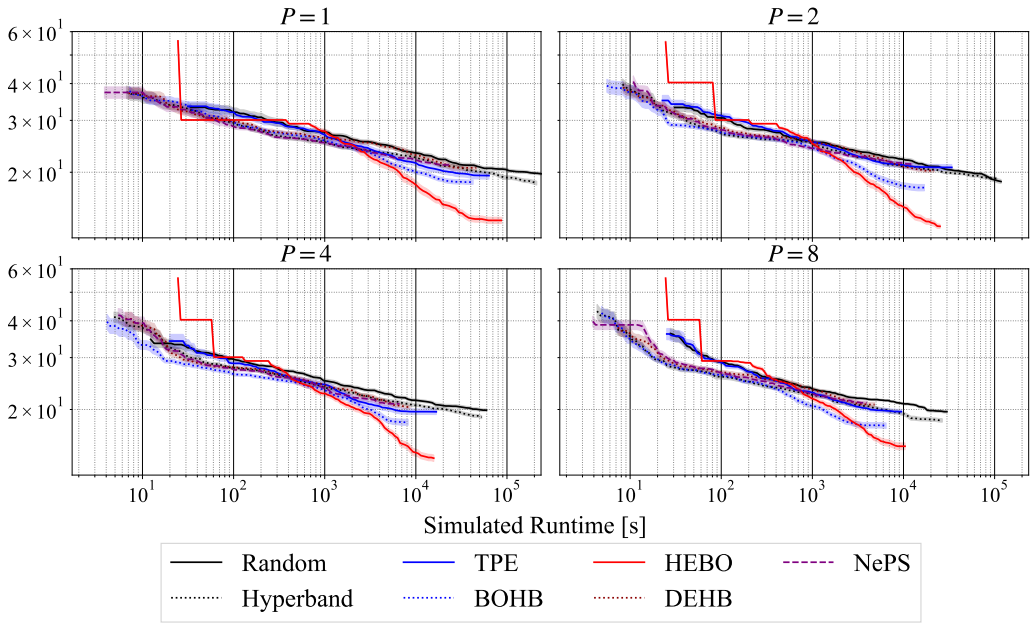


Figure 44.: The performance over time on OpenML ID 167181 from LCBench.

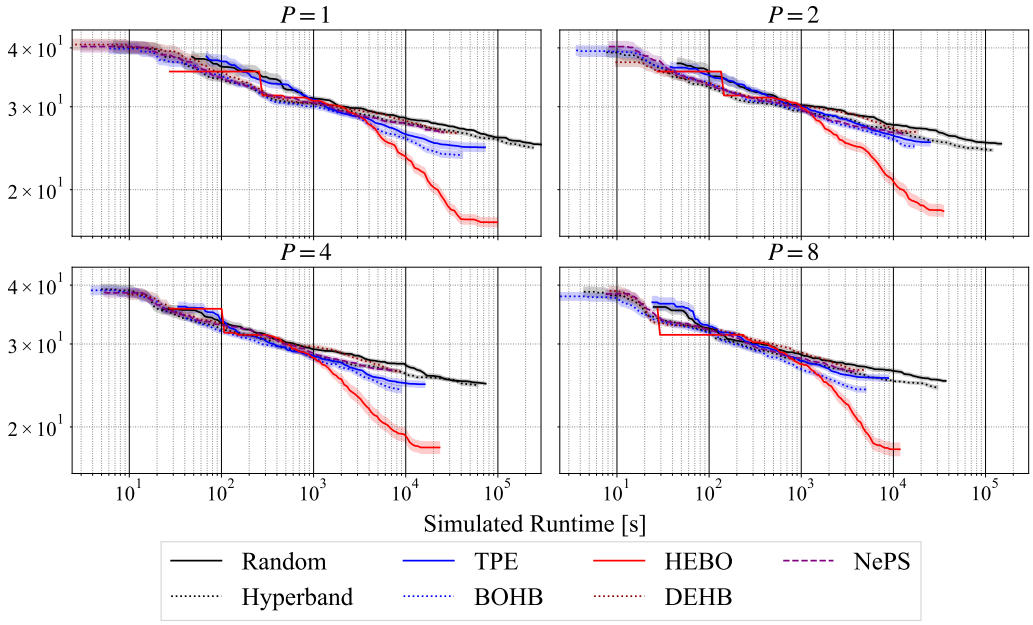


Figure 45.: The performance over time on OpenML ID 167184 from LCBench.

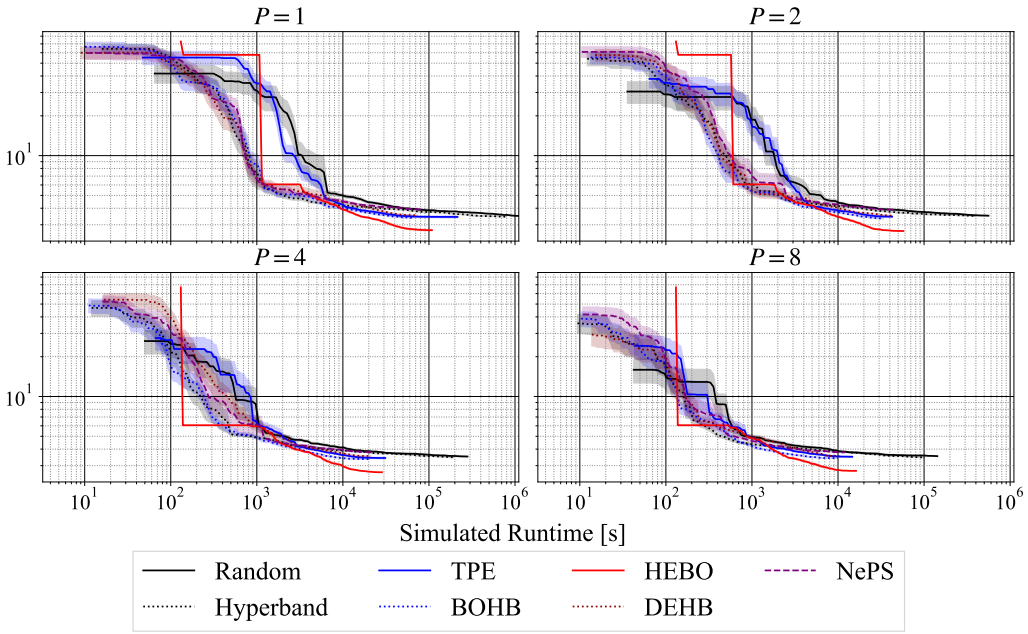


Figure 46.: The performance over time on OpenML ID 167185 from LCBench.

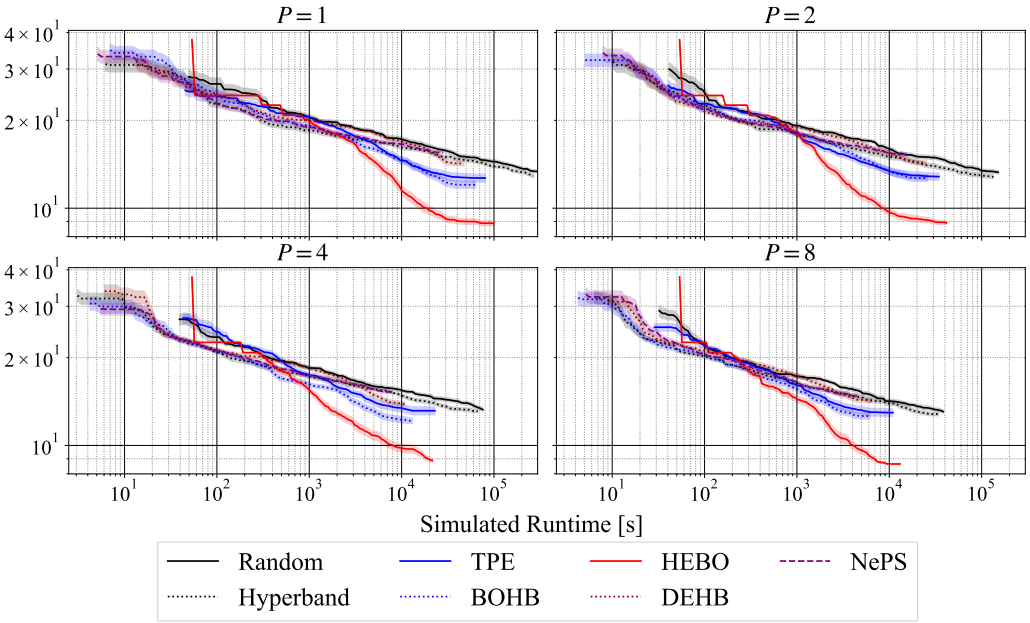


Figure 47.: The performance over time on OpenML ID 167190 from LCBench.

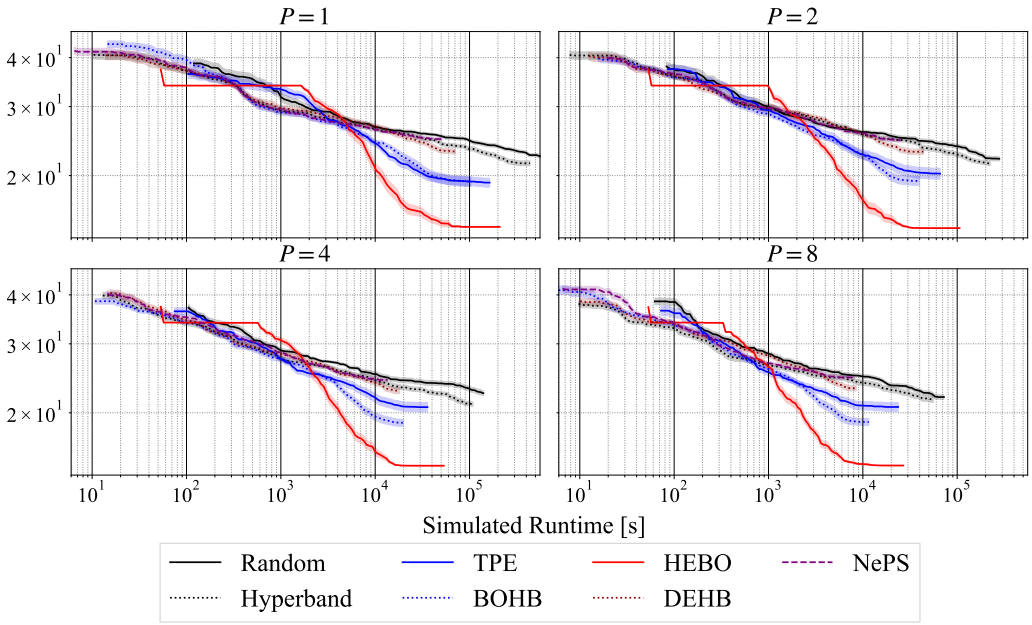


Figure 48.: The performance over time on OpenML ID 167200 from LCBench.

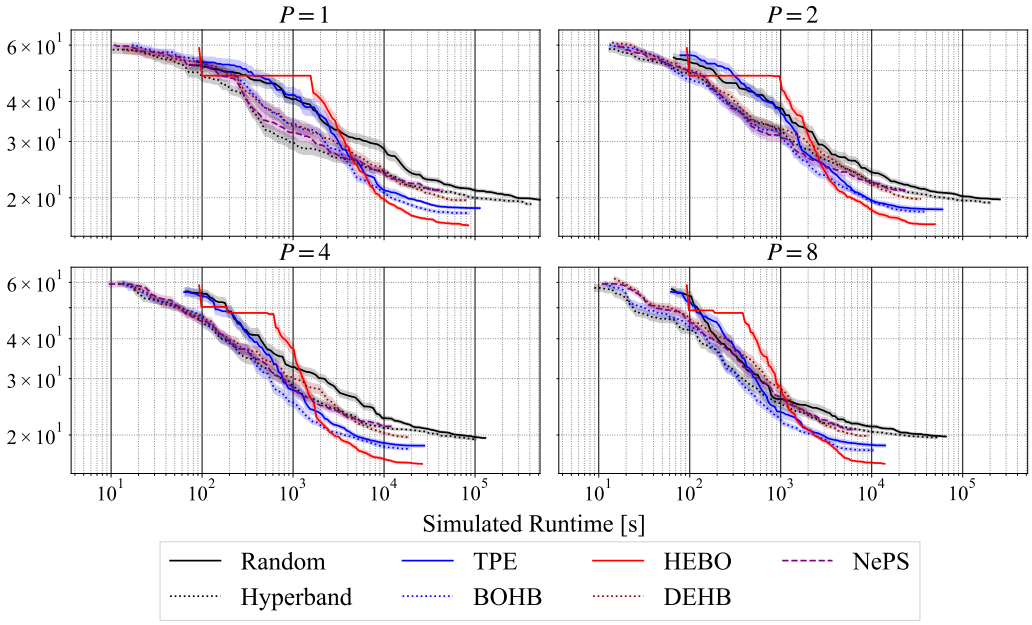


Figure 49.: The performance over time on OpenML ID 167201 from LCBench.

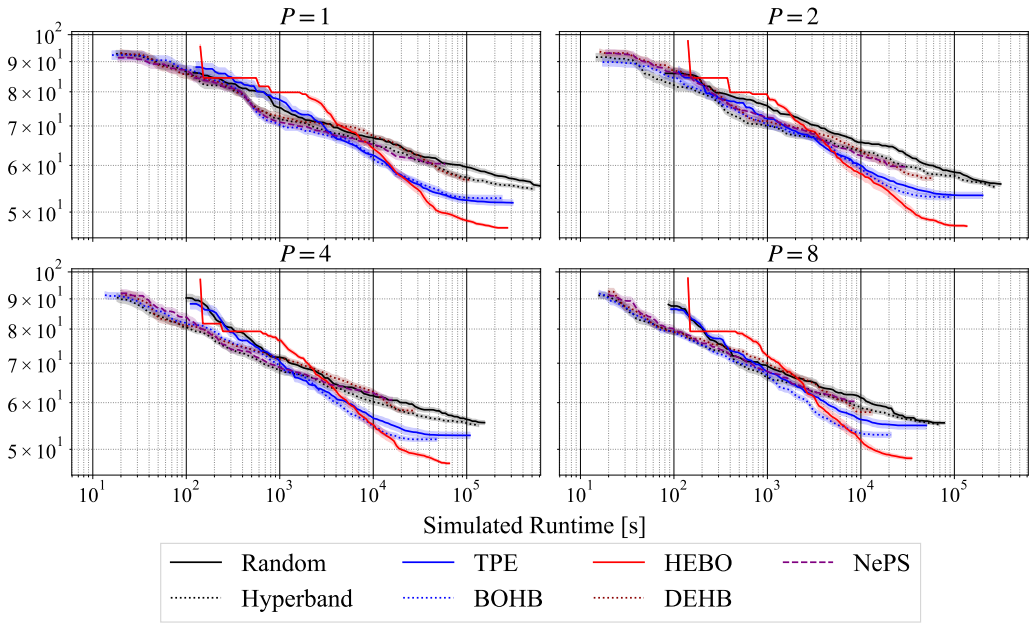


Figure 50.: The performance over time on OpenML ID 168329 from LCBench.

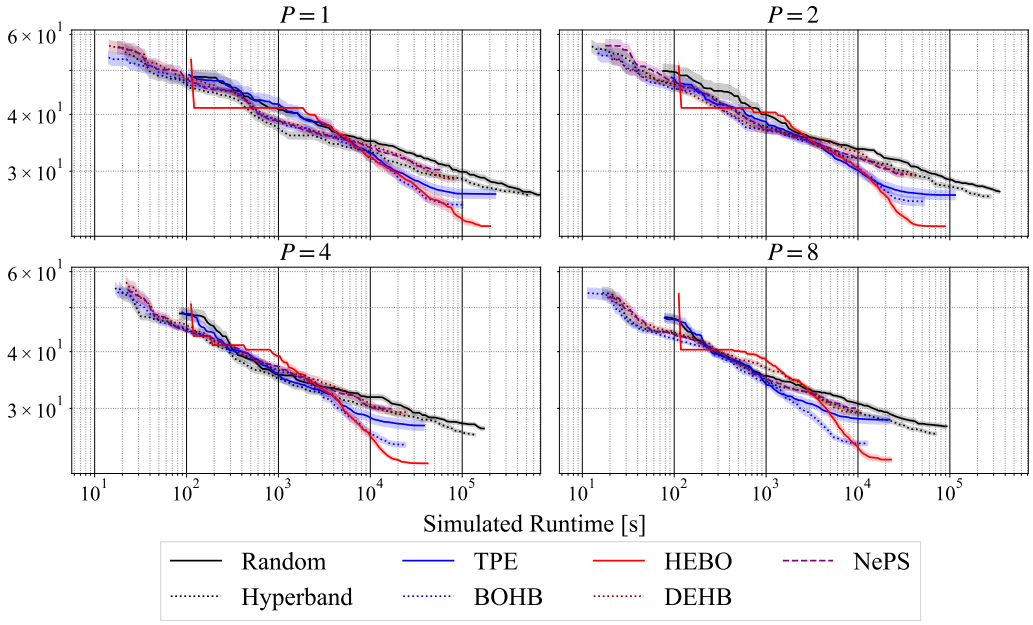


Figure 51.: The performance over time on OpenML ID 168330 from LCBench.

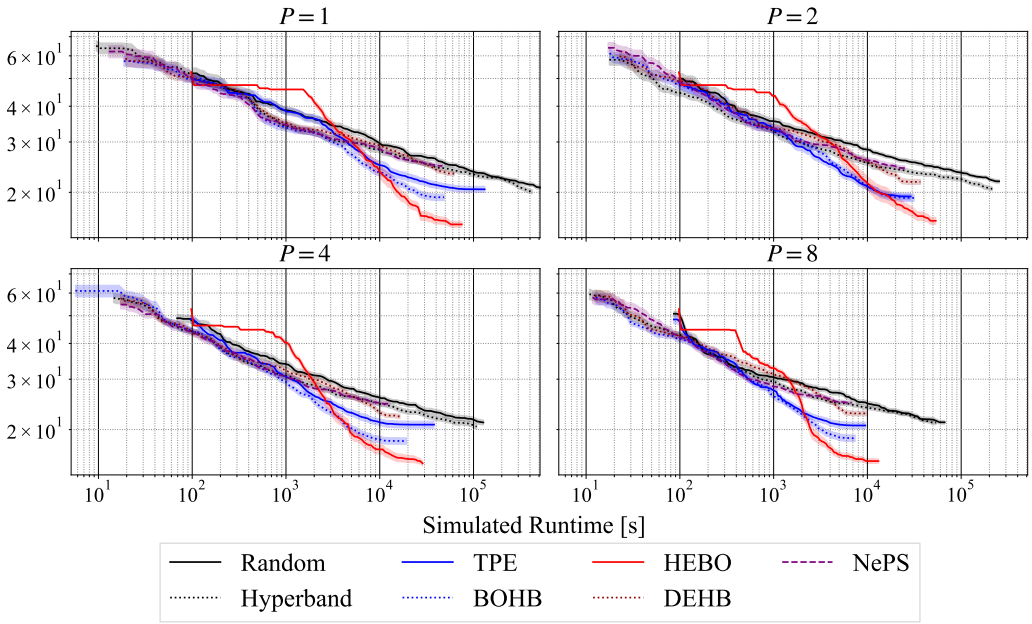


Figure 52.: The performance over time on OpenML ID 168331 from LCBench.

