A Probabilistic Machine Learning Approach to Scheduling Parallel Loops with Bayesian Optimization

Preprint · June 2020

CITATIONS
0

READS
165

3 authors:

Khu-rai Kim
Sogang University
6 PUBLICATIONS **5** CITATIONS
SEE PROFILE

Kim Youngjae
POSCO
104 PUBLICATIONS **2,352** CITATIONS
SEE PROFILE

Sungyong Park
Sogang University
72 PUBLICATIONS **222** CITATIONS
SEE PROFILE

A Probabilistic Machine Learning Approach to Scheduling Parallel Loops with Bayesian Optimization

Khu-rai Kim, *Student Member, IEEE,* Youngjae Kim, *Member, IEEE,* and Sungyong Park, *Member, IEEE*

Abstract—In this paper, we propose a new parallel loop scheduling algorithm: *Bayesian optimization augmented factoring self-scheduling* (BO FSS). BO FSS is an automatic self-tuning variant of the factoring self-scheduling (FSS) algorithm. It automatically tunes the internal parameter of FSS by solving an optimization problem using Bayesian optimization (BO), a black-box optimization algorithm. By the nature of BO, our framework only requires execution time measurement of the target loop for tuning. To apply BO, we model the loop execution time with two types of Gaussian process (GP) based models. Notably, our *locality-aware GP* model accelerates the convergence of BO by taking temporal locality effect into account. We implement our method on the GCC implementation of the `OpenMP` standard. Using our implementation, we evaluate the performance of BO FSS against other scheduling algorithms, including recently introduced workload-aware scheduling methods. Also, to quantify our method's performance variation on different workloads (*workload-robustness* in our terms), we use the *minimax regret* metric. According to the proposed metric, BO FSS shows the most robust performance compared to other considered algorithms. Lastly, within the considered workloads, BO FSS improves the execution time of FSS as much as 22% and 5% on average.

Index Terms—Parallel Loop Scheduling, Bayesian Optimization, Parallel Computing, OpenMP

1 Introduction

Loop parallelization is the de-facto standard method for performing shared-memory data-parallel computation. Parallel computing frameworks such as `OpenMP` [1] have enabled the acceleration of advances in many scientific and engineering fields such as astronomical physics [2], climate analytics [3], and machine learning [4]. A major challenge in enabling efficient loop parallelization is to deal with the inherent imbalance in workloads [5]. Under the presence of load imbalance, some computing units (CU) might end up remaining idle for a long time, wasting computational resources. It is thus critical to schedule the tasks to CUs efficiently.

Early on, dynamic loop scheduling algorithms [6], [7], [8], [9], [10], [11], [12] have emerged to attack the parallel loop scheduling problem. However, these algorithms exploit a limited amount of information about the workloads, such as *static imbalance*, resulting in an inconsistency in terms of performance [13]. In our terms, they do not achieve robust performance across a large range of workloads, hence not *workload-robust*. Static imbalance is an imbalance inherent to the workload. Unlike dynamic imbalance, which is an imbalance caused randomly during runtime, static imbalance can sometimes be accurately estimated before execution and can help improve the performance of applications. Workload-aware methods that utilize static imbalance information such as the history-aware self-scheduling (HSS) [14] and the bin-packing longest processing time (BinLPT) [15], [16] algorithms, have recently been introduced. Unfortunately, these methods are inapplicable when the static imbalance information, or a workload-profile, is not provided. It should be noted that many high-performance computing (HPC) applications have workloads where the static imbalance pattern is not known in advance before execution or merely is non-existent. As a result, workload-aware methods can only be applied to a limited range of workloads. Because of the aforementioned limitations in dynamic and workload-aware methods, additional efforts must be made to find an algorithm best-suited for a particular workload. Practitioners need to try out different scheduling algorithms and manually tune them for the best performance, which is tedious and time-consuming.