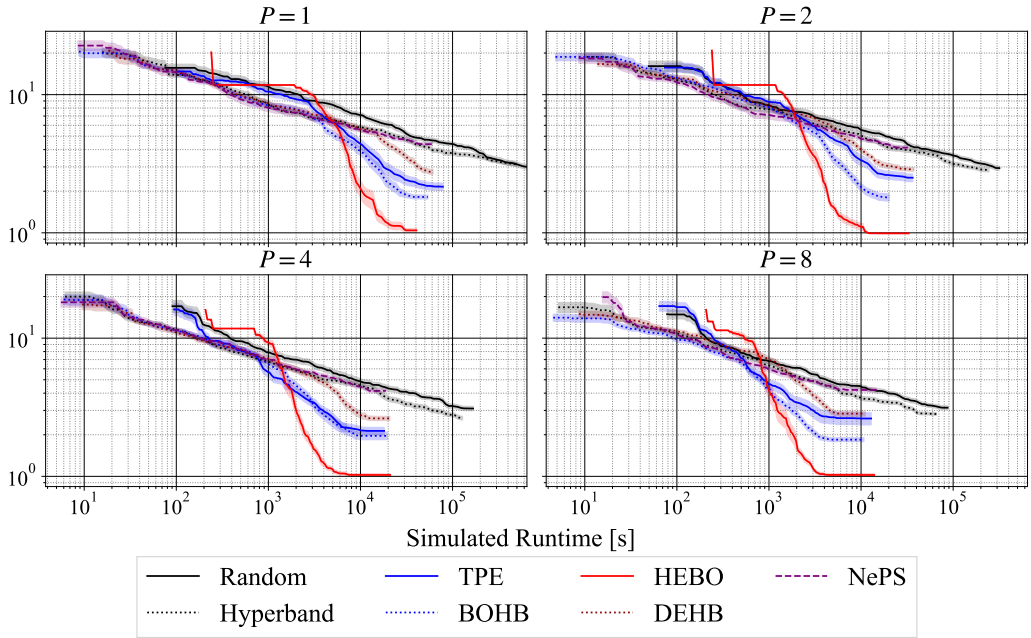


Figure 53.: The performance over time on OpenML ID 168335 from LCBench.

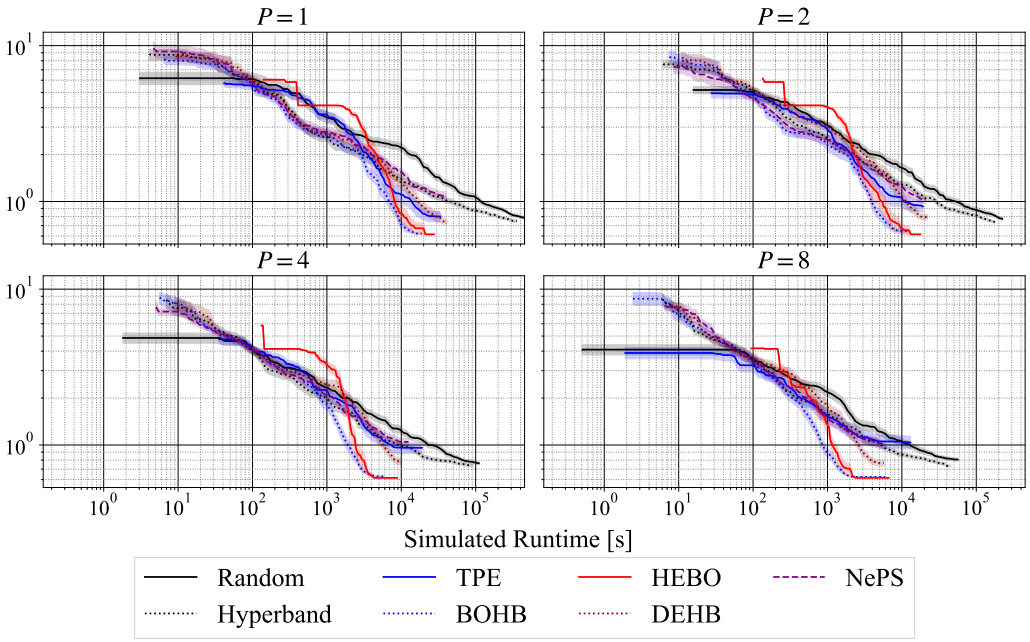


Figure 54.: The performance over time on OpenML ID 168868 from LCBench.

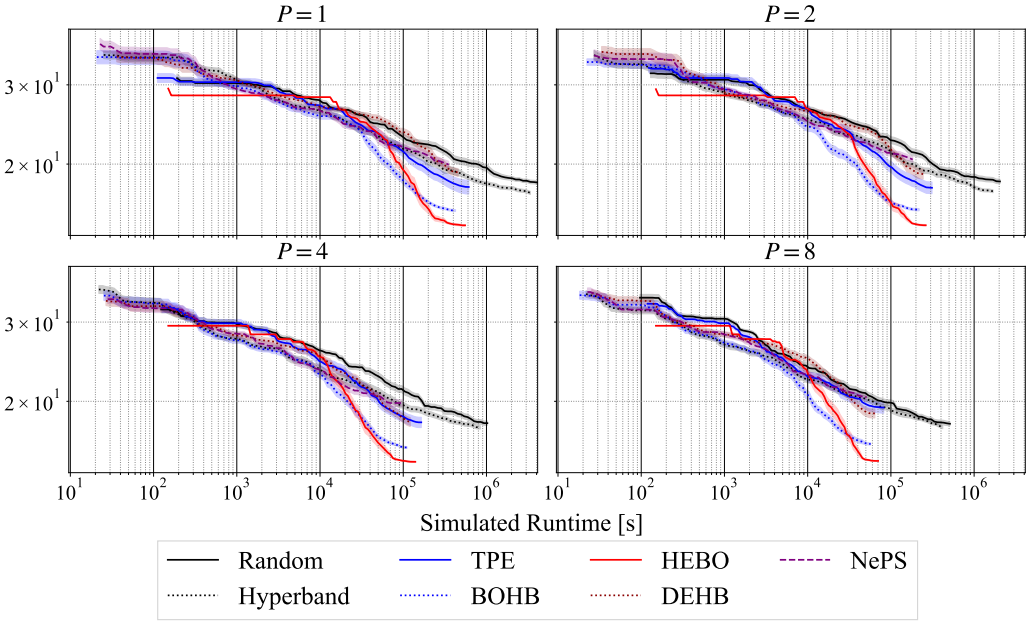


Figure 55.: The performance over time on OpenML ID 168908 from LCBench.

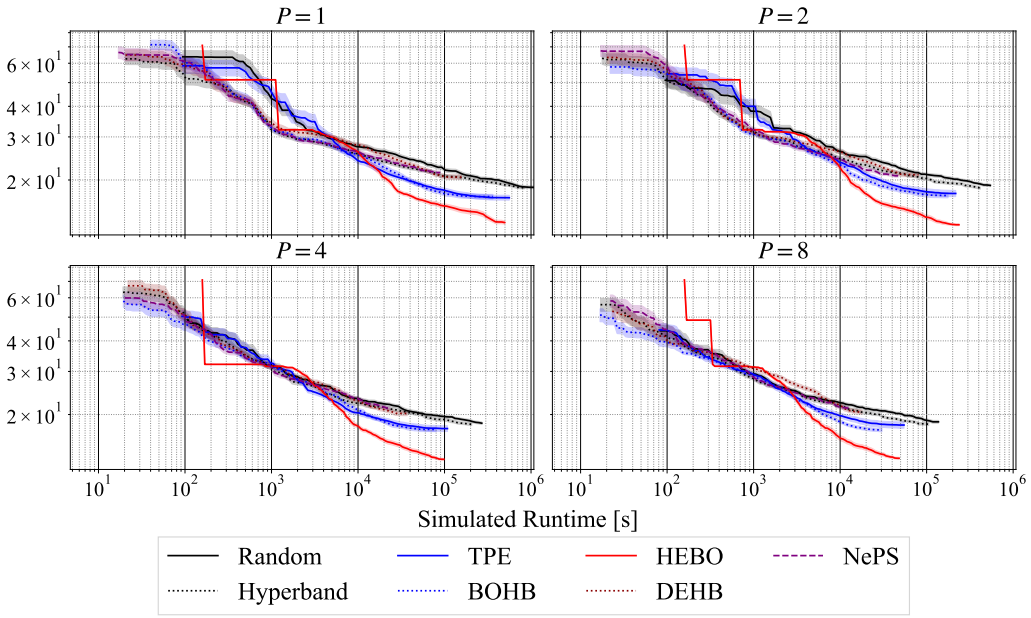


Figure 56.: The performance over time on OpenML ID 168910 from LCBench.

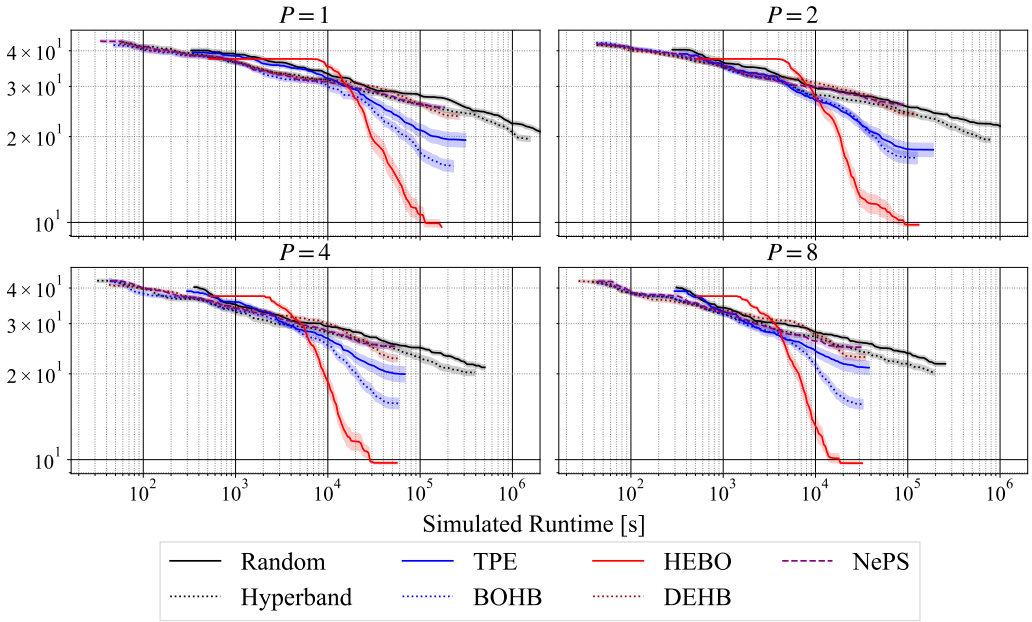


Figure 57.: The performance over time on OpenML ID 189354 from LCBench.

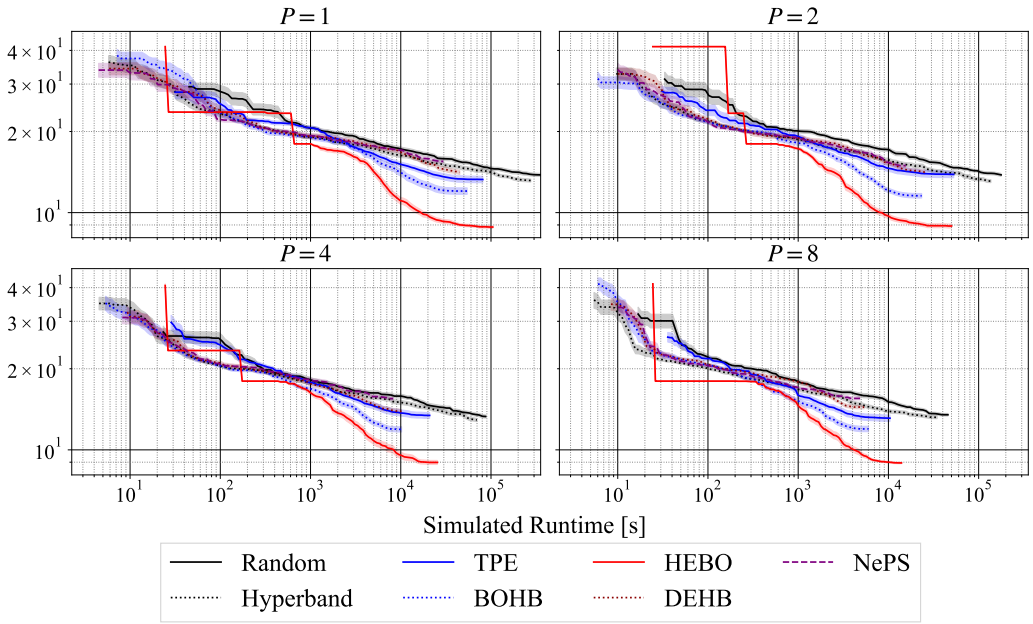


Figure 58.: The performance over time on OpenML ID 189862 from LCBench.

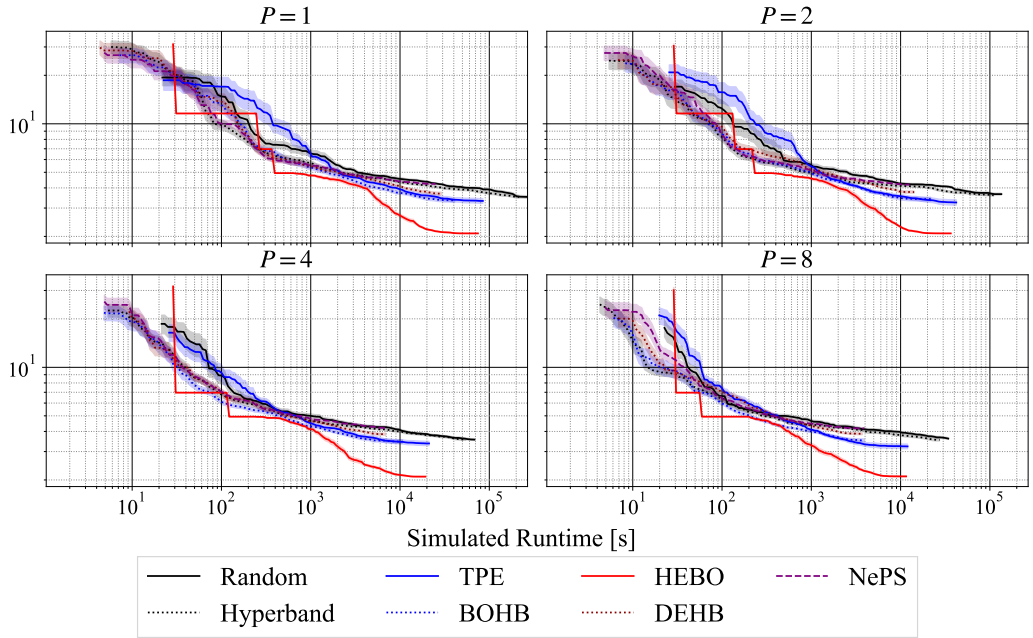


Figure 59.: The performance over time on OpenML ID 189865 from LCBench.

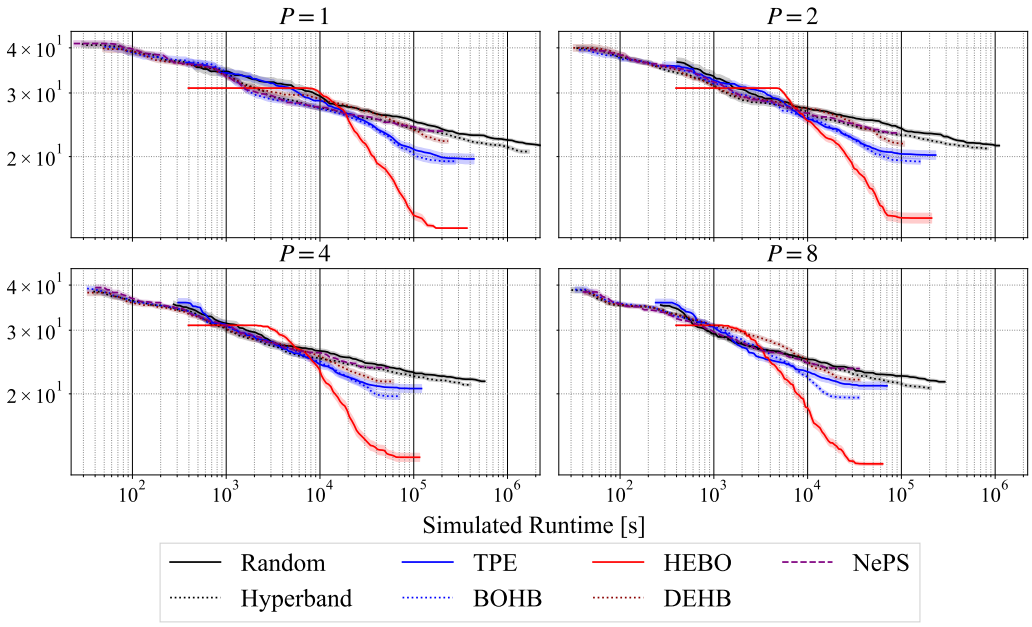


Figure 60.: The performance over time on OpenML ID 189866 from LCBench.

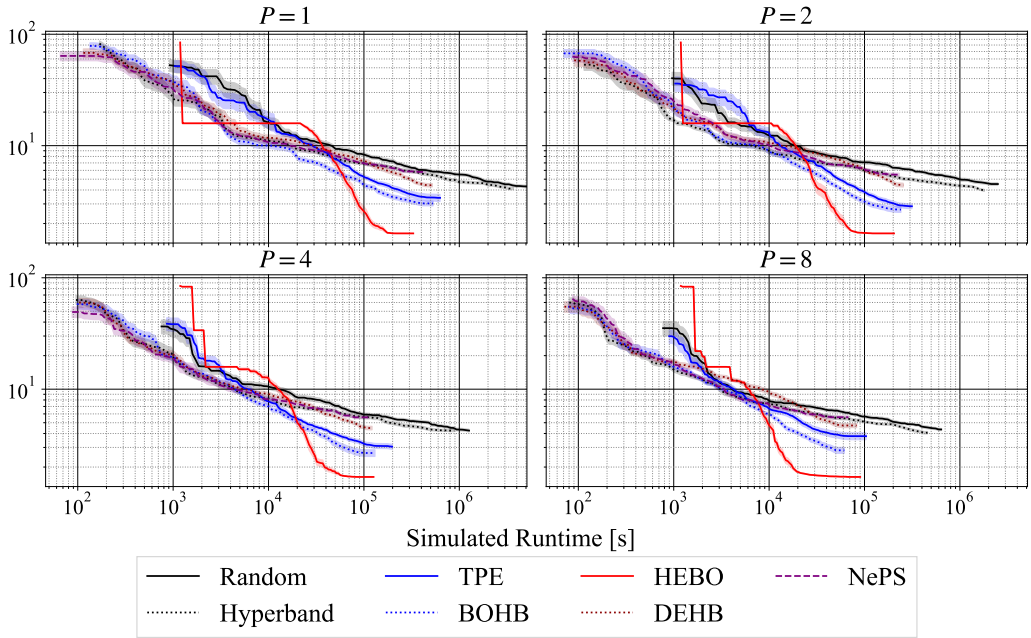


Figure 61.: The performance over time on OpenML ID 189873 from LCBench.

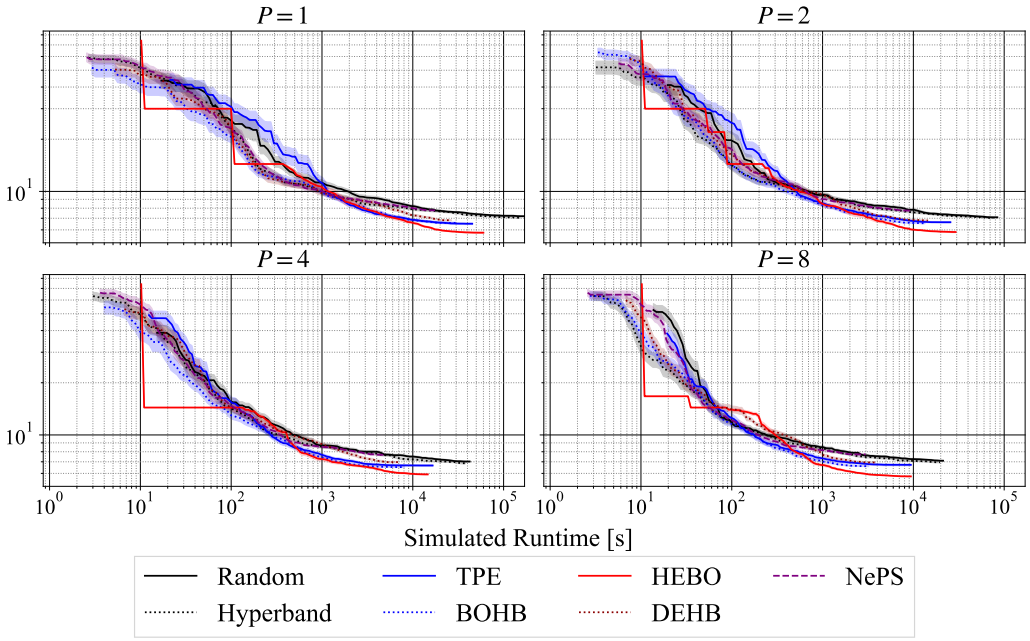


Figure 62.: The performance over time on OpenML ID 189905 from LCBench.