In this paper, we propose *Bayesian optimization augmented factoring self-scheduling* (BO FSS), a workload-robust parallel loop scheduling algorithm. BO FSS automatically infers properties of the target loop only using execution time measurements of the loop. Since BO FSS doesn't rely on a workload-profile, it is applicable to a wide range of workloads. First, we show that it is possible to achieve robust performance if we are able to appropriately *tune* the internal parameters of a classic scheduling algorithm to each workload individually. Based on this observation, BO FSS tunes the parameter of *factoring self-scheduling* (FSS) [7], a classic dynamic scheduling algorithm, only using execution time measurements of the target loop. This is achieved by solving an optimization problem using a black-box global optimization algorithm called Bayesian optimization (BO) [17]. BO is notable for being data efficient; it requires a minimal number of measurements until convergence [18]. It is also able to handle the presence of noise in the measurements efficiently. These properties lead to successful applications such as compiler optimization flag selection [19], garbage

- *K. Kim is with the Department of Electronics Engineering, Sogang University, Seoul, Republic of Korea. E-mail: msca8h@sogang.ac.kr*
- *Y. Kim and S. Park are with the Department Computer Science and Engineering, Sogang University, Seoul, Republic of Korea. E-mail: {youkim, parksy}@sogang.ac.kr*

This work was supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7080245). This paper was presented in part of the 27th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'19) held in Rennes, France, 2019. (Corresponding author: Sungyong Park.)

collector tuning [20], and cloud configuration selection [18]. By executing the target workload multiple times, our system gradually improves scheduling efficiency.

To apply BO, we need to provide a *surrogate model* that accurately describes the relationship between the scheduling algorithm's parameter, and the resulting execution time. By extending our previous work in [21], we propose two types of surrogate models. First, we model the total execution time contribution of a loop as *Gaussian processes* (GP). Second, for workloads where the loops are executed multiple times, we propose a *locality-aware GP* model. Our locality-aware GP can model the execution time more accurately by incorporating the temporal locality effect using *exponentially decreasing function kernels* [22]. Whenever applicable, our locality-aware construction results in faster convergence.

We implement BO FSS as well as other classic scheduling algorithms such as chunk self-scheduling (CSS) [6], FSS [7], trapezoid self-scheduling (TSS) [8], tapering self-scheduling (TAPER) [10] on the GCC implementation [23] of the `OpenMP` parallelism framework. Then, we evaluate the performance of BO FSS against these classical algorithms and workload-aware methods such as HSS and BinLPT. To quantify and compare the robustness of BO FSS, we adopt the *minimax regret* metric [24], [25]. We select workloads from the Rodinia 3.1 [26] and the GAP [27] benchmark suites for the evaluation. Results show that our method outperforms other scheduling algorithms by improving the execution time of FSS as much as 22% and 5% on average. In terms of workload-robustness, BO FSS achieves the lowest regret of 22.34, which is 21%p less than the method achieving the second-lowest regret.

The key contributions of this paper are as follows:

- **We show that, *when appropriately tuned*, FSS can achieve robust performance** (Section 2). In contrast, the performance of dynamic scheduling and workload-aware methods varies across different workloads.
- **We apply BO to tune the internal parameter of FSS** (Section 3). Results show that our approach achieves good performance consistently across different workloads (Section 5).
- **We propose to model the temporal locality effect of workload using locality-aware GPs** (Section 3.3). Our locality-aware GP incorporates the effect of temporal locality using exponentially decreasing function kernels.
- **We implement BO FSS over the `OpenMP` parallel computing framework** (Section 4). Our implementation includes other classic scheduling algorithms used for the evaluation and is publicly available online.
- **We propose to use *minimax regret* for quantifying workload-robustness of scheduling algorithms** (Section 5). According to the minimax regret criterion, BO FSS shows the most robust performance among considered algorithms.

2 Background and Motivation

In this section, we start by describing the loop scheduling problem. Then, we show that dynamic scheduling and workload-aware methods lack workload robustness. Our analysis is followed by proposing a strategy to solve this problem.

2.1 Background

Parallel loop scheduling. Loops in scientific computing applications are easily parallelizable because of their embarrassingly-data-parallel nature. A parallel loop scheduling algorithm attempts to map each task, or iteration, of a loop to CUs. The most basic scheduling strategy called static scheduling (STATIC) equally divides the tasks (T_i) by the number of CUs in compile time. Usually, a barrier is implied at the end of a loop, forcing all the CUs to wait until all tasks finish computing. If an imbalance is present across the tasks, some CUs may complete computation before other tasks, resulting in many CUs remaining idle. Since execution time variance is abundant in practice because of control-flow divergence and inherent noise in modern computer systems [5], more advanced scheduling schemes are often required.

Dynamic loop scheduling. Dynamic loop scheduling has been introduced to solve the inefficiency caused by execution time variance. In dynamic scheduling schemes, each CU self-assigns a chunk of K tasks in runtime by accessing a central task queue whenever it becomes idle. The queue access causes a small runtime scheduling overhead, denoted by the constant h. The case where $K = 1$ is called self-scheduling (SS) [28]. For SS, we can achieve the minimum amount of load imbalance. However, the amount of scheduling overhead grows proportionally to the number of tasks. Even for small values of h, the total scheduling overhead can quickly become overwhelming. The problem boils down to finding the optimal tradeoff between load imbalance and scheduling overhead. This problem has been mathematically formalized in [6], [29], and a general review of the problem is provided in [30].

2.2 Factoring Self-Scheduling

Among many dynamic scheduling algorithms, we focus on the factoring self-scheduling algorithm (FSS) [7]. Instead of using a constant chunk size K, FSS uses a chunk size that decreases along the loop execution. At the ith *batch*, the size of the next P chunks, K_i, is determined according to

$$R_0 = N, \quad R_{i+1} = R_i - PK_i, \quad K_i = \frac{R_i}{x_i P} \tag{1}$$

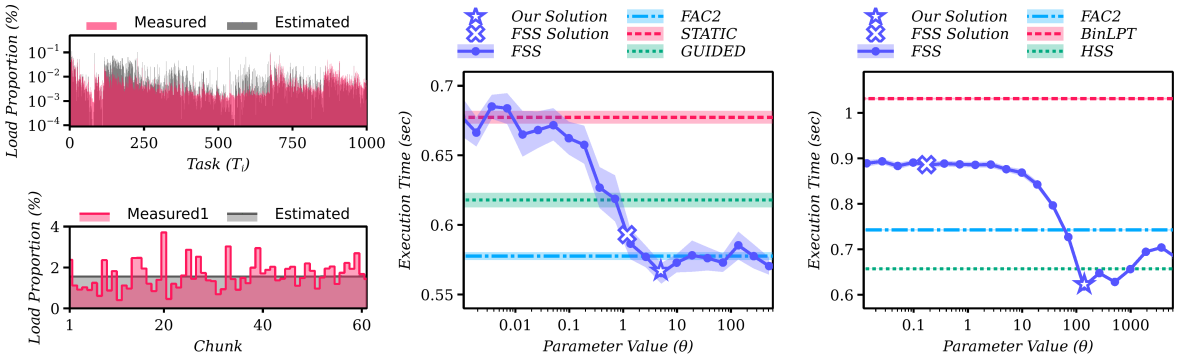
$$b_i = \frac{P}{2\sqrt{R_i}}\,\theta \tag{2}$$

$$x_0 = 1 + b_0^2 + b_0\sqrt{b_0^2 + 4} \tag{3}$$

$$x_i = 2 + b_i^2 + b_i\sqrt{b_i^2 + 4}. \tag{4}$$

where R_i is the number of remaining tasks at the ith batch. The parameter θ in (2) is crucial to the overall performance of FSS. The analysis in [31] indicates that $\theta = \sigma/\mu$ results in the best performance where μ and σ^2 are the mean ($\mathbb{E}[T_i]$) and variance ($\mathbb{V}[T_i]$) of the tasks. However, in Section (2.3), we show that this θ does not always perform well. Instead, we suggest a strategy that determines θ for each workload individually by solving an optimization problem.

The FAC2 scheduling strategy. Since determining μ and σ requires extensive profiling of the workload, the original authors of FSS suggest an unparameterized heuristic version [7]. This version is often abbreviated as FAC2 in the literature. and has been observed to outperform the original FSS [9], [11] despite being a heuristic modification.



(a) Accuracy of load estimation (b) Low static imbalance (c) High static imbalance

Fig. 1. ((a), top) Discrepancy between the workload-profile and actual execution time of the tasks. ((a), bottom) Discrepancy between the load of the chunks created by BinLPT, and their actual execution time. (b-c) Effect of the internal parameter (θ) of FSS on a workload with homogeneous tasks ((b), low static imbalance, `lavaMD` workload) and a workload with non-homogeneous tasks ((c), high static imbalance, `pr-journal` workload). The value of the parameter suggested by the original FSS algorithm is marked with a blue cross, while the actual optimal solution targeted by our proposed method is marked with a blue star. The error bands are the 95% empirical bootstrap confidence intervals of the execution time mean.