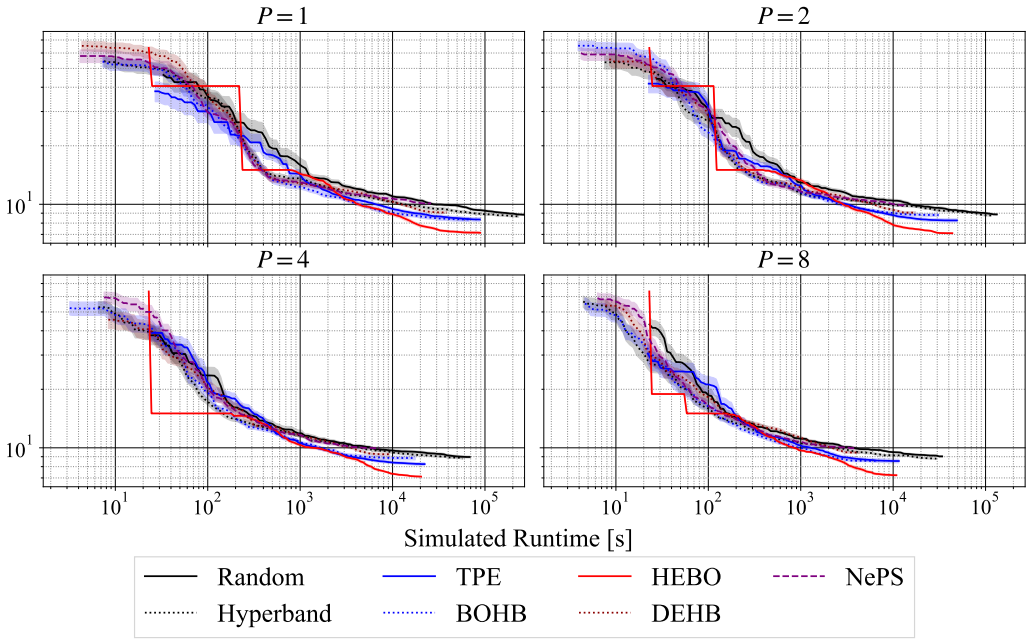


Figure 63.: The performance over time on OpenML ID 189906 from LCBench.

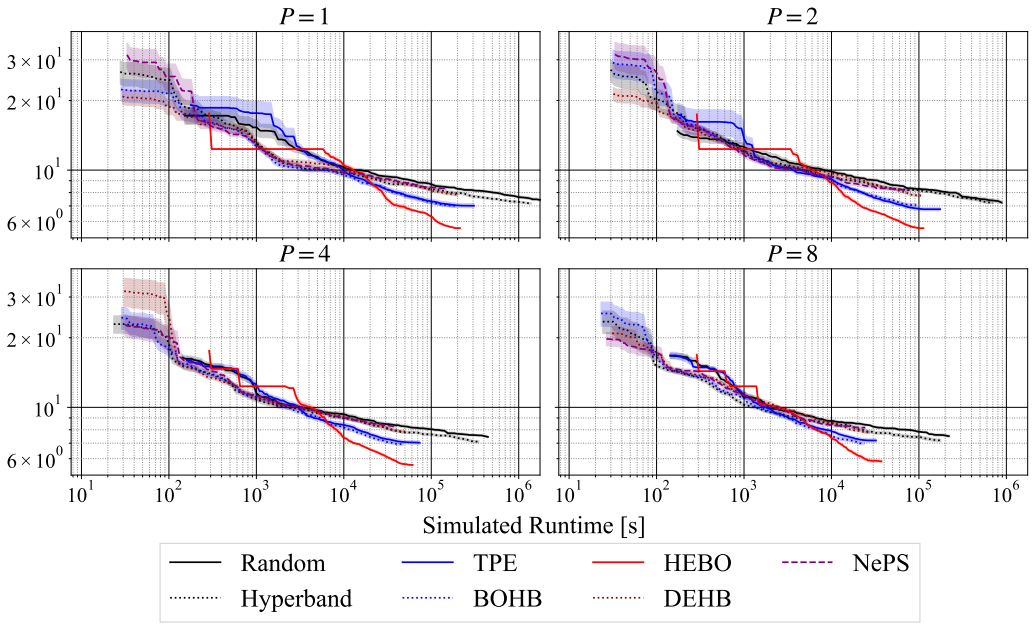


Figure 64.: The performance over time on OpenML ID 189908 from LCBench.

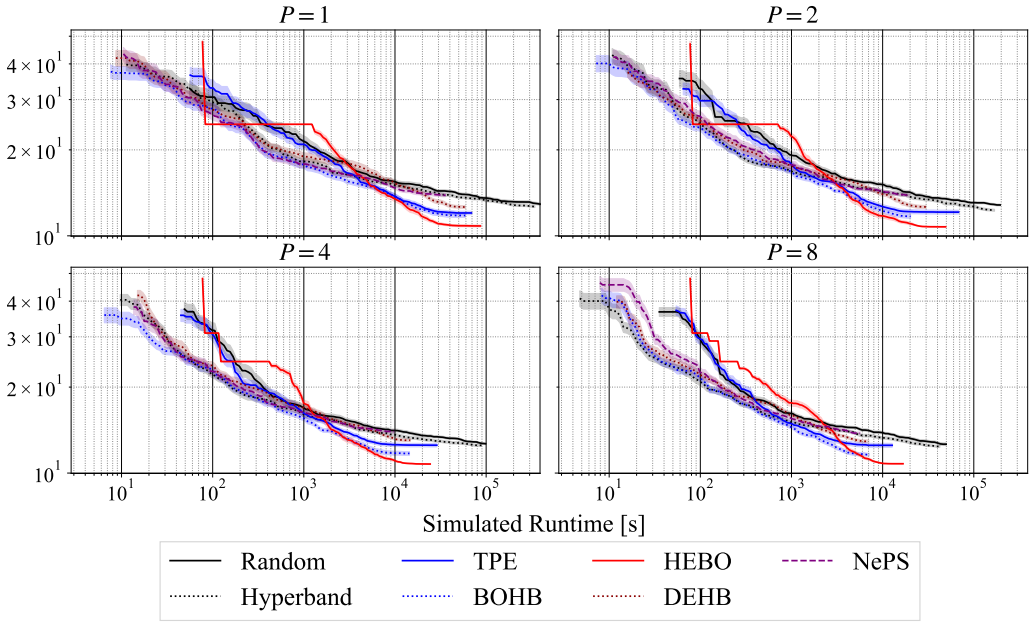


Figure 65.: The performance over time on OpenML ID 189909 from LCBench.

C.2. Actual & Simulated Runtimes for Each Setup

Tables 8–45 list the actual and simulated runtimes for each setup averaged over 30 random seeds. In each table, **Act.** is actual runtime, **Sim.** is simulated runtime, and $\times$ **Fast** is how many times the actual runtime was quicker compared to the simulated runtime.

Table 8.: Actual and simulated runtimes for Random on synthetic functions.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	1.3e+1	7.2e+6	5.5e+5	1.7e+1	3.6e+6	2.1e+5	2.1e+1	1.8e+6	8.5e+4	3.7e+1	9.0e+5	2.4e+4
2	1.4e+1	7.2e+6	5.0e+5	1.8e+1	3.6e+6	2.0e+5	2.1e+1	1.8e+6	8.5e+4	3.7e+1	9.0e+5	2.5e+4
3	1.4e+1	7.2e+6	5.1e+5	1.9e+1	3.6e+6	1.9e+5	2.2e+1	1.8e+6	8.2e+4	4.0e+1	9.0e+5	2.3e+4

Table 9.: Actual and simulated runtimes for HyperBand on synthetic functions.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	1.3e+2	5.2e+6	4.1e+4	1.6e+2	2.6e+6	1.6e+4	2.0e+2	1.3e+6	6.5e+3	3.0e+2	6.5e+5	2.2e+3
2	1.3e+2	5.5e+6	4.1e+4	1.7e+2	2.8e+6	1.6e+4	2.1e+2	1.4e+6	6.5e+3	3.1e+2	6.9e+5	2.2e+3
3	1.5e+2	5.7e+6	3.7e+4	2.0e+2	2.9e+6	1.5e+4	2.4e+2	1.4e+6	6.0e+3	3.4e+2	7.1e+5	2.1e+3

Table 10.: Actual and simulated runtimes for TPE on synthetic functions.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	2.2e+0	7.2e+5	3.2e+5	3.6e+0	3.6e+5	1.0e+5	4.7e+0	1.8e+5	3.9e+4	9.6e+0	9.4e+4	9.8e+3
2	3.4e+0	7.2e+5	2.1e+5	4.9e+0	3.6e+5	7.5e+4	6.3e+0	1.8e+5	2.9e+4	1.2e+1	9.4e+4	7.6e+3
3	5.5e+0	7.2e+5	1.3e+5	8.2e+0	3.6e+5	4.4e+4	1.1e+1	1.8e+5	1.7e+4	2.0e+1	9.4e+4	4.7e+3

Table 11.: Actual and simulated runtimes for BOHB on synthetic functions.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	1.7e+1	5.1e+5	3.0e+4	1.9e+1	2.6e+5	1.3e+4	2.5e+1	1.3e+5	5.1e+3	3.7e+1	6.8e+4	1.8e+3
2	2.0e+1	5.4e+5	2.6e+4	2.2e+1	2.7e+5	1.2e+4	2.9e+1	1.4e+5	4.7e+3	4.3e+1	7.2e+4	1.7e+3
3	2.7e+1	5.6e+5	2.1e+4	3.0e+1	2.8e+5	9.3e+3	3.9e+1	1.4e+5	3.6e+3	5.6e+1	7.4e+4	1.3e+3

Table 12.: Actual and simulated runtimes for HEBO on synthetic functions.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	2.3e+3	7.2e+5	3.2e+2	2.2e+3	3.6e+5	1.7e+2	2.2e+3	1.8e+5	8.1e+1	2.2e+3	9.0e+4	4.2e+1
2	3.0e+3	7.2e+5	2.4e+2	2.8e+3	3.6e+5	1.3e+2	2.9e+3	1.8e+5	6.3e+1	2.9e+3	9.1e+4	3.2e+1
3	4.0e+3	7.2e+5	1.8e+2	4.0e+3	3.6e+5	9.0e+1	3.8e+3	1.8e+5	4.8e+1	4.3e+3	9.1e+4	2.1e+1

Table 13.: Actual and simulated runtimes for DEHB on synthetic functions.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	3.1e+0	5.7e+5	1.8e+5	2.3e+1	2.8e+5	1.2e+4	2.5e+1	1.4e+5	5.8e+3	2.7e+1	7.5e+4	2.8e+3
2	3.3e+0	6.1e+5	1.8e+5	2.3e+1	3.1e+5	1.3e+4	2.5e+1	1.5e+5	6.2e+3	2.8e+1	8.1e+4	2.9e+3
3	3.1e+0	6.4e+5	2.1e+5	2.3e+1	3.2e+5	1.4e+4	2.5e+1	1.6e+5	6.5e+3	2.8e+1	8.4e+4	3.0e+3

Table 14.: Actual and simulated runtimes for NePS on synthetic functions.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	5.4e+2	5.1e+5	9.5e+2	5.5e+2	2.7e+5	4.9e+2	6.3e+2	1.4e+5	2.2e+2	6.8e+2	7.1e+4	1.0e+2
2	5.6e+2	5.4e+5	9.6e+2	5.9e+2	2.9e+5	4.9e+2	6.7e+2	1.5e+5	2.2e+2	7.4e+2	7.6e+4	1.0e+2
3	6.2e+2	5.6e+5	9.1e+2	6.4e+2	3.0e+5	4.6e+2	7.2e+2	1.5e+5	2.2e+2	9.3e+2	8.0e+4	8.6e+1

Table 15.: Actual and simulated runtimes for SMAC on synthetic functions.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	1.8e+1	5.0e+5	2.9e+4	3.8e+1	2.6e+5	6.8e+3	3.9e+1	1.4e+5	3.4e+3	4.2e+1	6.9e+4	1.7e+3
2	2.1e+1	5.3e+5	2.5e+4	4.3e+1	2.8e+5	6.5e+3	4.5e+1	1.5e+5	3.3e+3	4.8e+1	7.5e+4	1.6e+3
3	3.4e+1	5.5e+5	1.6e+4	5.9e+1	2.9e+5	5.0e+3	6.2e+1	1.5e+5	2.5e+3	6.8e+1	7.9e+4	1.2e+3

Table 16.: Actual and simulated runtimes for Random on HPOBench.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	1.3e+1	7.5e+4	5.6e+3	1.8e+1	4.0e+4	2.2e+3	2.1e+1	2.0e+4	9.5e+2	3.8e+1	1.1e+4	2.9e+2
2	1.3e+1	1.3e+5	9.6e+3	1.8e+1	6.3e+4	3.5e+3	2.2e+1	3.3e+4	1.6e+3	3.8e+1	1.8e+4	4.9e+2
3	1.3e+1	2.1e+5	1.6e+4	1.7e+1	1.0e+5	6.0e+3	2.1e+1	5.2e+4	2.4e+3	3.8e+1	3.1e+4	8.2e+2
4	1.3e+1	1.3e+5	9.7e+3	1.8e+1	6.3e+4	3.5e+3	2.2e+1	3.3e+4	1.5e+3	3.7e+1	1.9e+4	5.2e+2
5	1.3e+1	3.5e+5	2.6e+4	1.8e+1	1.8e+5	9.8e+3	2.1e+1	9.2e+4	4.3e+3	3.8e+1	4.9e+4	1.3e+3
6	1.3e+1	1.1e+6	8.2e+4	1.7e+1	5.6e+5	3.2e+4	2.1e+1	2.8e+5	1.3e+4	3.8e+1	1.6e+5	4.3e+3
7	1.3e+1	3.3e+5	2.4e+4	1.8e+1	1.7e+5	9.7e+3	2.2e+1	8.6e+4	4.0e+3	3.8e+1	5.1e+4	1.3e+3
8	1.3e+1	1.2e+5	8.9e+3	1.8e+1	5.9e+4	3.3e+3	2.2e+1	3.0e+4	1.4e+3	3.8e+1	1.6e+4	4.2e+2

Table 17.: Actual and simulated runtimes for HyperBand on HPOBench.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	1.4e+2	8.8e+4	6.4e+2	1.5e+2	4.5e+4	2.9e+2	1.9e+2	2.3e+4	1.2e+2	3.0e+2	1.2e+4	4.1e+1
2	1.4e+2	1.4e+5	9.9e+2	1.5e+2	7.0e+4	4.5e+2	1.9e+2	3.5e+4	1.8e+2	3.0e+2	1.9e+4	6.4e+1
3	1.4e+2	2.2e+5	1.6e+3	1.5e+2	1.1e+5	7.4e+2	1.9e+2	5.8e+4	3.0e+2	3.0e+2	3.1e+4	1.0e+2
4	1.4e+2	1.4e+5	1.0e+3	1.5e+2	7.1e+4	4.6e+2	1.9e+2	3.6e+4	1.9e+2	3.0e+2	1.9e+4	6.4e+1
5	1.3e+2	4.1e+5	3.1e+3	1.5e+2	2.0e+5	1.3e+3	1.9e+2	1.0e+5	5.2e+2	3.0e+2	5.6e+4	1.9e+2
6	1.4e+2	1.3e+6	9.7e+3	1.5e+2	6.8e+5	4.5e+3	1.9e+2	3.5e+5	1.8e+3	3.0e+2	1.7e+5	5.9e+2
7	1.3e+2	3.8e+5	2.9e+3	1.5e+2	1.9e+5	1.2e+3	1.9e+2	9.3e+4	4.8e+2	3.0e+2	5.1e+4	1.7e+2
8	1.4e+2	1.3e+5	9.9e+2	1.6e+2	6.7e+4	4.3e+2	1.9e+2	3.4e+4	1.8e+2	3.0e+2	1.8e+4	6.0e+1

Table 18.: Actual and simulated runtimes for TPE on HPOBench.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	5.6e+0	1.3e+4	2.3e+3	8.2e+0	7.4e+3	9.1e+2	1.1e+1	3.1e+3	2.9e+2	1.9e+1	2.8e+3	1.5e+2
2	5.5e+0	7.4e+4	1.3e+4	8.4e+0	2.3e+4	2.7e+3	1.1e+1	9.5e+3	8.8e+2	1.8e+1	6.6e+3	3.6e+2
3	5.7e+0	4.9e+4	8.7e+3	8.1e+0	1.0e+4	1.3e+3	1.1e+1	8.1e+3	7.5e+2	1.8e+1	7.7e+3	4.4e+2
4	5.6e+0	5.9e+3	1.0e+3	8.2e+0	5.2e+3	6.4e+2	1.1e+1	4.4e+3	4.2e+2	1.8e+1	2.9e+3	1.6e+2
5	5.5e+0	1.1e+5	2.1e+4	8.4e+0	5.7e+4	6.8e+3	1.1e+1	3.6e+4	3.4e+3	1.9e+1	2.0e+4	1.0e+3
6	5.5e+0	7.2e+5	1.3e+5	8.5e+0	2.6e+5	3.0e+4	1.1e+1	2.2e+5	2.1e+4	1.9e+1	1.0e+5	5.4e+3
7	5.5e+0	7.7e+4	1.4e+4	8.2e+0	5.8e+4	7.1e+3	1.1e+1	3.3e+4	3.0e+3	1.8e+1	1.7e+4	9.4e+2
8	5.6e+0	2.2e+4	3.9e+3	8.3e+0	2.4e+4	2.8e+3	1.1e+1	6.4e+3	5.8e+2	1.9e+1	5.8e+3	3.1e+2

Table 19.: Actual and simulated runtimes for BOHB on HPOBench.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	2.4e+1	1.1e+4	4.7e+2	2.7e+1	8.0e+3	3.0e+2	3.5e+1	2.8e+3	8.1e+1	5.1e+1	2.0e+3	3.9e+1
2	2.5e+1	4.5e+4	1.8e+3	2.7e+1	2.2e+4	8.2e+2	3.6e+1	1.0e+4	2.8e+2	5.1e+1	4.4e+3	8.6e+1
3	2.5e+1	5.7e+4	2.3e+3	2.7e+1	2.1e+4	8.0e+2	3.6e+1	9.2e+3	2.6e+2	5.1e+1	4.9e+3	9.7e+1
4	2.5e+1	1.6e+4	6.4e+2	2.7e+1	7.0e+3	2.6e+2	3.5e+1	4.2e+3	1.2e+2	5.2e+1	2.1e+3	4.0e+1
5	2.5e+1	1.4e+5	5.7e+3	2.7e+1	6.7e+4	2.5e+3	3.6e+1	2.7e+4	7.4e+2	5.1e+1	1.1e+4	2.2e+2
6	2.5e+1	5.0e+5	2.0e+4	2.7e+1	2.0e+5	7.2e+3	3.5e+1	1.0e+5	2.9e+3	5.2e+1	3.5e+4	6.8e+2
7	2.4e+1	1.6e+5	6.3e+3	2.7e+1	6.3e+4	2.3e+3	3.6e+1	2.4e+4	6.8e+2	5.2e+1	1.3e+4	2.4e+2
8	2.5e+1	3.9e+4	1.6e+3	2.7e+1	1.2e+4	4.4e+2	3.6e+1	1.0e+4	2.8e+2	5.1e+1	3.3e+3	6.4e+1

Table 20.: Actual and simulated runtimes for HEBO on HPOBench.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	3.4e+3	4.0e+4	1.2e+1	3.5e+3	1.7e+4	4.7e+0	3.6e+3	1.0e+4	2.9e+0	3.5e+3	5.2e+3	1.5e+0
2	3.5e+3	1.2e+5	3.5e+1	3.4e+3	5.1e+4	1.5e+1	3.5e+3	2.8e+4	8.1e+0	3.5e+3	1.5e+4	4.2e+0
3	3.5e+3	5.8e+4	1.7e+1	3.5e+3	3.1e+4	8.9e+0	3.8e+3	1.8e+4	4.9e+0	3.5e+3	1.0e+4	2.8e+0
4	3.4e+3	2.8e+4	8.3e+0	3.5e+3	1.7e+4	4.8e+0	3.5e+3	8.4e+3	2.4e+0	3.7e+3	4.6e+3	1.3e+0
5	3.3e+3	2.1e+5	6.5e+1	3.3e+3	1.3e+5	3.9e+1	3.2e+3	5.0e+4	1.5e+1	3.3e+3	3.3e+4	1.0e+1
6	3.1e+3	6.3e+5	2.0e+2	3.2e+3	4.4e+5	1.4e+2	3.2e+3	2.0e+5	6.4e+1	3.2e+3	9.7e+4	3.1e+1
7	3.3e+3	2.0e+5	6.0e+1	3.2e+3	9.8e+4	3.1e+1	3.1e+3	5.7e+4	1.8e+1	3.4e+3	2.3e+4	6.9e+0
8	3.4e+3	6.2e+4	1.8e+1	3.4e+3	2.8e+4	8.3e+0	3.5e+3	1.3e+4	3.7e+0	3.5e+3	7.6e+3	2.2e+0

Table 21.: Actual and simulated runtimes for DEHB on HPOBench.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	3.1e+0	6.1e+3	2.0e+3	2.4e+1	3.2e+3	1.4e+2	2.6e+1	1.8e+3	6.7e+1	2.9e+1	1.1e+3	3.9e+1
2	3.1e+0	1.4e+4	4.5e+3	2.4e+1	5.9e+3	2.4e+2	2.6e+1	2.9e+3	1.1e+2	3.0e+1	1.6e+3	5.2e+1
3	3.1e+0	1.6e+4	5.1e+3	2.4e+1	9.5e+3	3.9e+2	2.6e+1	5.7e+3	2.1e+2	3.0e+1	3.3e+3	1.1e+2
4	3.1e+0	8.1e+3	2.6e+3	2.4e+1	5.9e+3	2.4e+2	2.6e+1	2.3e+3	8.7e+1	2.9e+1	1.3e+3	4.5e+1
5	3.1e+0	4.9e+4	1.6e+4	2.4e+1	2.8e+4	1.2e+3	2.6e+1	1.3e+4	4.9e+2	2.9e+1	6.3e+3	2.2e+2
6	3.1e+0	1.9e+5	6.1e+4	2.4e+1	9.7e+4	4.0e+3	2.6e+1	4.6e+4	1.7e+3	3.0e+1	2.5e+4	8.5e+2
7	3.1e+0	3.2e+4	1.0e+4	2.4e+1	2.4e+4	1.0e+3	2.6e+1	9.4e+3	3.6e+2	2.9e+1	5.2e+3	1.8e+2
8	3.1e+0	1.5e+4	4.9e+3	2.4e+1	6.5e+3	2.7e+2	2.7e+1	3.1e+3	1.2e+2	2.9e+1	1.3e+3	4.4e+1

Table 22.: Actual and simulated runtimes for NePS on HPOBench.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	6.1e+2	9.3e+3	1.5e+1	7.9e+2	4.9e+3	6.3e+0	1.3e+3	3.0e+3	2.3e+0	1.6e+3	1.9e+3	1.2e+0
2	6.1e+2	1.4e+4	2.3e+1	7.3e+2	7.1e+3	9.8e+0	1.0e+3	4.0e+3	3.8e+0	1.9e+3	2.5e+3	1.3e+0
3	6.1e+2	2.2e+4	3.6e+1	7.3e+2	1.3e+4	1.8e+1	1.1e+3	6.9e+3	6.2e+0	1.8e+3	3.7e+3	2.0e+0
4	6.1e+2	1.4e+4	2.3e+1	7.8e+2	7.2e+3	9.3e+0	1.2e+3	4.5e+3	3.8e+0	1.9e+3	2.5e+3	1.3e+0
5	6.1e+2	4.1e+4	6.7e+1	6.8e+2	2.1e+4	3.1e+1	9.2e+2	1.2e+4	1.3e+1	1.8e+3	6.4e+3	3.6e+0
6	6.1e+2	1.3e+5	2.1e+2	6.4e+2	7.5e+4	1.2e+2	7.5e+2	3.8e+4	5.0e+1	1.2e+3	1.8e+4	1.6e+1
7	6.1e+2	3.8e+4	6.2e+1	6.8e+2	2.0e+4	3.0e+1	1.0e+3	9.4e+3	9.4e+0	1.8e+3	6.2e+3	3.4e+0
8	6.1e+2	1.4e+4	2.3e+1	7.4e+2	7.9e+3	1.1e+1	1.1e+3	4.0e+3	3.6e+0	1.8e+3	2.4e+3	1.4e+0

Table 23.: Actual and simulated runtimes for SMAC on HPOBench.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	3.1e+1	1.1e+4	3.6e+2	5.5e+1	6.1e+3	1.1e+2	5.9e+1	3.0e+3	5.1e+1	6.6e+1	2.4e+3	3.6e+1
2	3.1e+1	1.8e+4	5.8e+2	5.5e+1	1.2e+4	2.2e+2	5.9e+1	5.7e+3	9.6e+1	6.5e+1	3.0e+3	4.6e+1
3	3.4e+1	2.8e+4	8.4e+2	5.7e+1	1.5e+4	2.6e+2	6.0e+1	1.1e+4	1.9e+2	6.6e+1	4.1e+3	6.1e+1
4	3.2e+1	1.5e+4	4.7e+2	5.5e+1	9.0e+3	1.6e+2	5.9e+1	4.8e+3	8.1e+1	6.6e+1	3.6e+3	5.4e+1
5	3.2e+1	8.3e+4	2.6e+3	5.6e+1	4.7e+4	8.5e+2	6.0e+1	2.3e+4	3.8e+2	6.5e+1	1.1e+4	1.7e+2
6	3.2e+1	2.4e+5	7.6e+3	5.6e+1	1.2e+5	2.2e+3	5.9e+1	7.2e+4	1.2e+3	6.6e+1	3.8e+4	5.8e+2
7	3.2e+1	5.2e+4	1.6e+3	5.6e+1	4.6e+4	8.1e+2	5.9e+1	1.9e+4	3.2e+2	6.6e+1	1.0e+4	1.6e+2
8	3.2e+1	1.8e+4	5.6e+2	5.6e+1	1.2e+4	2.2e+2	5.9e+1	5.4e+3	9.3e+1	6.6e+1	3.1e+3	4.6e+1

Table 24.: Actual and simulated runtimes for Random on HPOlib.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	1.4e+1	1.1e+6	7.7e+4	2.1e+1	5.4e+5	2.6e+4	2.5e+1	2.7e+5	1.1e+4	4.2e+1	1.4e+5	3.2e+3
2	1.4e+1	5.6e+5	4.0e+4	2.1e+1	2.8e+5	1.4e+4	2.4e+1	1.4e+5	5.8e+3	4.3e+1	7.1e+4	1.6e+3
3	1.4e+1	1.6e+5	1.1e+4	2.1e+1	8.1e+4	3.9e+3	2.6e+1	4.1e+4	1.6e+3	4.2e+1	2.1e+4	4.9e+2
4	1.4e+1	8.3e+4	5.9e+3	2.1e+1	4.1e+4	2.0e+3	2.5e+1	2.1e+4	8.3e+2	4.3e+1	1.0e+4	2.4e+2

Table 25.: Actual and simulated runtimes for HyperBand on HPOlib.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	1.5e+2	8.2e+5	5.5e+3	1.9e+2	4.2e+5	2.2e+3	2.3e+2	2.1e+5	8.8e+2	3.4e+2	1.1e+5	3.1e+2
2	1.5e+2	4.4e+5	2.9e+3	1.9e+2	2.2e+5	1.2e+3	2.4e+2	1.1e+5	4.7e+2	3.4e+2	5.5e+4	1.6e+2
3	1.6e+2	1.3e+5	8.1e+2	1.9e+2	6.4e+4	3.3e+2	2.4e+2	3.2e+4	1.4e+2	3.4e+2	1.6e+4	4.7e+1
4	1.6e+2	6.6e+4	4.2e+2	1.9e+2	3.3e+4	1.7e+2	2.4e+2	1.6e+4	7.0e+1	3.4e+2	8.3e+3	2.4e+1

Table 26.: Actual and simulated runtimes for TPE on HPOlib.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	6.8e+0	1.6e+5	2.3e+4	1.0e+1	8.2e+4	7.9e+3	1.3e+1	3.4e+4	2.6e+3	2.2e+1	1.9e+4	8.7e+2
2	6.7e+0	7.0e+4	1.0e+4	1.1e+1	3.5e+4	3.3e+3	1.3e+1	1.8e+4	1.4e+3	2.2e+1	9.0e+3	4.1e+2
3	6.7e+0	2.8e+4	4.1e+3	1.0e+1	1.3e+4	1.2e+3	1.4e+1	5.4e+3	4.0e+2	2.2e+1	2.7e+3	1.2e+2
4	6.7e+0	1.0e+4	1.6e+3	1.1e+1	6.1e+3	5.8e+2	1.3e+1	2.9e+3	2.1e+2	2.3e+1	1.4e+3	5.9e+1

Table 27.: Actual and simulated runtimes for BOHB on HPOlib.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	3.0e+1	1.1e+5	3.9e+3	3.5e+1	5.8e+4	1.7e+3	4.7e+1	2.9e+4	6.1e+2	7.5e+1	1.4e+4	1.9e+2
2	3.0e+1	5.1e+4	1.7e+3	3.4e+1	2.7e+4	7.8e+2	4.7e+1	1.3e+4	2.7e+2	7.4e+1	7.1e+3	9.5e+1
3	3.0e+1	1.9e+4	6.2e+2	3.5e+1	9.6e+3	2.8e+2	4.8e+1	4.1e+3	8.6e+1	7.5e+1	2.2e+3	2.9e+1
4	3.0e+1	1.0e+4	3.4e+2	3.4e+1	4.5e+3	1.3e+2	4.8e+1	2.4e+3	5.0e+1	7.5e+1	9.8e+2	1.3e+1

Table 28.: Actual and simulated runtimes for HEBO on HPOlib.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	6.7e+3	1.6e+5	2.3e+1	6.7e+3	8.0e+4	1.2e+1	6.9e+3	4.1e+4	6.0e+0	7.1e+3	2.2e+4	3.0e+0
2	8.2e+3	7.2e+4	8.8e+0	8.0e+3	3.7e+4	4.7e+0	8.2e+3	2.0e+4	2.4e+0	8.1e+3	1.2e+4	1.5e+0
3	6.3e+3	3.0e+4	4.8e+0	6.3e+3	1.6e+4	2.5e+0	6.8e+3	9.8e+3	1.4e+0	6.6e+3	7.1e+3	1.1e+0
4	5.7e+3	1.8e+4	3.2e+0	6.1e+3	1.0e+4	1.7e+0	6.1e+3	7.0e+3	1.1e+0	6.2e+3	6.3e+3	1.0e+0

Table 29.: Actual and simulated runtimes for DEHB on HPOlib.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	3.2e+0	8.4e+4	2.7e+4	2.5e+1	4.3e+4	1.7e+3	2.7e+1	2.1e+4	7.7e+2	3.0e+1	1.1e+4	3.7e+2
2	3.2e+0	5.2e+4	1.6e+4	2.5e+1	2.5e+4	9.9e+2	2.8e+1	1.2e+4	4.5e+2	3.1e+1	6.2e+3	2.0e+2
3	3.2e+0	1.3e+4	4.2e+3	2.5e+1	6.8e+3	2.7e+2	2.8e+1	3.3e+3	1.2e+2	3.0e+1	1.7e+3	5.6e+1
4	3.2e+0	6.6e+3	2.1e+3	2.5e+1	3.4e+3	1.4e+2	2.8e+1	1.6e+3	5.9e+1	3.0e+1	8.5e+2	2.8e+1

Table 30.: Actual and simulated runtimes for NePS on HPOlib.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	7.1e+2	8.1e+4	1.1e+2	7.9e+2	4.3e+4	5.4e+1	9.6e+2	2.2e+4	2.2e+1	1.2e+3	1.1e+4	9.4e+0
2	7.1e+2	4.3e+4	6.1e+1	8.3e+2	2.3e+4	2.8e+1	1.1e+3	1.2e+4	1.1e+1	1.5e+3	6.3e+3	4.1e+0
3	7.1e+2	1.3e+4	1.8e+1	1.1e+3	7.1e+3	6.6e+0	1.6e+3	3.9e+3	2.5e+0	1.8e+3	2.3e+3	1.3e+0
4	7.1e+2	7.1e+3	9.9e+0	1.2e+3	4.1e+3	3.3e+0	1.7e+3	2.5e+3	1.5e+0	1.6e+3	1.6e+3	1.0e+0

Table 31.: Actual and simulated runtimes for SMAC on HPOlib.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	3.2e+1/	1.0e+5/	3.3e+3	5.7e+1/	5.5e+4/	9.6e+2	6.1e+1/	2.8e+4/	4.6e+2	6.6e+1/	1.4e+4/	2.2e+2
2	3.2e+1/	4.8e+4/	1.5e+3	5.6e+1/	2.5e+4/	4.4e+2	6.1e+1/	1.4e+4/	2.3e+2	6.6e+1/	7.1e+3/	1.1e+2
3	3.2e+1/	1.5e+4/	4.7e+2	5.7e+1/	8.4e+3/	1.5e+2	6.1e+1/	4.2e+3/	7.0e+1	6.5e+1/	2.3e+3/	3.5e+1
4	3.2e+1/	7.3e+3/	2.3e+2	5.7e+1/	3.9e+3/	6.9e+1	6.0e+1/	1.9e+3/	3.1e+1	6.6e+1/	1.1e+3/	1.7e+1

Table 32.: Actual and simulated runtimes for Random on JAHS-Bench-201.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	3.9e+1/	1.4e+8/	3.5e+6	6.1e+1/	7.1e+7/	1.2e+6	7.1e+1/	3.5e+7/	4.9e+5	1.0e+2/	1.8e+7/	1.7e+5
2	3.9e+1/	2.1e+8/	5.2e+6	6.4e+1/	1.0e+8/	1.6e+6	7.0e+1/	5.2e+7/	7.4e+5	1.1e+2/	2.6e+7/	2.4e+5
3	3.9e+1/	2.3e+7/	5.9e+5	6.2e+1/	1.2e+7/	1.9e+5	7.0e+1/	5.9e+6/	8.4e+4	1.1e+2/	2.9e+6/	2.8e+4

Table 33.: Actual and simulated runtimes for HyperBand on JAHS-Bench-201.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	3.1e+2/	1.1e+8/	3.6e+5	3.2e+2/	5.6e+7/	1.8e+5	3.6e+2/	2.8e+7/	7.7e+4	4.9e+2/	1.4e+7/	2.9e+4
2	3.1e+2/	1.6e+8/	5.3e+5	3.3e+2/	8.1e+7/	2.5e+5	3.8e+2/	4.1e+7/	1.1e+5	5.0e+2/	2.0e+7/	4.1e+4
3	3.1e+2/	2.2e+7/	6.9e+4	3.3e+2/	1.1e+7/	3.3e+4	3.7e+2/	5.4e+6/	1.5e+4	5.0e+2/	2.7e+6/	5.5e+3

Table 34.: Actual and simulated runtimes for TPE on JAHS-Bench-201.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	7.8e+0/	2.1e+7/	2.7e+6	1.2e+1/	1.2e+7/	9.4e+5	1.6e+1/	6.4e+6/	4.0e+5	2.6e+1/	2.9e+6/	1.1e+5
2	8.0e+0/	2.9e+7/	3.6e+6	1.2e+1/	1.5e+7/	1.2e+6	1.6e+1/	7.6e+6/	4.7e+5	2.8e+1/	4.0e+6/	1.5e+5
3	7.9e+0/	1.8e+6/	2.3e+5	1.2e+1/	8.5e+5/	6.9e+4	1.6e+1/	4.8e+5/	2.9e+4	2.7e+1/	2.5e+5/	9.4e+3

Table 35.: Actual and simulated runtimes for BOHB on JAHS-Bench-201.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	4.9e+1/	1.3e+7/	2.7e+5	5.3e+1/	7.0e+6/	1.3e+5	6.9e+1/	3.5e+6/	5.0e+4	1.1e+2/	1.9e+6/	1.7e+4
2	4.9e+1/	1.9e+7/	3.8e+5	5.3e+1/	9.2e+6/	1.7e+5	7.0e+1/	4.6e+6/	6.6e+4	1.1e+2/	2.5e+6/	2.3e+4
3	4.9e+1/	1.9e+6/	3.8e+4	5.4e+1/	9.4e+5/	1.8e+4	7.0e+1/	4.7e+5/	6.6e+3	1.1e+2/	2.5e+5/	2.2e+3

Table 36.: Actual and simulated runtimes for HEBO on JAHS-Bench-201.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	1.3e+4/	3.4e+7/	2.7e+3	1.3e+4/	1.7e+7/	1.3e+3	1.3e+4/	8.8e+6/	6.5e+2	1.3e+4/	4.3e+6/	3.3e+2
2	1.3e+4/	3.1e+7/	2.3e+3	1.3e+4/	1.6e+7/	1.2e+3	1.3e+4/	7.9e+6/	6.0e+2	1.3e+4/	4.1e+6/	3.1e+2
3	1.4e+4/	1.7e+6/	1.2e+2	1.4e+4/	8.4e+5/	6.1e+1	1.5e+4/	4.1e+5/	2.7e+1	1.4e+4/	2.2e+5/	1.5e+1

Table 37.: Actual and simulated runtimes for DEHB on JAHS-Bench-201.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	9.0e+0/	1.7e+7/	1.9e+6	2.2e+3/	8.4e+6/	3.7e+3	2.4e+3/	4.2e+6/	1.7e+3	2.6e+3/	2.1e+6/	8.1e+2
2	9.1e+0/	2.2e+7/	2.5e+6	1.9e+4/	1.1e+7/	5.5e+2	2.1e+4/	5.4e+6/	2.6e+2	2.3e+4/	2.7e+6/	1.2e+2
3	9.1e+0/	2.3e+6/	2.5e+5	1.4e+4/	1.3e+6/	9.0e+1	1.5e+4/	6.0e+5/	4.1e+1	1.6e+4/	3.1e+5/	1.9e+1

Table 38.: Actual and simulated runtimes for NePS on JAHS-Bench-201.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	7.7e+2/	1.1e+7/	1.4e+4	7.9e+2/	5.7e+6/	7.2e+3	8.7e+2/	3.0e+6/	3.4e+3	9.2e+2/	1.5e+6/	1.7e+3
2	7.8e+2/	1.6e+7/	2.1e+4	7.8e+2/	8.4e+6/	1.1e+4	8.7e+2/	4.3e+6/	4.9e+3	9.3e+2/	2.3e+6/	2.4e+3
3	7.8e+2/	2.2e+6/	2.8e+3	8.0e+2/	1.1e+6/	1.4e+3	8.8e+2/	5.8e+5/	6.6e+2	9.5e+2/	3.0e+5/	3.2e+2