2.3 Motivation

Limitations of workload-aware methods. The HSS and BinLPT strategies have significant drawbacks despite being able to fully incorporate the information about load imbalance. First, both the HSS and BinLPT methods require an accurate workload-profile. This is a significant limiting factor since many HPC workloads are comprised of homogeneous tasks where the imbalance is caused dynamically during runtime; there is no static imbalance in the first place. Also, even if a workload-profile is present, it imposes a runtime memory overhead of $O(N)$ for each loop. For large-scale applications where the task count N is huge, the memory overhead is a significant nuisance.

Both the HSS and BinLPT algorithms also have their own caveats. The HSS algorithm has high scheduling overhead [16]. We show in Section 5 that HSS performs well only when high levels of imbalance (such as in the `pr-wiki` workload) are present. The performance of BinLPT algorithm is highly sensitive to the accuracy of the workload-profile. In practice, discrepancies between the true workload and the workload-profile are inevitable. We illustrate this fact using the `pr-journal` graph analytics workload in the upper plot of Figure 1a. For example, we estimate the load of each task using the in-degree of the corresponding vertex in the graph. The grey region is the estimated load of each task, while the red region is the measured load. As shown in the figure, the estimated load does not accurately describe the actual load. Likewise, the *chunks* created by BinLPT using these estimates are equally inaccurate, as shown in the lower plot of Figure 1a. If the number of tasks is minimal, some level of discrepancy may be acceptable. Indeed, the original analysis in [16] considers at most $N = 3074$ tasks. In practice, the number of tasks scales with data leading to a very large N.

Effect of tuning the parameter of FSS Similarly, classical scheduling algorithms such as FSS are not workload-robust [13]. However, we reveal an interesting property by tuning the parameter (θ) of FSS. Figure 1b and Figure 1c illustrate the evaluation results of FSS using the `lavaMD` (a workload with low static imbalance) and `pr-journal` (a workload with high static imbalance) workloads with different values of θ, respectively. The solution suggested in the original FSS algorithm (as discussed in Section 2.2) is denoted by a blue cross. For the `lavaMD` workload (Figure 1b), this solution is arguably close to the optimal value. However, for the `pr-journal` workload (Figure 1c), it leads to poor performance. The original FSS strategy is thus not workload-robust since its performance varies greatly across workloads.

In contrast, by using an optimal value of θ (blue star), FSS outperforms all other algorithms as shown in the plots. Even in Figure 1c where HSS and BinLPT are equipped with an accurate workload-profile, FSS outperforms both methods. This means that tuning the parameter of FSS on a per-workload basis can uncover a new algorithm with robust performance.

Motivational remarks Workload-aware methods and classical dynamic scheduling methods tend to vary in applicability and performance. Meanwhile, classic scheduling algorithms such as FSS achieve optimal performance when they are appropriately tuned to the target workload. This performance potential of FSS points towards the possibility of creating a novel robust scheduling algorithm.

3 Augmenting Factoring Self-Scheduling with Bayesian Optimization

In this section, we describe BO FSS, a self-tuning variant of the FSS algorithm. First, we provide an optimization perspective on the loop scheduling problem. Next, we describe a solution to the optimization problem using BO. Since solving our problem requires modeling of the execution time using surrogate models, we describe two ways to construct surrogate models.

3.1 Scheduling as an Optimization Problem

The main idea of our proposed method is to design an optimal scheduling algorithm by finding its optimal configurations based on execution time measurements. First, we define a set of scheduling algorithms $\mathcal{S} = \{S_{\theta_1}, S_{\theta_2}, \ldots\}$ indexed by a tunable parameter θ. In our case, $\mathcal{S}$ is the set of configurations of the FSS algorithm with the parameter θ discussed in Section 2.2. Within this set of configurations, we choose the *optimal configuration* that minimizes the mean of the total execution time contribution (T_{total}) of a parallel loop.

Algorithm 1: Bayesian optimization

Initial dataset $\mathcal{D}_0 = \{(\theta_0, \tau_0), \ldots, (\theta_N, \tau_N)\}$ **for** $t \in [1, T]$ **do**
1. Fit surrogate model $\mathcal{M}$ using $\mathcal{D}_t$.
2. Solve inner optimization problem. $\theta_{t+1} = \arg\max_\theta \alpha(\theta|\mathcal{M}, \mathcal{D}_t)$
3. Evaluate parameter. $\tau_{t+1} \sim T_{\text{total}}(S_{\theta_{t+1}})$
4. Update dataset. $\mathcal{D}_{t+1} = \mathcal{D}_{t+1} \cup (\theta, \tau)$

end

This problem is now of the form of an optimization problem denoted as,

$$\underset{\theta}{\text{minimize}} \quad \mathbb{E}[\, T_{total}(S_\theta)\,]. \tag{5}$$

Problem structure. Now that the optimization problem is formulated, we are now supposed to apply an optimization solver. However, this optimization problem is ill-formed, prohibiting the use of any typical solver. First, the objective function is noisy because of the inherent noise in computer systems. Second, we do not have enough knowledge about the structure of T. Different workloads interact differently with scheduling algorithms [13]. It is thus difficult to obtain an analytic model of T that is universally accurate. Moreover, most conventional optimization algorithms require knowledge about the gradient $\nabla_\theta T$, which we do not have.

Solution using Bayesian optimization. For solving this problem, we leverage *Bayesian Optimization* (BO). We initially attempt to apply other gradient-free optimization methods such as *stochastic approximation* [32]. However, the noise level in execution time is so extreme that most gradient-based methods fail to converge. Conveniently, BO has recently been shown to be effective for solving such kind of optimization problems [18], [19], [20]. Compared to other black-box optimization methods, BO requires less objective function evaluations and can handle the presence of noise well [18].

Description of Bayesian optimization. The overall flow of BO is shown in Algorithm 1. First, we build a surrogate model $\mathcal{M}$ of T_{total}. Let (θ, τ) denote a *data point* of an observation where θ is a parameter value, and τ is the resulting execution time measurement such that $\tau \sim T_{\text{total}}$. Based on a dataset of previous observations denoted as $\mathcal{D}_t = \{(\theta_1, \tau_1), \ldots, (\theta_t, \tau_t)\}$, a surrogate model provides a prediction of $T_{\text{total}}(\theta)$ and the uncertainty of the prediction. In our context, the prediction and uncertainty are given as the mean of the predictive distribution ($\mu(\theta \,|\, \mathcal{D}_t)$) and its variance ($\sigma^2(\theta \,|\, \mathcal{D}_t)$).

Using $\mathcal{M}$, we now solve what is known as the *inner optimization problem*. In this step, we choose to *exploit* our current knowledge about the optimal value or *explore* entirely new values that we have yet tried out. In the extremes, minimizing $\mu(\theta \,|\, \mathcal{D}_t)$ gives us the optimal parameter *given our current knowledge*, while minimizing $\sigma^2(\theta \,|\, \mathcal{D}_t)$ gives us the parameter we are currently the most uncertain. The solution is given by a tradeoff of the two ends (often called the exploration-exploitation tradeoff), found by solving the optimization problem

$$\theta_{i+1} = \arg\max_\theta \, \alpha(\theta \,|\, \mathcal{M}, \mathcal{D}_t) \tag{6}$$

where the function α is called the *acquisition function*. Based on the predictions and uncertainty estimates of $\mathcal{M}$, α returns our utility of trying out a specific value of θ. Evidently, the quality of the prediction and uncertainty estimates of $\mathcal{M}$ are crucial to the overall performance. By maximizing α, we obtain the parameter value that has the highest utility, according to α. In this work, we use the *max-value entropy search* (MES) [33] acquisition function. After solving the inner optimization problem, we obtain the next value to try out, θ_{t+1}. We can then try out this parameter and append the result $(\theta_{t+1}, \tau_{t+1})$ to the dataset. For a comprehensive review of BO, please refer to [17]. We will later explain our `OpenMP` framework implementation of this overall procedure in Section 4.

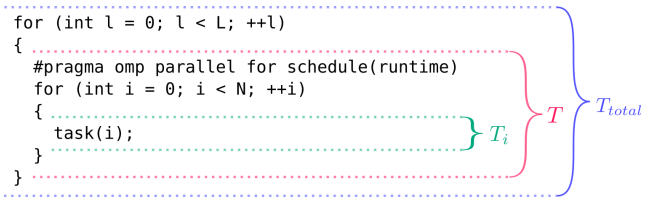


Fig. 2. Visualization of our execution time models. The execution time of the parallel loop (red bracket) is denoted as T, while the execution time of the tasks in the parallel loop (green bracket) is denoted as T_i. The outer loop (blue bracket) represents repeated execution (L times) of the parallel loop within the application, where T_{total} is the total execution time contribution of the loop.