Table 39.: Actual and simulated runtimes for Random on LCBench.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	1.1e+2	4.8e+5	4.5e+3	1.4e+2	2.4e+5	1.8e+3	1.5e+2	1.2e+5	7.9e+2	1.7e+2	6.1e+4	3.6e+2
2	1.1e+2	2.3e+6	2.2e+4	1.4e+2	1.2e+6	8.4e+3	1.6e+2	6.0e+5	3.8e+3	1.7e+2	3.0e+5	1.8e+3
3	1.1e+2	3.6e+5	3.4e+3	1.4e+2	1.8e+5	1.3e+3	1.5e+2	9.1e+4	5.9e+2	1.7e+2	4.6e+4	2.7e+2
4	1.1e+2	3.4e+5	3.1e+3	1.4e+2	1.7e+5	1.2e+3	1.5e+2	8.5e+4	5.5e+2	1.7e+2	4.3e+4	2.5e+2
5	1.1e+2	3.2e+5	3.0e+3	1.4e+2	1.6e+5	1.2e+3	1.5e+2	8.1e+4	5.3e+2	1.7e+2	4.1e+4	2.4e+2
6	1.1e+2	3.2e+5	3.0e+3	1.4e+2	1.6e+5	1.2e+3	1.5e+2	8.1e+4	5.3e+2	1.7e+2	4.1e+4	2.4e+2
7	1.1e+2	3.4e+5	3.1e+3	1.3e+2	1.7e+5	1.3e+3	1.5e+2	8.5e+4	5.6e+2	1.7e+2	4.3e+4	2.5e+2
8	1.1e+2	2.1e+5	2.0e+3	1.4e+2	1.1e+5	7.9e+2	1.5e+2	5.4e+4	3.5e+2	1.7e+2	2.7e+4	1.6e+2
9	1.1e+2	2.0e+5	1.9e+3	1.3e+2	1.0e+5	7.6e+2	1.5e+2	5.1e+4	3.3e+2	1.7e+2	2.6e+4	1.5e+2
10	1.1e+2	3.5e+5	3.2e+3	1.4e+2	1.7e+5	1.3e+3	1.5e+2	8.8e+4	5.7e+2	1.7e+2	4.4e+4	2.6e+2
11	1.1e+2	2.2e+5	2.1e+3	1.4e+2	1.1e+5	7.7e+2	1.5e+2	5.6e+4	3.6e+2	1.7e+2	2.8e+4	1.6e+2
12	1.1e+2	3.0e+5	2.8e+3	1.3e+2	1.5e+5	1.1e+3	1.5e+2	7.6e+4	4.9e+2	1.7e+2	3.9e+4	2.3e+2
13	1.1e+2	2.3e+5	2.1e+3	1.4e+2	1.1e+5	8.2e+2	1.6e+2	5.7e+4	3.7e+2	1.7e+2	2.9e+4	1.7e+2
14	1.1e+2	2.8e+5	2.6e+3	1.4e+2	1.4e+5	1.0e+3	1.6e+2	7.1e+4	4.6e+2	1.7e+2	3.6e+4	2.1e+2
15	1.1e+2	1.1e+6	1.0e+4	1.4e+2	5.5e+5	4.0e+3	1.5e+2	2.7e+5	1.8e+3	1.7e+2	1.4e+5	8.0e+2
16	1.1e+2	2.9e+5	2.7e+3	1.4e+2	1.4e+5	1.1e+3	1.6e+2	7.2e+4	4.7e+2	1.7e+2	3.7e+4	2.1e+2
17	1.1e+2	5.4e+5	4.9e+3	1.4e+2	2.7e+5	1.9e+3	1.5e+2	1.3e+5	8.7e+2	1.7e+2	6.7e+4	3.9e+2
18	1.1e+2	5.0e+5	4.6e+3	1.4e+2	2.5e+5	1.8e+3	1.5e+2	1.2e+5	8.1e+2	1.7e+2	6.3e+4	3.6e+2
19	1.1e+2	6.0e+5	5.6e+3	1.4e+2	3.0e+5	2.1e+3	1.6e+2	1.5e+5	9.5e+2	1.7e+2	7.6e+4	4.4e+2
20	1.1e+2	6.7e+5	6.2e+3	1.4e+2	3.3e+5	2.4e+3	1.5e+2	1.7e+5	1.1e+3	1.7e+2	8.5e+4	4.9e+2
21	1.1e+2	4.9e+5	4.5e+3	1.4e+2	2.4e+5	1.7e+3	1.5e+2	1.2e+5	7.8e+2	1.7e+2	6.1e+4	3.6e+2
22	1.1e+2	6.1e+5	5.6e+3	1.4e+2	3.0e+5	2.2e+3	1.5e+2	1.5e+5	1.0e+3	1.7e+2	7.9e+4	4.6e+2
23	1.1e+2	4.3e+5	4.0e+3	1.4e+2	2.1e+5	1.5e+3	1.5e+2	1.1e+5	6.9e+2	1.7e+2	5.3e+4	3.1e+2
24	1.1e+2	4.0e+6	3.7e+4	1.4e+2	2.0e+6	1.4e+4	1.5e+2	1.0e+6	6.6e+3	1.7e+2	5.0e+5	2.9e+3
25	1.1e+2	1.1e+6	9.7e+3	1.4e+2	5.3e+5	3.8e+3	1.5e+2	2.6e+5	1.7e+3	1.7e+2	1.3e+5	7.7e+2
26	1.1e+2	1.9e+6	1.8e+4	1.4e+2	9.6e+5	6.9e+3	1.5e+2	4.8e+5	3.1e+3	1.7e+2	2.4e+5	1.4e+3
27	1.1e+2	3.4e+5	3.1e+3	1.4e+2	1.7e+5	1.2e+3	1.5e+2	8.5e+4	5.5e+2	1.7e+2	4.3e+4	2.5e+2
28	1.1e+2	2.6e+5	2.4e+3	1.3e+2	1.3e+5	9.7e+2	1.5e+2	6.5e+4	4.2e+2	1.7e+2	3.3e+4	1.9e+2
29	1.1e+2	2.1e+6	1.9e+4	1.4e+2	1.1e+6	7.8e+3	1.6e+2	5.4e+5	3.4e+3	1.7e+2	2.7e+5	1.6e+3
30	1.1e+2	4.8e+6	4.4e+4	1.4e+2	2.4e+6	1.7e+4	1.5e+2	1.2e+6	7.8e+3	1.7e+2	6.1e+5	3.5e+3
31	1.1e+2	1.6e+5	1.5e+3	1.4e+2	8.0e+4	5.9e+2	1.5e+2	4.0e+4	2.6e+2	1.7e+2	2.0e+4	1.2e+2
32	1.1e+2	2.6e+5	2.3e+3	1.4e+2	1.3e+5	9.2e+2	1.5e+2	6.5e+4	4.2e+2	1.7e+2	3.2e+4	1.9e+2
33	1.1e+2	1.7e+6	1.6e+4	1.4e+2	8.6e+5	6.2e+3	1.6e+2	4.3e+5	2.8e+3	1.7e+2	2.2e+5	1.2e+3
34	1.1e+2	3.8e+5	3.5e+3	1.4e+2	1.9e+5	1.3e+3	1.6e+2	9.5e+4	6.1e+2	1.7e+2	4.8e+4	2.8e+2

Table 40.: Actual and simulated runtimes for HyperBand on LCBench.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	3.8e+2	4.0e+5	1.1e+3	4.3e+2	2.0e+5	4.7e+2	4.9e+2	1.0e+5	2.1e+2	6.0e+2	5.1e+4	8.5e+1
2	3.8e+2	1.9e+6	5.0e+3	4.3e+2	9.4e+5	2.2e+3	4.9e+2	4.8e+5	9.7e+2	6.0e+2	2.4e+5	4.0e+2
3	3.8e+2	3.1e+5	8.2e+2	4.3e+2	1.6e+5	3.6e+2	4.9e+2	7.8e+4	1.6e+2	6.0e+2	3.9e+4	6.5e+1
4	3.8e+2	2.8e+5	7.5e+2	4.3e+2	1.4e+5	3.3e+2	4.9e+2	7.0e+4	1.4e+2	6.0e+2	3.5e+4	5.9e+1
5	3.8e+2	2.4e+5	6.3e+2	4.3e+2	1.2e+5	2.8e+2	4.8e+2	6.0e+4	1.2e+2	6.0e+2	3.0e+4	5.0e+1
6	3.8e+2	2.6e+5	6.9e+2	4.3e+2	1.3e+5	3.1e+2	4.9e+2	6.6e+4	1.4e+2	6.0e+2	3.4e+4	5.6e+1
7	3.8e+2	3.0e+5	7.9e+2	4.3e+2	1.5e+5	3.5e+2	4.9e+2	7.5e+4	1.5e+2	6.0e+2	3.8e+4	6.3e+1
8	3.8e+2	1.8e+5	4.9e+2	4.3e+2	9.2e+4	2.1e+2	4.8e+2	4.6e+4	9.6e+1	6.0e+2	2.3e+4	3.9e+1
9	3.8e+2	1.6e+5	4.3e+2	4.3e+2	8.1e+4	1.9e+2	4.8e+2	4.1e+4	8.4e+1	6.0e+2	2.0e+4	3.4e+1
10	3.8e+2	2.7e+5	7.2e+2	4.3e+2	1.4e+5	3.2e+2	4.9e+2	6.8e+4	1.4e+2	6.0e+2	3.5e+4	5.7e+1
11	3.8e+2	1.9e+5	5.1e+2	4.4e+2	9.7e+4	2.2e+2	4.9e+2	4.9e+4	9.9e+1	6.0e+2	2.4e+4	4.0e+1
12	3.8e+2	2.5e+5	6.6e+2	4.3e+2	1.2e+5	2.9e+2	4.9e+2	6.2e+4	1.3e+2	6.0e+2	3.1e+4	5.2e+1
13	3.8e+2	2.1e+5	5.4e+2	4.3e+2	1.0e+5	2.4e+2	4.9e+2	5.1e+4	1.1e+2	6.0e+2	2.6e+4	4.3e+1
14	3.8e+2	2.3e+5	6.2e+2	4.4e+2	1.2e+5	2.7e+2	4.9e+2	5.9e+4	1.2e+2	6.0e+2	2.9e+4	4.9e+1
15	3.8e+2	7.7e+5	2.0e+3	4.3e+2	3.9e+5	9.0e+2	4.8e+2	1.9e+5	4.0e+2	6.0e+2	9.7e+4	1.6e+2
16	3.8e+2	2.6e+5	6.8e+2	4.4e+2	1.3e+5	3.0e+2	4.9e+2	6.4e+4	1.3e+2	6.0e+2	3.2e+4	5.3e+1
17	3.8e+2	4.2e+5	1.1e+3	4.4e+2	2.1e+5	4.9e+2	4.9e+2	1.1e+5	2.2e+2	6.0e+2	5.3e+4	8.9e+1
18	3.8e+2	4.0e+5	1.1e+3	4.3e+2	2.0e+5	4.6e+2	4.9e+2	1.0e+5	2.0e+2	6.0e+2	5.0e+4	8.3e+1
19	3.8e+2	5.2e+5	1.4e+3	4.4e+2	2.6e+5	5.9e+2	4.9e+2	1.3e+5	2.7e+2	6.0e+2	6.6e+4	1.1e+2
20	3.8e+2	5.3e+5	1.4e+3	4.3e+2	2.7e+5	6.2e+2	4.9e+2	1.3e+5	2.7e+2	6.0e+2	6.7e+4	1.1e+2
21	3.8e+2	4.1e+5	1.1e+3	4.4e+2	2.1e+5	4.8e+2	4.9e+2	1.0e+5	2.1e+2	6.0e+2	5.1e+4	8.5e+1
22	3.8e+2	4.7e+5	1.2e+3	4.3e+2	2.3e+5	5.4e+2	4.9e+2	1.2e+5	2.4e+2	6.0e+2	6.0e+4	9.9e+1
23	3.8e+2	3.4e+5	9.0e+2	4.3e+2	1.7e+5	4.0e+2	4.9e+2	8.6e+4	1.8e+2	6.0e+2	4.3e+4	7.1e+1
24	3.8e+2	3.3e+6	8.7e+3	4.3e+2	1.6e+6	3.8e+3	4.9e+2	8.2e+5	1.7e+3	6.0e+2	4.1e+5	6.9e+2
25	3.8e+2	8.1e+5	2.1e+3	4.3e+2	4.1e+5	9.4e+2	4.9e+2	2.0e+5	4.2e+2	6.0e+2	1.0e+5	1.7e+2
26	3.8e+2	1.5e+6	4.0e+3	4.3e+2	7.6e+5	1.8e+3	4.9e+2	3.8e+5	7.7e+2	6.0e+2	1.9e+5	3.1e+2
27	3.8e+2	2.7e+5	7.0e+2	4.3e+2	1.3e+5	3.1e+2	4.9e+2	6.7e+4	1.4e+2	6.0e+2	3.3e+4	5.5e+1
28	3.8e+2	2.1e+5	5.5e+2	4.3e+2	1.1e+5	2.4e+2	4.9e+2	5.2e+4	1.1e+2	6.0e+2	2.6e+4	4.4e+1
29	3.8e+2	1.6e+6	4.2e+3	4.3e+2	7.9e+5	1.8e+3	4.9e+2	3.9e+5	8.1e+2	6.0e+2	2.0e+5	3.3e+2
30	3.8e+2	3.5e+6	9.1e+3	4.4e+2	1.7e+6	4.0e+3	4.9e+2	8.6e+5	1.7e+3	6.0e+2	4.3e+5	7.2e+2
31	3.8e+2	1.5e+5	3.8e+2	4.3e+2	7.3e+4	1.7e+2	4.9e+2	3.6e+4	7.4e+1	6.0e+2	1.8e+4	3.0e+1
32	3.8e+2	2.2e+5	5.9e+2	4.4e+2	1.1e+5	2.5e+2	4.9e+2	5.5e+4	1.1e+2	6.0e+2	2.8e+4	4.6e+1
33	3.8e+2	1.4e+6	3.6e+3	4.3e+2	6.8e+5	1.6e+3	4.9e+2	3.4e+5	7.0e+2	6.0e+2	1.7e+5	2.8e+2
34	3.8e+2	3.3e+5	8.7e+2	4.3e+2	1.6e+5	3.8e+2	4.9e+2	8.2e+4	1.7e+2	6.0e+2	4.1e+4	6.8e+1

Table 41.: Actual and simulated runtimes for TPE on LCBench.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	1.6e+1	3.9e+4	2.4e+3	2.1e+1	2.0e+4	9.6e+2	2.6e+1	9.5e+3	3.7e+2	3.9e+1	4.9e+3	1.3e+2
2	1.6e+1	2.7e+5	1.6e+4	2.1e+1	1.4e+5	6.4e+3	2.6e+1	6.3e+4	2.5e+3	3.9e+1	2.8e+4	7.2e+2
3	1.6e+1	6.8e+4	4.1e+3	2.2e+1	2.6e+4	1.2e+3	2.5e+1	1.6e+4	6.4e+2	3.9e+1	7.7e+3	2.0e+2
4	1.6e+1	2.9e+4	1.7e+3	2.1e+1	1.2e+4	5.7e+2	2.5e+1	7.5e+3	2.9e+2	3.8e+1	3.5e+3	9.2e+1
5	1.6e+1	1.5e+4	9.3e+2	2.0e+1	9.4e+3	4.6e+2	2.6e+1	4.0e+3	1.5e+2	3.8e+1	2.3e+3	6.0e+1
6	1.6e+1	2.1e+4	1.3e+3	2.1e+1	1.1e+4	5.5e+2	2.5e+1	5.8e+3	2.3e+2	3.9e+1	3.5e+3	9.2e+1
7	1.6e+1	4.0e+4	2.4e+3	2.1e+1	2.0e+4	9.7e+2	2.6e+1	9.4e+3	3.7e+2	3.8e+1	5.0e+3	1.3e+2
8	1.7e+1	3.3e+4	2.0e+3	2.1e+1	1.8e+4	8.4e+2	2.6e+1	7.2e+3	2.8e+2	3.8e+1	3.9e+3	1.0e+2
9	1.6e+1	1.5e+4	9.2e+2	2.1e+1	7.6e+3	3.7e+2	2.6e+1	4.2e+3	1.6e+2	3.9e+1	2.5e+3	6.4e+1
10	1.6e+1	3.0e+4	1.8e+3	2.1e+1	1.6e+4	7.4e+2	2.6e+1	8.3e+3	3.2e+2	3.9e+1	4.1e+3	1.1e+2
11	1.6e+1	4.0e+4	2.5e+3	2.2e+1	2.0e+4	9.3e+2	2.6e+1	1.1e+4	4.3e+2	3.9e+1	4.7e+3	1.2e+2
12	1.6e+1	5.3e+4	3.3e+3	2.2e+1	2.6e+4	1.2e+3	2.5e+1	1.3e+4	5.1e+2	3.8e+1	6.3e+3	1.6e+2
13	1.6e+1	2.8e+4	1.7e+3	2.2e+1	1.5e+4	6.7e+2	2.5e+1	6.6e+3	2.6e+2	3.8e+1	4.3e+3	1.1e+2
14	1.6e+1	3.6e+4	2.2e+3	2.1e+1	1.7e+4	7.8e+2	2.6e+1	8.6e+3	3.3e+2	3.8e+1	4.2e+3	1.1e+2
15	1.6e+1	7.8e+4	4.8e+3	2.1e+1	3.2e+4	1.5e+3	2.6e+1	1.7e+4	6.7e+2	3.8e+1	9.5e+3	2.5e+2
16	1.6e+1	3.8e+4	2.3e+3	2.1e+1	1.9e+4	8.9e+2	2.6e+1	1.2e+4	4.5e+2	3.8e+1	5.3e+3	1.4e+2
17	1.6e+1	6.8e+4	4.2e+3	2.1e+1	3.4e+4	1.6e+3	2.5e+1	1.7e+4	6.7e+2	3.8e+1	9.6e+3	2.5e+2
18	1.7e+1	5.7e+4	3.4e+3	2.1e+1	2.7e+4	1.3e+3	2.5e+1	1.4e+4	5.5e+2	3.9e+1	7.1e+3	1.8e+2
19	1.6e+1	1.3e+5	8.0e+3	2.2e+1	7.1e+4	3.3e+3	2.5e+1	4.1e+4	1.6e+3	3.8e+1	1.9e+4	4.9e+2
20	1.6e+1	8.2e+4	5.0e+3	2.1e+1	4.2e+4	2.0e+3	2.6e+1	2.0e+4	7.6e+2	3.8e+1	9.6e+3	2.5e+2
21	1.6e+1	5.1e+4	3.1e+3	2.1e+1	1.8e+4	8.6e+2	2.6e+1	1.3e+4	4.9e+2	3.8e+1	5.0e+3	1.3e+2
22	1.6e+1	2.9e+4	1.8e+3	2.1e+1	1.8e+4	8.6e+2	2.5e+1	7.5e+3	3.0e+2	3.8e+1	4.2e+3	1.1e+2
23	1.6e+1	1.2e+4	7.3e+2	2.0e+1	6.3e+3	3.2e+2	2.5e+1	4.8e+3	2.0e+2	3.7e+1	2.7e+3	7.3e+1
24	1.6e+1	5.2e+5	3.2e+4	2.2e+1	2.7e+5	1.2e+4	2.6e+1	1.3e+5	5.2e+3	3.8e+1	7.0e+4	1.8e+3
25	1.6e+1	2.9e+5	1.8e+4	2.2e+1	1.3e+5	5.9e+3	2.6e+1	6.7e+4	2.6e+3	3.8e+1	3.1e+4	8.3e+2
26	1.6e+1	1.5e+5	9.4e+3	2.1e+1	7.9e+4	3.8e+3	2.6e+1	4.2e+4	1.6e+3	3.8e+1	2.4e+4	6.2e+2
27	1.6e+1	4.3e+4	2.6e+3	2.1e+1	2.6e+4	1.2e+3	2.6e+1	1.2e+4	4.6e+2	3.8e+1	6.0e+3	1.6e+2
28	1.6e+1	4.0e+4	2.4e+3	2.2e+1	2.0e+4	9.3e+2	2.6e+1	1.0e+4	3.9e+2	3.8e+1	4.9e+3	1.3e+2
29	1.6e+1	1.9e+5	1.2e+4	2.1e+1	8.8e+4	4.2e+3	2.6e+1	4.8e+4	1.9e+3	3.9e+1	2.7e+4	6.8e+2
30	1.6e+1	4.8e+5	2.9e+4	2.1e+1	2.7e+5	1.3e+4	2.6e+1	1.3e+5	5.2e+3	3.8e+1	6.1e+4	1.6e+3
31	1.6e+1	2.3e+4	1.4e+3	2.1e+1	1.2e+4	5.7e+2	2.6e+1	5.4e+3	2.1e+2	3.8e+1	3.2e+3	8.2e+1
32	1.6e+1	5.9e+4	3.6e+3	2.1e+1	3.0e+4	1.4e+3	2.5e+1	1.4e+4	5.7e+2	3.9e+1	6.9e+3	1.8e+2
33	1.6e+1	2.0e+5	1.3e+4	2.2e+1	1.0e+5	4.6e+3	2.6e+1	5.1e+4	2.0e+3	3.8e+1	2.6e+4	6.8e+2
34	1.6e+1	3.6e+4	2.2e+3	2.1e+1	2.3e+4	1.1e+3	2.5e+1	1.0e+4	4.1e+2	3.8e+1	4.7e+3	1.2e+2