3.2 Modeling Execution Time with Gaussian Processes

As previously stated, having a good surrogate model $\mathcal{M}$ is essential. Modeling the execution time of parallel programs has been a classic problem in the field of performance modeling. It is known that parallel programs tend to follow a Gaussian distribution when the execution time variance is not very high [34]. This result follows from the *central limit theorem* (CLT), which states that the influence of multiple *i.i.d.* noise sources asymptotically form a Gaussian distribution. Considering this, we model the total execution time contribution of a loop as

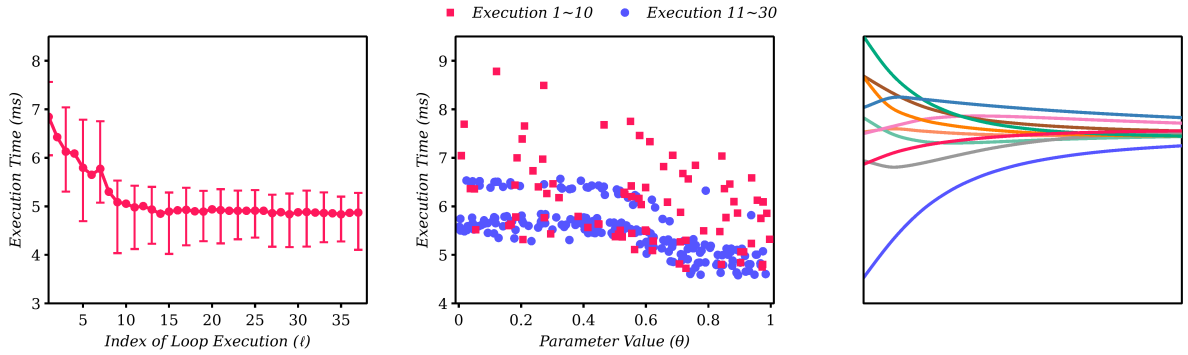
$$T_{total} = \sum_{\ell=1}^{L} T(S_\theta) + \epsilon \tag{7}$$

where L is the total number of times a specific loop is executed within the application, indexed by ℓ. Following the conclusions of [34], we naturally assume that ϵ follows a Gaussian distribution. Note that, at this point, we assume T is independent of the index ℓ. For an illustration of the models used in our discussion, please see Figure 3.

Gaussian Process formulation. From the dataset $\mathcal{D}_t$, we infer the model of the execution time $T_{total}(\theta)$ using *Gaussian processes* (GPs). A GP is a nonparametric Bayesian probabilistic machine learning model for nonlinear regression. Unlike parametric models such as polynomial curve fitting and random forest, GPs automatically tune their complexity based on data [35]. Also, more importantly, GPs can naturally incorporate the assumption of additive noise (such as ϵ in (7)). The prediction of a GP is given as a univariate Gaussian distribution fully described by its mean ($\mu(x|\mathcal{D}_t)$) and variance ($\sigma^2(x|\mathcal{D}_t)$). These are computed in a closed form as

$$\mu(\theta|\mathcal{D}_t) = \mathbf{k}(\theta)^T (\mathbf{K} + \sigma_n^2 I)^{-1} \mathbf{y} \tag{8}$$

$$\sigma^2(\theta|\mathcal{D}_t) = k(\theta, \theta) - \mathbf{k}(\theta)^T (\mathbf{K} + \sigma_\epsilon^2 I)^{-1} \mathbf{k}(\theta) \tag{9}$$



(a) Locality effect ℓ-axis view (b) Locality effect θ-axis view (c) Samples from the Exp. kernel

Fig. 3. (a) (b) Visualization of the temporal locality effect on the execution time of the `kmeans` workload. (a) ℓ-axis view. The error bars are the 95% empirical confidence intervals. (b) θ-axis view. The red squares are measurements of the earlier executions ($\ell \leq 10$) while the blue circles are measurements of the later executions ($\ell > 10$). (c) Randomly sampled functions from a GP prior with an exponentially decreasing function kernel.

where $\mathbf{y} = [\tau_1, \tau_2, \ldots, \tau_t]$, $\mathbf{k}(\theta)$ is a vector valued function such that $[\mathbf{k}(\theta)]_i = k(\theta, \theta_i),\ \forall \theta_i \in \mathcal{D}_t$, and $\mathbf{K}$ is the Gram matrix such that $[\mathbf{K}]_{i,j} = k(\theta_i, \theta_j),\ \forall \theta_i, \theta_j \in \mathcal{D}_t$; $k(x, y)$ denotes the *covariance kernel* function which is a design choice. We use the *Matern 5/2* kernel which is computed as,

$$k(x, x'; \sigma^2, \rho^2) = \sigma^2 \left(1 + \sqrt{5}r + \frac{5}{3}r^2\right) \exp(-\sqrt{5}r) \quad (10)$$

$$\text{where } r = ||x - x'||_2/\rho^2. \quad (11)$$

For a detailed introduction to GP regression, please refer to [36].

Non-Gaussian noise. Despite the remarks in [34] saying that parallel programs mostly follow Gaussian distributions, we experience cases where the execution time of individual parallel loops does not quite follow a Gaussian distribution. For example, occasional L2, L3 cache-misses results in large deviations, or *outliers*, in execution time. To correctly model these events, it is advisable to use heavy-tail distributions such as the Student-T. Methods for dealing with such outliers are described in [37] and [38]. However, to narrow the scope of our discussion, we stay within the Gaussian assumption.

3.3 Modeling with Locality-Aware Gaussian Processes

Until now, we only consider acquiring samples of T_{total} by summing our measurements of T. For the case where the parallel loop in question is executed more than once (that is, $L > 1$), we acquire L observations of T in a single run of the workload. By exploiting our model's structure in (7), it is possible to utilize all L samples instead of aggregating them into a single one. Since the Gaussian distribution is additive, we can decompose the distribution of T_{total} such that

$$T_{total} = \sum_{\ell=1}^{L} T(S_\theta, \ell) \quad (12)$$

$$\sim \sum_{\ell=1}^{L} \mathcal{N}(\mathbb{E}[T(S_\theta, \ell)], \mathbb{V}[T(S_\theta, \ell)],) \quad (13)$$

$$= \mathcal{N}(\sum_{\ell=1}^{L} \mathbb{E}[T(S_\theta, \ell)], \sum_{\ell=1}^{L} \mathbb{V}[T(S_\theta, \ell)]) \quad (14)$$

$$\approx \mathcal{N}(\sum_{\ell=1}^{L} \mu(\theta, \ell \,|\, \mathcal{D}_t), \sum_{\ell=1}^{L} \sigma^2(\theta, \ell \,|\, \mathcal{D}_t)). \quad (15)$$

Note the dependence of T on the index of execution ℓ. From (14), we can retrieve T_{total} from the mean ($\mathbb{E}[T(S_\theta, \ell)]$) and variance ($\mathbb{V}[T(S_\theta, \ell)]$) estimates of T, which are given by modeling T using GPs as denoted in (15).

Temporal locality effect. However, this is not as simple as assuming that all L measurements of T are independent (ignoring the argument ℓ of T). The execution time distribution of a loop changes dramatically within a single application run because of the temporal locality effect. This is shown in Figure 3 using measurements of a loop in the `kmeans` benchmark. In Figure 3a, it is clear that earlier executions of the loop ($\ell \leq 10$) are much longer than the later executions ($\ell > 10$). Also, different moments of executions are effected differently by θ, as shown in Figure 3b. It is thus necessary to accurately model the effect of ℓ to better distinguish the effect of θ.

Exponentially decreasing function kernel. To model the temporal locality effect, we expand our GP model to include the index of execution ℓ. Now, the model is a 2-dimensional GP receiving ℓ and θ. Within the workloads we consider, the temporal locality effect is shown an exponentially decreasing tendency. We thus assume that the locality effect can be represented with *exponentially decreasing functions* (Exp.) of the form of $e^{-\lambda \ell}$. The kernel for these functions has been introduced in [22] for modeling the learning curves of machine learning algorithms. The exponentially decreasing function kernel is computed as

$$k(\ell, \ell') = \frac{\beta^\alpha}{(\ell + \ell' + \beta)^\alpha}. \quad (16)$$

Random functions sampled from the space induced by the Exp. kernel are shown in Figure 3c. Notice the similarity of the sampled functions and the visualized locality effect in Figure 3a. Modeling more complex locality effects such as periodicity can be achieved by combining more different kernels. An automatic procedure for doing this is described in [39].