Table 42.: Actual and simulated runtimes for BOHB on LCBench.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	6.8e+1	2.9e+4	4.3e+2	7.1e+1	1.6e+4	2.2e+2	7.8e+1	7.7e+3	9.8e+1	9.1e+1	4.2e+3	4.6e+1
2	6.8e+1	2.0e+5	2.9e+3	7.2e+1	9.2e+4	1.3e+3	7.8e+1	4.9e+4	6.3e+2	9.1e+1	2.7e+4	2.9e+2
3	6.8e+1	4.3e+4	6.3e+2	7.2e+1	2.5e+4	3.5e+2	7.9e+1	1.1e+4	1.4e+2	9.0e+1	5.4e+3	5.9e+1
4	6.8e+1	2.2e+4	3.2e+2	7.2e+1	1.0e+4	1.4e+2	7.9e+1	5.7e+3	7.2e+1	9.1e+1	3.1e+3	3.4e+1
5	6.8e+1	1.6e+4	2.4e+2	7.2e+1	7.7e+3	1.1e+2	7.9e+1	4.5e+3	5.7e+1	9.0e+1	2.5e+3	2.7e+1
6	6.8e+1	2.0e+4	3.0e+2	7.2e+1	1.0e+4	1.4e+2	7.9e+1	5.1e+3	6.6e+1	9.0e+1	2.8e+3	3.2e+1
7	6.9e+1	3.6e+4	5.2e+2	7.1e+1	1.9e+4	2.6e+2	7.9e+1	9.4e+3	1.2e+2	9.0e+1	4.9e+3	5.5e+1
8	6.8e+1	2.4e+4	3.5e+2	7.2e+1	1.0e+4	1.4e+2	7.9e+1	5.9e+3	7.4e+1	9.0e+1	2.9e+3	3.3e+1
9	6.8e+1	1.2e+4	1.8e+2	7.2e+1	6.5e+3	9.1e+1	7.9e+1	3.1e+3	4.0e+1	9.0e+1	1.7e+3	1.9e+1
10	6.8e+1	2.5e+4	3.6e+2	7.1e+1	1.2e+4	1.7e+2	7.9e+1	6.1e+3	7.8e+1	9.0e+1	3.2e+3	3.6e+1
11	6.9e+1	3.2e+4	4.6e+2	7.2e+1	1.6e+4	2.2e+2	7.9e+1	8.2e+3	1.0e+2	9.0e+1	4.0e+3	4.5e+1
12	6.8e+1	3.2e+4	4.7e+2	7.2e+1	1.8e+4	2.5e+2	7.9e+1	9.7e+3	1.2e+2	9.0e+1	4.3e+3	4.8e+1
13	6.9e+1	2.3e+4	3.4e+2	7.2e+1	1.1e+4	1.5e+2	7.8e+1	5.7e+3	7.3e+1	9.1e+1	3.1e+3	3.5e+1
14	6.8e+1	2.7e+4	4.0e+2	7.2e+1	1.3e+4	1.9e+2	7.8e+1	7.0e+3	9.0e+1	9.0e+1	3.7e+3	4.1e+1
15	6.8e+1	5.1e+4	7.5e+2	7.2e+1	2.7e+4	3.7e+2	7.8e+1	1.4e+4	1.8e+2	9.0e+1	7.3e+3	8.1e+1
16	6.8e+1	3.4e+4	5.0e+2	7.2e+1	1.6e+4	2.2e+2	7.8e+1	8.2e+3	1.1e+2	9.1e+1	4.3e+3	4.7e+1
17	6.8e+1	5.5e+4	8.1e+2	7.2e+1	2.5e+4	3.5e+2	7.8e+1	1.3e+4	1.7e+2	9.0e+1	7.0e+3	7.8e+1
18	6.8e+1	4.4e+4	6.5e+2	7.2e+1	2.4e+4	3.3e+2	7.8e+1	1.2e+4	1.6e+2	9.0e+1	6.2e+3	6.9e+1
19	6.9e+1	1.0e+5	1.5e+3	7.1e+1	4.9e+4	6.9e+2	7.9e+1	2.2e+4	2.8e+2	9.0e+1	1.2e+4	1.3e+2
20	6.9e+1	6.9e+4	1.0e+3	7.1e+1	3.2e+4	4.5e+2	7.9e+1	1.8e+4	2.3e+2	9.0e+1	8.8e+3	9.7e+1
21	6.8e+1	3.6e+4	5.3e+2	7.2e+1	1.8e+4	2.5e+2	7.8e+1	1.0e+4	1.3e+2	9.1e+1	5.1e+3	5.6e+1
22	6.8e+1	2.8e+4	4.1e+2	7.2e+1	1.4e+4	1.9e+2	7.8e+1	8.0e+3	1.0e+2	9.0e+1	4.4e+3	4.9e+1
23	6.8e+1	1.4e+4	2.1e+2	7.1e+1	7.7e+3	1.1e+2	7.8e+1	4.1e+3	5.3e+1	9.0e+1	2.5e+3	2.8e+1
24	6.8e+1	3.9e+5	5.7e+3	7.2e+1	2.0e+5	2.8e+3	7.9e+1	1.0e+5	1.3e+3	9.0e+1	5.3e+4	5.9e+2
25	6.8e+1	1.9e+5	2.8e+3	7.1e+1	1.1e+5	1.5e+3	7.9e+1	4.6e+4	5.9e+2	9.0e+1	2.0e+4	2.2e+2
26	6.8e+1	1.6e+5	2.3e+3	7.2e+1	7.2e+4	9.9e+2	7.8e+1	3.9e+4	4.9e+2	9.0e+1	2.2e+4	2.5e+2
27	6.8e+1	3.1e+4	4.6e+2	7.1e+1	1.6e+4	2.3e+2	7.8e+1	8.0e+3	1.0e+2	9.0e+1	4.3e+3	4.8e+1
28	6.7e+1	2.2e+4	3.3e+2	7.2e+1	1.1e+4	1.5e+2	7.9e+1	5.3e+3	6.7e+1	9.0e+1	2.7e+3	3.0e+1
29	6.8e+1	1.5e+5	2.2e+3	7.2e+1	8.0e+4	1.1e+3	7.9e+1	4.2e+4	5.3e+2	9.0e+1	2.0e+4	2.2e+2
30	6.9e+1	3.9e+5	5.7e+3	7.2e+1	2.0e+5	2.8e+3	7.8e+1	1.0e+5	1.3e+3	9.1e+1	5.0e+4	5.5e+2
31	6.8e+1	1.7e+4	2.5e+2	7.2e+1	8.6e+3	1.2e+2	7.8e+1	3.9e+3	5.0e+1	9.0e+1	2.3e+3	2.5e+1
32	6.8e+1	4.0e+4	5.9e+2	7.2e+1	1.6e+4	2.2e+2	7.9e+1	7.7e+3	9.8e+1	9.1e+1	4.0e+3	4.4e+1
33	6.8e+1	1.6e+5	2.4e+3	7.2e+1	8.1e+4	1.1e+3	7.9e+1	4.1e+4	5.2e+2	8.9e+1	2.1e+4	2.3e+2
34	6.9e+1	3.7e+4	5.3e+2	7.2e+1	1.7e+4	2.3e+2	7.8e+1	8.9e+3	1.1e+2	9.1e+1	4.9e+3	5.4e+1

Table 43.: Actual and simulated runtimes for HEBO on LCBench.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	5.0e+3	2.5e+4	5.1e+0	4.9e+3	1.5e+4	3.0e+0	4.8e+3	8.9e+3	1.9e+0	4.9e+3	7.0e+3	1.4e+0
2	4.3e+3	3.1e+5	7.2e+1	4.6e+3	1.6e+5	3.4e+1	4.3e+3	8.7e+4	2.0e+1	4.3e+3	5.3e+4	1.2e+1
3	4.8e+3	7.8e+4	1.6e+1	4.7e+3	4.0e+4	8.4e+0	4.7e+3	2.1e+4	4.4e+0	4.6e+3	1.2e+4	2.6e+0
4	4.9e+3	2.1e+4	4.3e+0	4.7e+3	1.4e+4	3.0e+0	5.0e+3	9.5e+3	1.9e+0	5.0e+3	7.7e+3	1.5e+0
5	4.4e+3	2.1e+4	4.7e+0	4.3e+3	1.2e+4	2.9e+0	4.4e+3	8.8e+3	2.0e+0	4.3e+3	6.6e+3	1.5e+0
6	4.8e+3	2.1e+4	4.4e+0	4.7e+3	1.4e+4	3.0e+0	4.6e+3	9.6e+3	2.1e+0	4.6e+3	7.1e+3	1.5e+0
7	3.4e+3	5.5e+4	1.6e+1	3.3e+3	2.8e+4	8.6e+0	3.4e+3	1.6e+4	4.7e+0	3.5e+3	8.9e+3	2.6e+0
8	3.4e+3	4.6e+4	1.3e+1	3.5e+3	2.4e+4	6.7e+0	3.6e+3	1.3e+4	3.8e+0	3.9e+3	8.0e+3	2.1e+0
9	3.7e+3	2.4e+4	6.6e+0	3.7e+3	1.4e+4	3.9e+0	3.7e+3	8.2e+3	2.2e+0	3.9e+3	5.8e+3	1.5e+0
10	3.9e+3	3.9e+4	9.9e+0	3.8e+3	2.0e+4	5.2e+0	3.9e+3	1.0e+4	2.6e+0	4.0e+3	5.8e+3	1.4e+0
11	4.1e+3	7.6e+4	1.8e+1	4.0e+3	3.9e+4	9.8e+0	4.1e+3	2.0e+4	5.0e+0	4.1e+3	1.1e+4	2.8e+0
12	3.7e+3	7.3e+4	2.0e+1	3.8e+3	3.6e+4	9.5e+0	4.0e+3	1.7e+4	4.4e+0	4.2e+3	9.4e+3	2.3e+0
13	4.0e+3	3.9e+4	9.8e+0	3.8e+3	2.0e+4	5.3e+0	3.8e+3	1.2e+4	3.0e+0	4.0e+3	7.4e+3	1.8e+0
14	3.9e+3	4.1e+4	1.1e+1	3.7e+3	2.1e+4	5.7e+0	3.7e+3	1.2e+4	3.1e+0	4.0e+3	7.0e+3	1.8e+0
15	4.4e+3	8.8e+4	2.0e+1	4.6e+3	4.6e+4	9.9e+0	4.4e+3	2.3e+4	5.3e+0	4.8e+3	1.3e+4	2.7e+0
16	4.3e+3	6.2e+4	1.4e+1	4.3e+3	3.0e+4	6.9e+0	4.1e+3	1.7e+4	4.2e+0	4.2e+3	9.8e+3	2.3e+0
17	4.9e+3	1.2e+5	2.4e+1	4.7e+3	7.0e+4	1.5e+1	4.6e+3	3.8e+4	8.3e+0	4.6e+3	1.8e+4	3.9e+0
18	3.7e+3	6.7e+4	1.8e+1	4.0e+3	3.6e+4	9.0e+0	3.9e+3	2.0e+4	5.0e+0	4.0e+3	1.1e+4	2.7e+0
19	4.0e+3	1.6e+5	4.1e+1	4.0e+3	8.6e+4	2.1e+1	3.9e+3	4.7e+4	1.2e+1	4.0e+3	2.4e+4	6.2e+0
20	4.1e+3	1.4e+5	3.4e+1	4.0e+3	6.1e+4	1.5e+1	4.1e+3	3.2e+4	7.8e+0	4.1e+3	1.7e+4	4.1e+0
21	4.0e+3	5.1e+4	1.3e+1	4.1e+3	3.1e+4	7.6e+0	4.0e+3	1.6e+4	4.1e+0	4.0e+3	8.6e+3	2.1e+0
22	4.7e+3	3.2e+4	6.9e+0	4.6e+3	2.0e+4	4.2e+0	4.6e+3	1.3e+4	2.7e+0	4.8e+3	7.4e+3	1.5e+0
23	4.4e+3	1.6e+4	3.6e+0	4.4e+3	1.0e+4	2.3e+0	4.4e+3	7.4e+3	1.7e+0	4.6e+3	5.7e+3	1.2e+0
24	4.2e+3	4.8e+5	1.1e+2	3.8e+3	2.3e+5	6.1e+1	4.0e+3	1.2e+5	3.0e+1	3.9e+3	6.1e+4	1.6e+1
25	4.1e+3	3.4e+5	8.1e+1	4.2e+3	1.5e+5	3.6e+1	4.1e+3	7.6e+4	1.9e+1	4.3e+3	3.5e+4	8.0e+0
26	4.9e+3	1.4e+5	2.8e+1	4.7e+3	7.4e+4	1.6e+1	4.8e+3	4.4e+4	9.1e+0	4.7e+3	2.3e+4	4.9e+0
27	4.1e+3	7.0e+4	1.7e+1	3.9e+3	3.5e+4	8.8e+0	4.0e+3	1.8e+4	4.6e+0	4.1e+3	1.0e+4	2.5e+0
28	4.0e+3	6.2e+4	1.5e+1	3.7e+3	3.1e+4	8.2e+0	3.9e+3	1.6e+4	4.2e+0	4.1e+3	8.7e+3	2.1e+0
29	4.6e+3	3.0e+5	6.5e+1	4.6e+3	1.4e+5	3.1e+1	4.7e+3	8.2e+4	1.8e+1	4.3e+3	5.1e+4	1.2e+1
30	5.0e+3	2.9e+5	5.7e+1	5.0e+3	1.6e+5	3.2e+1	5.0e+3	9.4e+4	1.9e+1	4.8e+3	6.4e+4	1.3e+1
31	4.0e+3	4.2e+4	1.1e+1	4.1e+3	2.2e+4	5.3e+0	4.1e+3	1.1e+4	2.6e+0	4.2e+3	7.2e+3	1.7e+0
32	4.1e+3	6.7e+4	1.6e+1	4.2e+3	3.6e+4	8.6e+0	3.9e+3	1.8e+4	4.5e+0	4.1e+3	8.4e+3	2.0e+0
33	3.2e+3	1.9e+5	5.9e+1	3.2e+3	9.8e+4	3.0e+1	3.4e+3	5.1e+4	1.5e+1	3.6e+3	2.7e+4	7.4e+0
34	3.9e+3	6.1e+4	1.5e+1	4.0e+3	3.1e+4	7.9e+0	4.0e+3	1.7e+4	4.3e+0	4.1e+3	1.0e+4	2.6e+0

Table 44.: Actual and simulated runtimes for DEHB on LCBench.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	2.4e+1	4.5e+4	1.8e+3	5.6e+1	2.3e+4	4.1e+2	5.9e+1	1.2e+4	2.0e+2	6.0e+1	6.0e+3	9.9e+1
2	2.4e+1	2.2e+5	9.2e+3	5.7e+1	1.1e+5	2.0e+3	5.8e+1	5.7e+4	9.7e+2	6.0e+1	3.0e+4	4.9e+2
3	2.5e+1	4.7e+4	1.9e+3	5.7e+1	2.4e+4	4.3e+2	5.9e+1	1.2e+4	2.0e+2	6.0e+1	6.0e+3	1.0e+2
4	2.4e+1	3.1e+4	1.3e+3	5.6e+1	1.5e+4	2.7e+2	5.9e+1	8.3e+3	1.4e+2	6.0e+1	4.6e+3	7.7e+1
5	2.4e+1	2.3e+4	9.4e+2	5.7e+1	1.1e+4	1.9e+2	5.8e+1	5.8e+3	9.9e+1	6.0e+1	3.1e+3	5.1e+1
6	2.4e+1	3.3e+4	1.4e+3	5.7e+1	1.6e+4	2.9e+2	5.8e+1	8.0e+3	1.4e+2	6.0e+1	4.1e+3	6.9e+1
7	2.4e+1	2.9e+4	1.2e+3	5.6e+1	1.6e+4	2.8e+2	5.8e+1	7.8e+3	1.3e+2	6.0e+1	4.1e+3	6.9e+1
8	2.4e+1	2.4e+4	1.0e+3	5.6e+1	1.2e+4	2.2e+2	5.8e+1	6.0e+3	1.0e+2	6.0e+1	3.1e+3	5.1e+1
9	2.5e+1	1.5e+4	6.1e+2	5.6e+1	7.1e+3	1.3e+2	5.9e+1	3.5e+3	6.0e+1	6.0e+1	1.9e+3	3.1e+1
10	2.4e+1	2.6e+4	1.1e+3	5.7e+1	1.4e+4	2.4e+2	5.9e+1	6.5e+3	1.1e+2	6.0e+1	3.4e+3	5.7e+1
11	2.5e+1	3.0e+4	1.2e+3	5.6e+1	1.5e+4	2.7e+2	5.9e+1	7.2e+3	1.2e+2	6.0e+1	4.0e+3	6.6e+1
12	2.4e+1	3.0e+4	1.2e+3	5.6e+1	1.5e+4	2.7e+2	5.8e+1	7.9e+3	1.4e+2	6.0e+1	3.8e+3	6.4e+1
13	2.4e+1	2.8e+4	1.2e+3	5.7e+1	1.4e+4	2.5e+2	5.9e+1	7.3e+3	1.2e+2	6.0e+1	3.6e+3	6.1e+1
14	2.4e+1	2.8e+4	1.2e+3	5.6e+1	1.4e+4	2.4e+2	5.9e+1	7.0e+3	1.2e+2	6.0e+1	3.7e+3	6.2e+1
15	2.4e+1	6.1e+4	2.5e+3	5.7e+1	3.2e+4	5.7e+2	5.9e+1	1.5e+4	2.6e+2	6.0e+1	8.4e+3	1.4e+2
16	2.4e+1	3.6e+4	1.5e+3	5.7e+1	1.8e+4	3.1e+2	5.8e+1	8.6e+3	1.5e+2	6.0e+1	4.8e+3	7.9e+1
17	2.4e+1	5.2e+4	2.1e+3	5.7e+1	2.8e+4	4.9e+2	5.8e+1	1.3e+4	2.3e+2	6.0e+1	6.6e+3	1.1e+2
18	2.4e+1	5.6e+4	2.3e+3	5.7e+1	2.7e+4	4.8e+2	5.9e+1	1.4e+4	2.4e+2	6.0e+1	6.8e+3	1.1e+2
19	2.4e+1	8.0e+4	3.3e+3	5.7e+1	4.1e+4	7.3e+2	5.9e+1	2.0e+4	3.5e+2	6.0e+1	1.0e+4	1.7e+2
20	2.4e+1	6.2e+4	2.5e+3	5.7e+1	3.1e+4	5.4e+2	5.9e+1	1.6e+4	2.7e+2	6.0e+1	8.0e+3	1.3e+2
21	2.4e+1	4.9e+4	2.0e+3	5.6e+1	2.5e+4	4.4e+2	5.9e+1	1.2e+4	2.0e+2	6.0e+1	6.5e+3	1.1e+2
22	2.4e+1	4.2e+4	1.7e+3	5.6e+1	2.1e+4	3.8e+2	5.9e+1	1.1e+4	1.9e+2	6.0e+1	6.1e+3	1.0e+2
23	2.4e+1	2.8e+4	1.1e+3	5.7e+1	1.5e+4	2.6e+2	5.9e+1	7.6e+3	1.3e+2	6.0e+1	4.1e+3	6.9e+1
24	2.5e+1	3.9e+5	1.6e+4	5.7e+1	2.0e+5	3.6e+3	5.9e+1	1.1e+5	1.8e+3	6.1e+1	5.2e+4	8.7e+2
25	2.4e+1	1.0e+5	4.1e+3	5.6e+1	5.0e+4	8.8e+2	5.9e+1	2.6e+4	4.5e+2	6.0e+1	1.4e+4	2.3e+2
26	2.4e+1	1.8e+5	7.3e+3	5.7e+1	8.7e+4	1.5e+3	5.9e+1	4.4e+4	7.5e+2	6.0e+1	2.4e+4	4.0e+2
27	2.4e+1	3.4e+4	1.4e+3	5.7e+1	1.6e+4	2.8e+2	5.8e+1	8.1e+3	1.4e+2	6.0e+1	4.2e+3	6.9e+1
28	2.4e+1	2.1e+4	8.8e+2	5.7e+1	1.1e+4	1.9e+2	5.9e+1	5.3e+3	9.0e+1	6.0e+1	2.6e+3	4.4e+1
29	2.4e+1	1.6e+5	6.7e+3	5.7e+1	8.4e+4	1.5e+3	5.9e+1	4.0e+4	6.9e+2	6.0e+1	2.4e+4	3.9e+2
30	2.4e+1	4.3e+5	1.8e+4	5.7e+1	2.1e+5	3.7e+3	5.9e+1	1.0e+5	1.8e+3	6.0e+1	6.0e+4	1.0e+3
31	2.4e+1	2.0e+4	8.0e+2	5.6e+1	9.3e+3	1.7e+2	5.9e+1	5.0e+3	8.5e+1	6.0e+1	2.4e+3	3.9e+1
32	2.4e+1	2.7e+4	1.1e+3	5.7e+1	1.3e+4	2.3e+2	5.9e+1	6.5e+3	1.1e+2	6.0e+1	3.2e+3	5.4e+1
33	2.4e+1	1.8e+5	7.3e+3	5.7e+1	9.2e+4	1.6e+3	5.8e+1	4.4e+4	7.6e+2	6.0e+1	2.4e+4	3.9e+2
34	2.4e+1	4.5e+4	1.8e+3	5.6e+1	2.2e+4	3.9e+2	5.9e+1	1.1e+4	1.9e+2	6.0e+1	5.7e+3	9.5e+1

Table 45.: Actual and simulated runtimes for NePS on LCBench.

ID	$P = 1$			$P = 2$			$P = 4$			$P = 8$		
	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast	Act.	Sim.	× Fast
1	6.8e+2	4.0e+4	5.9e+1	8.0e+2	2.1e+4	2.7e+1	1.1e+3	1.1e+4	1.0e+1	1.4e+3	6.0e+3	4.2e+0
2	6.9e+2	1.9e+5	2.7e+2	7.2e+2	1.0e+5	1.4e+2	8.5e+2	5.2e+4	6.1e+1	1.0e+3	2.6e+4	2.7e+1
3	6.8e+2	3.1e+4	4.6e+1	8.4e+2	1.7e+4	2.0e+1	1.2e+3	8.8e+3	7.4e+0	1.6e+3	4.7e+3	3.0e+0
4	6.8e+2	2.8e+4	4.1e+1	8.7e+2	1.5e+4	1.8e+1	1.1e+3	8.0e+3	7.1e+0	1.6e+3	4.3e+3	2.7e+0
5	6.9e+2	2.4e+4	3.5e+1	8.5e+2	1.3e+4	1.5e+1	1.2e+3	6.8e+3	5.4e+0	1.7e+3	3.7e+3	2.2e+0
6	6.8e+2	2.7e+4	3.9e+1	8.7e+2	1.4e+4	1.6e+1	1.2e+3	7.4e+3	6.3e+0	1.7e+3	4.1e+3	2.4e+0
7	6.8e+2	3.0e+4	4.4e+1	8.2e+2	1.6e+4	2.0e+1	1.2e+3	8.3e+3	7.1e+0	1.5e+3	4.5e+3	2.9e+0
8	6.8e+2	1.9e+4	2.8e+1	9.2e+2	1.0e+4	1.1e+1	1.3e+3	5.4e+3	4.1e+0	1.8e+3	3.0e+3	1.7e+0
9	6.8e+2	1.7e+4	2.4e+1	9.1e+2	8.9e+3	9.7e+0	1.4e+3	4.8e+3	3.5e+0	1.7e+3	2.6e+3	1.5e+0
10	6.9e+2	2.8e+4	4.1e+1	8.5e+2	1.5e+4	1.7e+1	1.2e+3	7.7e+3	6.6e+0	1.6e+3	4.2e+3	2.6e+0
11	6.8e+2	2.0e+4	2.9e+1	9.0e+2	1.1e+4	1.2e+1	1.3e+3	5.8e+3	4.5e+0	1.7e+3	3.1e+3	1.8e+0
12	6.9e+2	2.5e+4	3.7e+1	8.7e+2	1.3e+4	1.5e+1	1.2e+3	7.0e+3	5.9e+0	1.6e+3	3.8e+3	2.3e+0
13	6.9e+2	2.1e+4	3.0e+1	9.2e+2	1.1e+4	1.2e+1	1.3e+3	5.9e+3	4.6e+0	1.7e+3	3.2e+3	1.9e+0
14	6.8e+2	2.4e+4	3.5e+1	8.9e+2	1.2e+4	1.4e+1	1.3e+3	6.6e+3	5.3e+0	1.7e+3	3.6e+3	2.2e+0
15	6.9e+2	7.7e+4	1.1e+2	7.6e+2	4.0e+4	5.3e+1	9.6e+2	2.0e+4	2.1e+1	1.3e+3	1.1e+4	8.1e+0
16	6.8e+2	2.6e+4	3.8e+1	8.5e+2	1.4e+4	1.6e+1	1.2e+3	7.3e+3	6.2e+0	1.7e+3	3.9e+3	2.3e+0
17	6.9e+2	4.2e+4	6.2e+1	8.1e+2	2.2e+4	2.7e+1	1.1e+3	1.2e+4	1.1e+1	1.4e+3	6.2e+3	4.3e+0
18	6.8e+2	4.0e+4	5.8e+1	8.3e+2	2.1e+4	2.5e+1	1.1e+3	1.1e+4	9.7e+0	1.4e+3	5.8e+3	4.0e+0
19	6.8e+2	5.2e+4	7.6e+1	7.7e+2	2.8e+4	3.6e+1	1.1e+3	1.4e+4	1.4e+1	1.3e+3	7.5e+3	5.6e+0
20	6.8e+2	5.3e+4	7.8e+1	8.1e+2	2.8e+4	3.5e+1	1.0e+3	1.5e+4	1.4e+1	1.3e+3	7.7e+3	6.0e+0
21	6.8e+2	4.1e+4	6.0e+1	8.1e+2	2.2e+4	2.7e+1	1.1e+3	1.1e+4	1.0e+1	1.5e+3	6.0e+3	4.1e+0
22	6.9e+2	4.7e+4	6.8e+1	7.9e+2	2.5e+4	3.1e+1	1.1e+3	1.3e+4	1.2e+1	1.4e+3	6.7e+3	4.6e+0
23	6.8e+2	3.4e+4	5.0e+1	8.2e+2	1.8e+4	2.2e+1	1.1e+3	9.4e+3	8.2e+0	1.6e+3	5.0e+3	3.2e+0
24	6.9e+2	3.2e+5	4.7e+2	7.1e+2	1.7e+5	2.4e+2	8.8e+2	8.8e+4	1.0e+2	9.2e+2	4.6e+4	5.1e+1
25	6.9e+2	8.1e+4	1.2e+2	7.4e+2	4.2e+4	5.7e+1	9.3e+2	2.2e+4	2.3e+1	1.2e+3	1.1e+4	9.6e+0
26	6.8e+2	1.5e+5	2.2e+2	7.3e+2	7.8e+4	1.1e+2	8.5e+2	4.0e+4	4.8e+1	1.0e+3	2.1e+4	2.1e+1
27	6.9e+2	2.7e+4	3.9e+1	8.5e+2	1.4e+4	1.7e+1	1.2e+3	7.4e+3	6.2e+0	1.6e+3	4.0e+3	2.4e+0
28	6.8e+2	2.1e+4	3.1e+1	8.9e+2	1.1e+4	1.3e+1	1.3e+3	6.0e+3	4.8e+0	1.7e+3	3.3e+3	1.9e+0
29	6.9e+2	1.6e+5	2.3e+2	7.3e+2	8.1e+4	1.1e+2	8.5e+2	4.3e+4	5.0e+1	1.0e+3	2.2e+4	2.2e+1
30	6.9e+2	3.4e+5	4.9e+2	7.2e+2	1.8e+5	2.5e+2	8.2e+2	9.2e+4	1.1e+2	9.0e+2	4.8e+4	5.3e+1
31	6.9e+2	1.5e+4	2.2e+1	9.4e+2	8.1e+3	8.6e+0	1.4e+3	4.3e+3	3.0e+0	1.7e+3	2.5e+3	1.4e+0
32	6.9e+2	2.3e+4	3.3e+1	8.6e+2	1.2e+4	1.4e+1	1.2e+3	6.4e+3	5.2e+0	1.7e+3	3.5e+3	2.1e+0
33	6.9e+2	1.4e+5	2.0e+2	7.5e+2	7.3e+4	9.7e+1	8.9e+2	3.7e+4	4.2e+1	1.1e+3	2.0e+4	1.8e+1
34	6.9e+2	3.3e+4	4.8e+1	8.5e+2	1.8e+4	2.1e+1	1.2e+3	9.1e+3	7.9e+0	1.6e+3	4.9e+3	3.2e+0

C.3. Critical Difference Diagrams for Different Budget Size

Figures 66–87 present the critical difference diagrams for different budget size. Since SMAC is not compatible with JAHS-Bench-201 and LCBench, we prepare for two setups:

1. the Friedman test using all the optimizers on all the 52 benchmark problems, and
2. the Friedman test using all the optimizers execept SMAC on 18 benchmark problems, which simply excluded LCBench and JAHS-Bench-201 from the benchmark problems.

We tested the hypothesis "The performance are identical on all the optimizers." and each red bar connects all the optimizers that show no significant performance difference. The plots in this section relies on `scikit-posthoc` [1]. The budget size used in the visualizations are $\{T_k^{\max}/2^i\}_{i=0}^{10}$ and $T_k^{\max}$ is defined in Section 6.1. Note that the budgets for random search and HyperBand are ten times more than the other methods, so only random search and HyperBand can improve the performance after $b \geq T_k^{\max}/2^3$. The titles of each figure show the number of workers used in the experiments. Furthermore, "[x.xx]" shows the average rank of each optimizer after using the specified amount of runtime. For example, "BOHB [2.90]" means that BOHB achieved the average rank of 2.90 among all the optimizers after running the specified amount of budget.

[1] `https://scikit-posthocs.readthedocs.io/en/latest/plotting_api.html`.

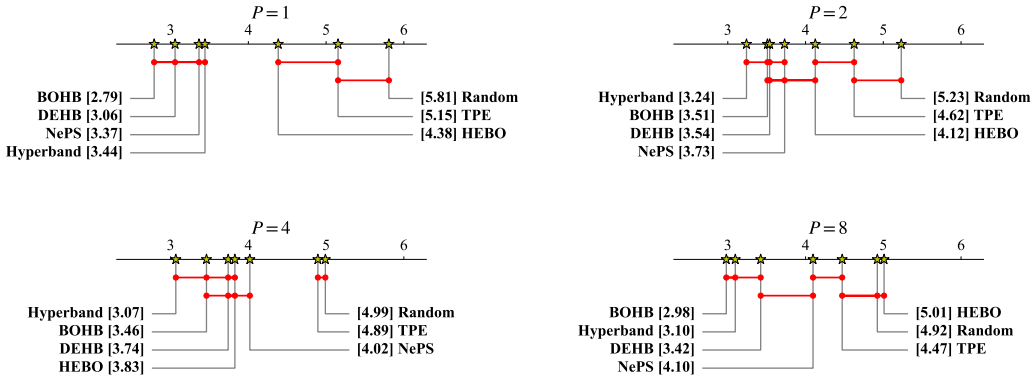


Figure 66.: The critical difference diagrams for the setup without SMAC with the budget of $T_k^{\max}/2^{10}$.

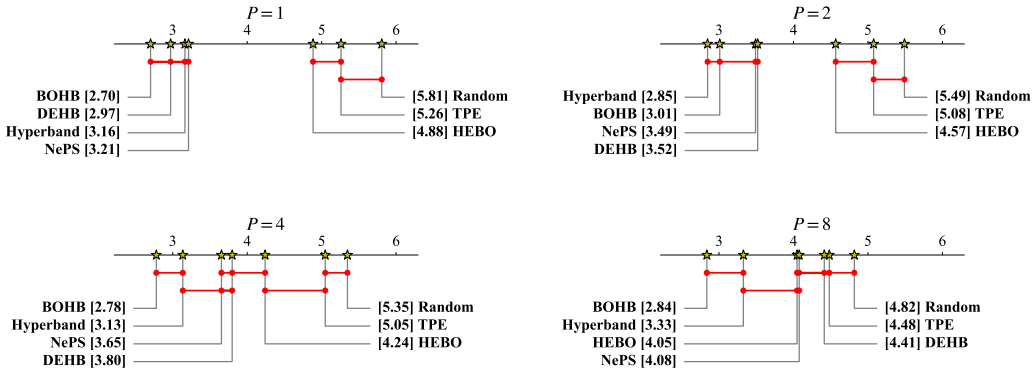


Figure 67.: The critical difference diagrams for the setup without SMAC with the budget of $T_k^{\max}/2^{9}$.

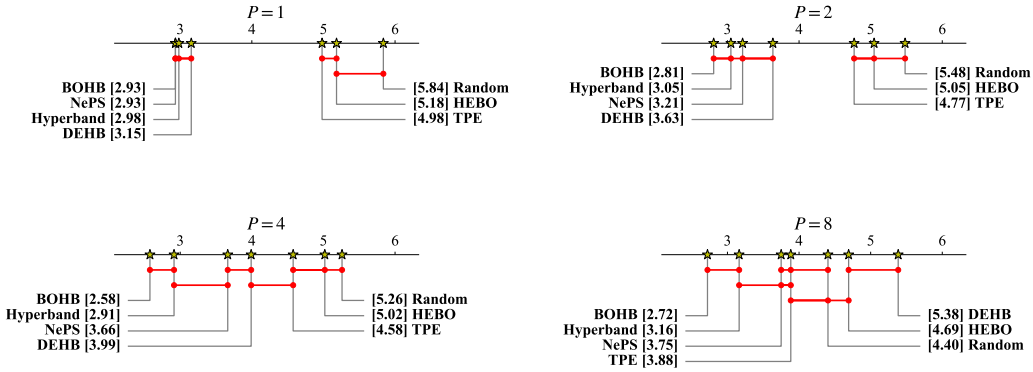


Figure 68.: The critical difference diagrams for the setup without SMAC with the budget of $T_k^{\max}/2^{8}$.

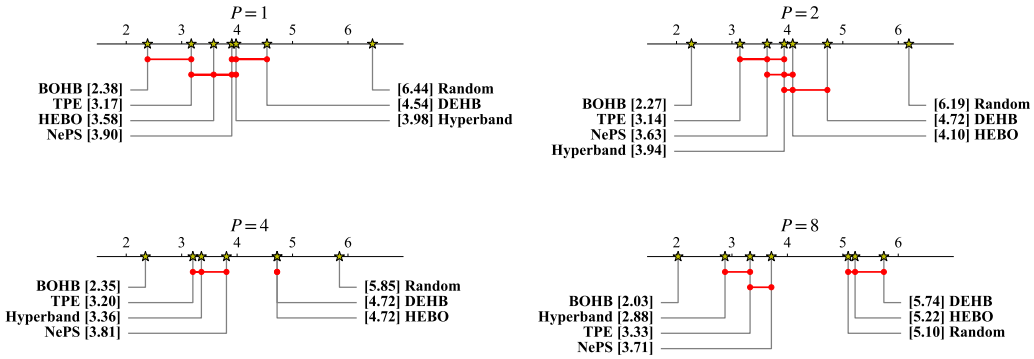


Figure 69.: The critical difference diagrams for the setup without SMAC with the budget of $T_k^{\max}/2^7$.

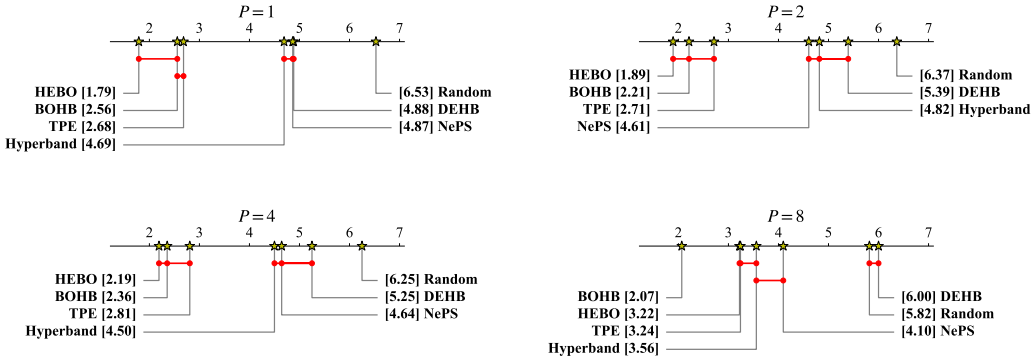


Figure 70.: The critical difference diagrams for the setup without SMAC with the budget of $T_k^{\max}/2^6$.

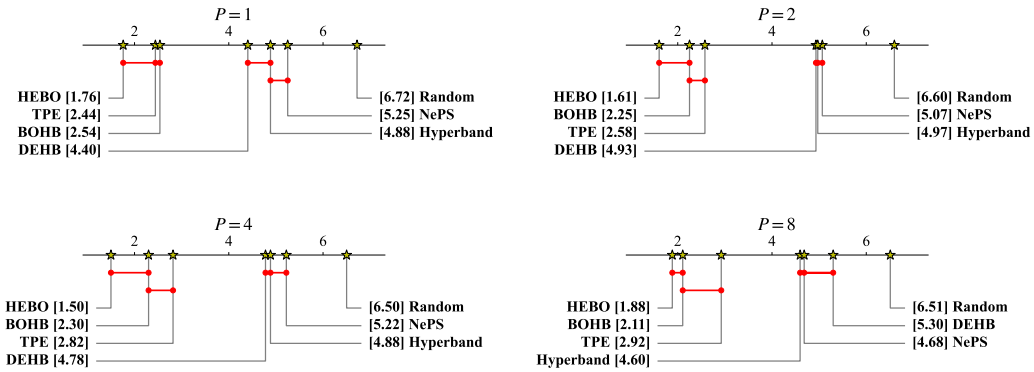


Figure 71.: The critical difference diagrams for the setup without SMAC with the budget of $T_k^{\max}/2^5$.

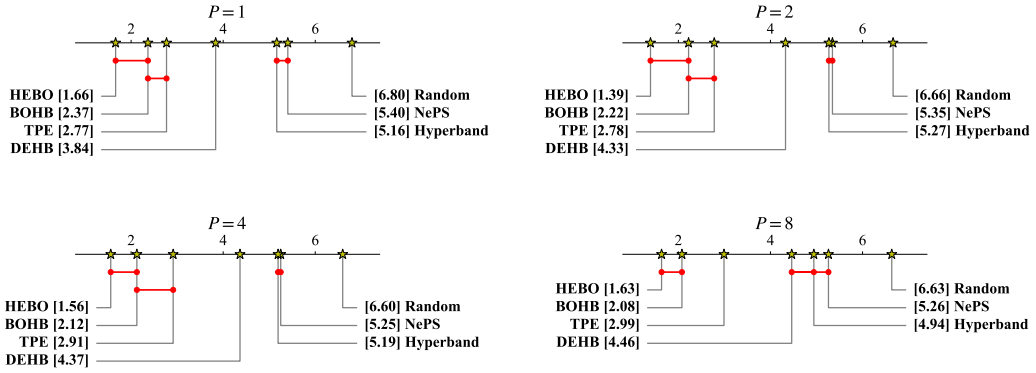


Figure 72.: The critical difference diagrams for the setup without SMAC with the budget of $T_k^{\max}/2^4$.

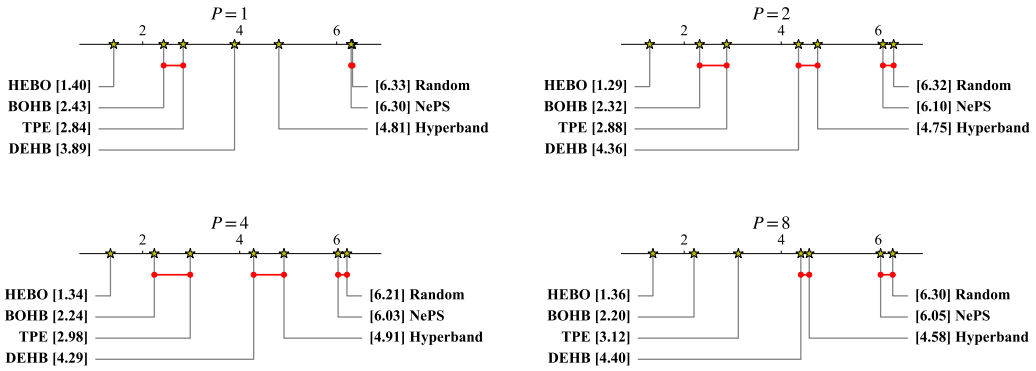


Figure 73.: The critical difference diagrams for the setup without SMAC with the budget of $T_k^{\max}/2^3$.

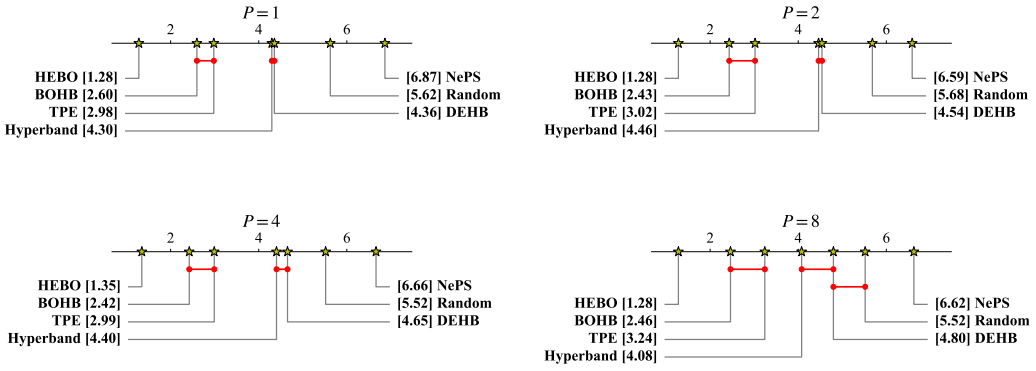


Figure 74.: The critical difference diagrams for the setup without SMAC with the budget of $T_k^{\max}/2^2$.

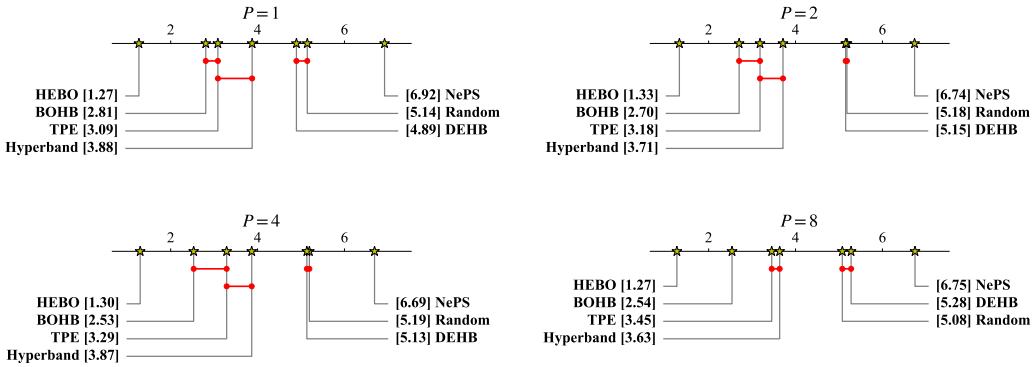


Figure 75.: The critical difference diagrams for the setup without SMAC with the budget of $T_k^{\max}/2$.

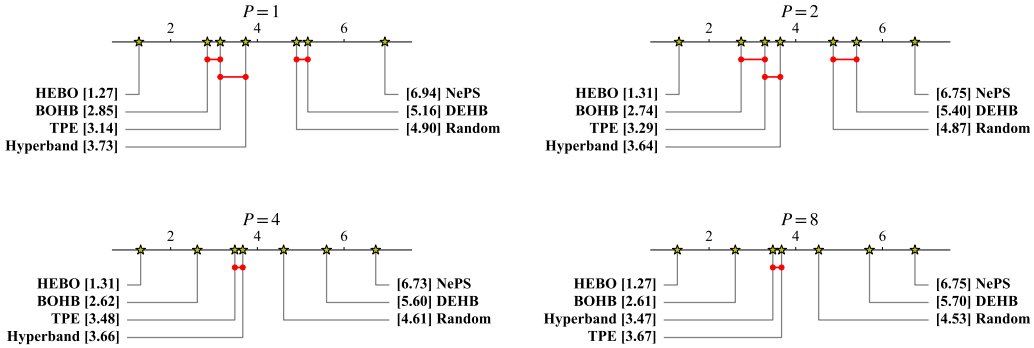


Figure 76.: The critical difference diagrams for the setup without SMAC with the budget of $T_k^{\max}$.

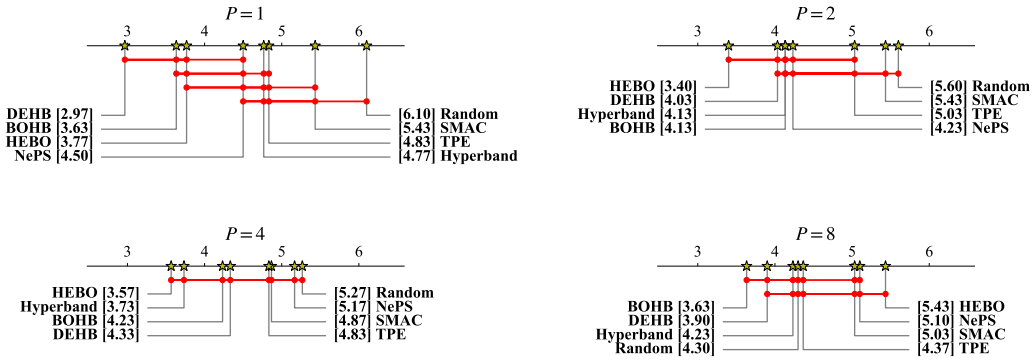


Figure 77.: The critical difference diagrams for the setup with SMAC with the budget of $T_k^{\max}/2^{10}$.

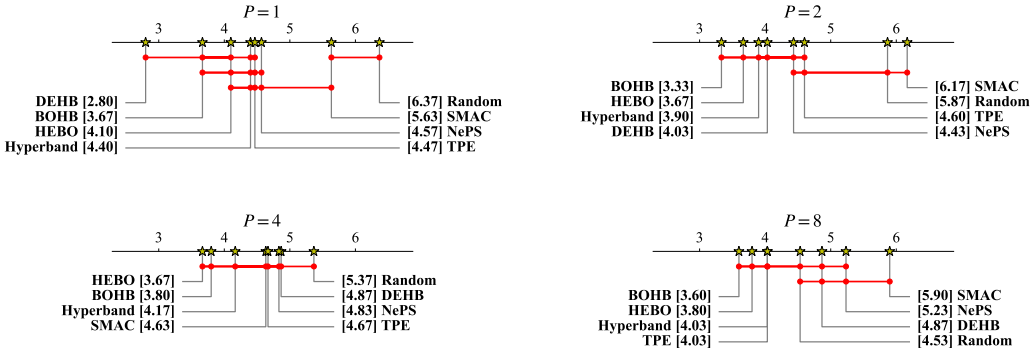


Figure 78.: The critical difference diagrams for the setup with SMAC with the budget of $T_k^{\max}/2^9$.

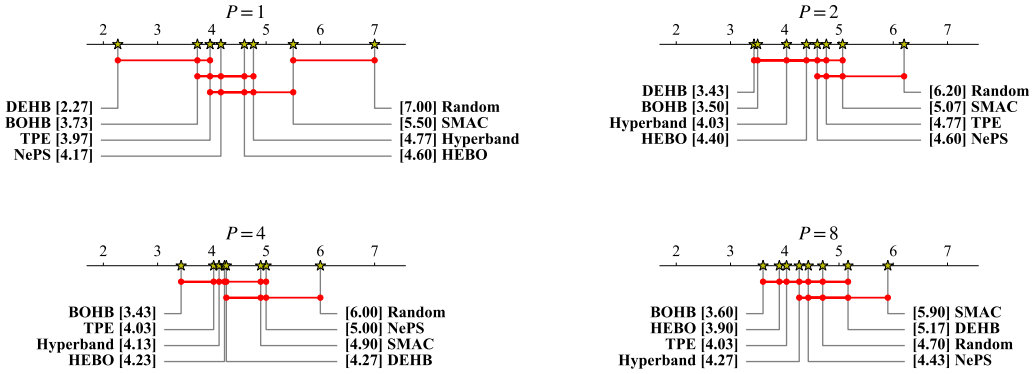


Figure 79.: The critical difference diagrams for the setup with SMAC with the budget of $T_k^{\max}/2^8$.

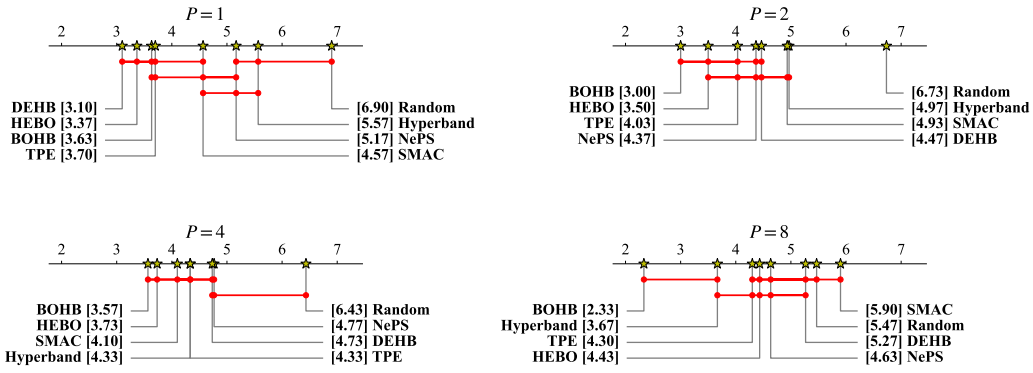


Figure 80.: The critical difference diagrams for the setup with SMAC with the budget of $T_k^{\max}/2^7$.

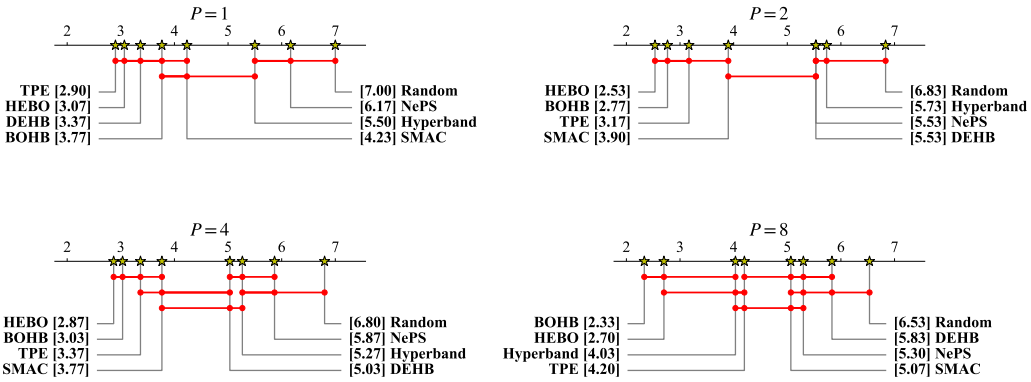


Figure 81.: The critical difference diagrams for the setup with SMAC with the budget of $T_k^{\max}/2^6$.

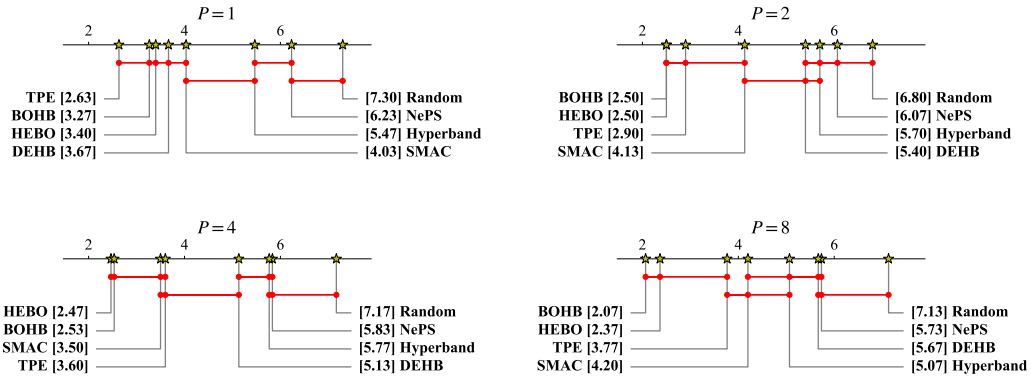


Figure 82.: The critical difference diagrams for the setup with SMAC with the budget of $T_k^{\max}/2^5$.

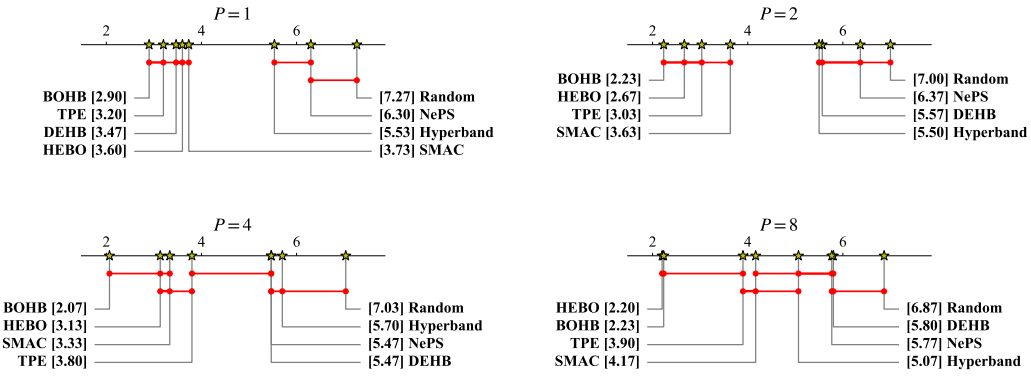


Figure 83.: The critical difference diagrams for the setup with SMAC with the budget of $T_k^{\max}/2^4$.

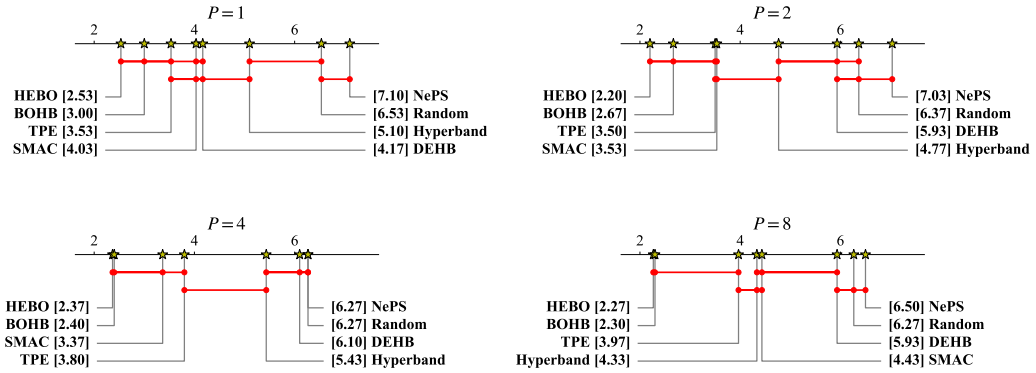


Figure 84.: The critical difference diagrams for the setup with SMAC with the budget of $T_k^{\max}/2^3$.

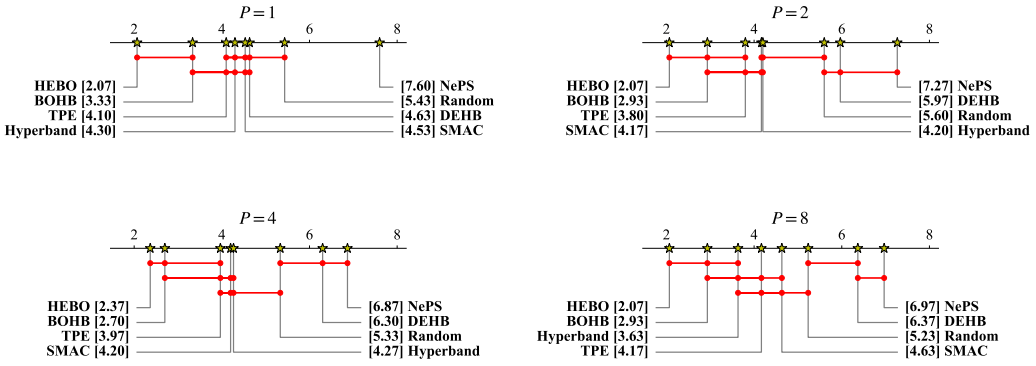


Figure 85.: The critical difference diagrams for the setup with SMAC with the budget of $T_k^{\max}/2^2$.

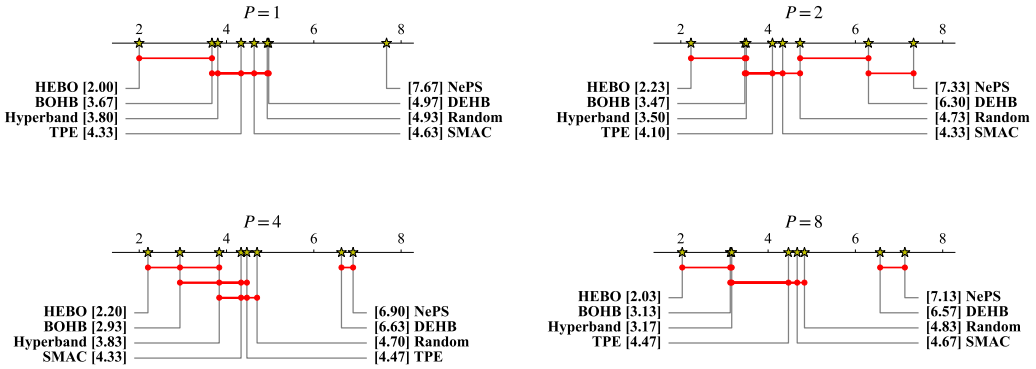


Figure 86.: The critical difference diagrams for the setup with SMAC with the budget of $T_k^{\max}/2$.

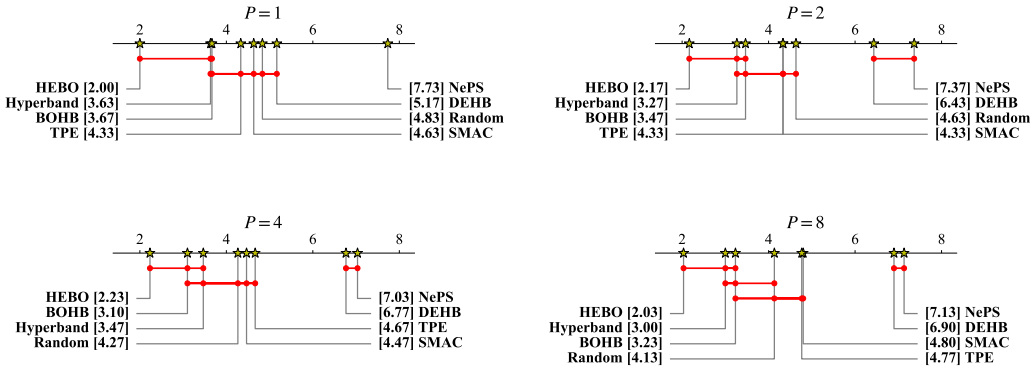


Figure 87.: The critical difference diagrams for the setup with SMAC with the budget of $T_k^{\max}$.