Kernel of locality-aware GPs. Since the sum of covariance kernels is also a valid covariance kernel [36], we define our 2-dimensional kernel as

$$k(x, x') = k_{\text{Matern}}(\theta, \theta') + k_{\text{Exp}}(\ell, \ell') \quad (17)$$

$$\textit{where } x = [\theta, \ell],\ x' = [\theta', \ell']. \quad (18)$$

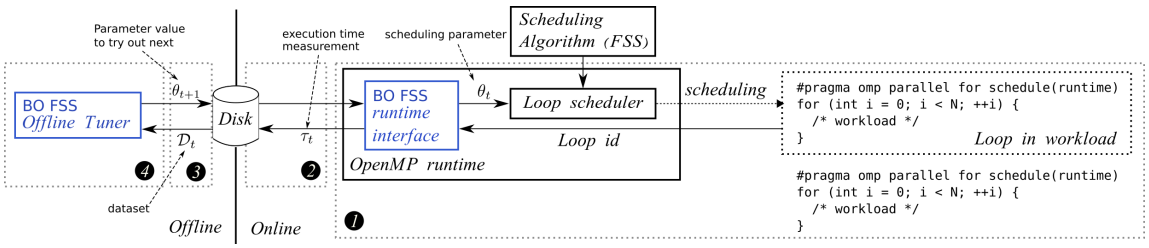


Fig. 4. System overview of BO FSS. *Online* denotes the time we are actually executing the workload, while *offline* denotes the time we are not executing the workload. For a detailed description, refer to the text in Section 4.

Intuitively, this definition implies that we assume the effect of scheduling (resulting from θ) and locality (resulting from ℓ) to be superimposed (additive).

Reducing computational cost. The computational complexity of computing a GP is in $O(T^3)$ where T represents the total number of BO iterations. The locality aware construction uses all the independent loop executions resulting in computational complexity in $O((LT)^3)$. To reduce the computational cost, we subsample data along the axis of ℓ by using every kth measurement of the loop, such that $\ell \in \{1, k+1, 2k+1, \dots, L\}$. As a result, the computational complexity is reduced by a constant factor such that $O((\frac{L}{k}T)^3)$. In all of our experiments, we use a large value of k so that $L/k = 4$.

3.4 Treatment of Gaussian Process Hyperparameters

GPs have multiple hyperparameters that need to be predetermined. The suitability of these hyperparameters is directly related to the optimization performance of BO [40]. Unfortunately, whether a set of hyperparameters is appropriate depends on the characteristics of the workload. Since real-life workloads are very diverse, it is thus essential to automatically handle these parameters. The Matern 5/2 kernel has two hyperparameters ρ and σ, while the Exp. kernel has two hyperparameters α and β. GPs also have hyperparameters themselves, the function mean μ and the noise variance σ_ϵ^2. We denote the hyperparameters using the concatenation $\phi = [\mu, \sigma_\epsilon, \sigma, \rho, \dots]$.

Since the marginal likelihood $p(\mathcal{D}_t|\phi)$ is available in a closed form [36], we can infer the hyperparameters using maximum likelihood estimation type-II or the *fully Bayesian treatment*. The fully Bayesian treatment has been empirically shown to give better optimization performance in the context of BO [40], [41]. It is performed by approximating the integral

$$\alpha(x \mid \mathcal{M}, \mathcal{D}_t) = \int \alpha(x \mid \mathcal{M}, \phi, \mathcal{D}_t) p(\phi|\mathcal{D}_t) d\theta \tag{19}$$

$$\approx \frac{1}{N} \sum_{\phi_i \sim p(\phi|\mathcal{D}_t)} \alpha(x \mid \mathcal{M}, \phi_i, \mathcal{D}_t), \tag{20}$$

using samples from the posterior ϕ_i where N is the number of samples. For sampling from the posterior, we use the no-u-turn sampler (NUTS, [42]).

4 System Implementation

We now describe our implementation of BO FSS[1]. Our implementation is based on the GCC implementation of the `OpenMP` 4.5 framework [1]. An illustration of BO FSS is shown in Figure 4. The overall workflow is as follows:

0. First, we randomly generate initial scheduling parameters $\theta_0, \dots, \theta_{N_0}$ using a Sobol quasi-random sequence [43].
1. During execution, for each loop in the workload, we schedule the parallel loop using the parameter θ_t. We measure the resulting execution time of the loop and acquire a measurement τ_t.
2. Once we finish executing the workload, store the pair (θ_t, τ_t) to disk in `JSON` format.
3. Then, we run the offline tuner, which loads the dataset $\mathcal{D}_t$ from disk.
4. Using this dataset, we solve the inner optimization problem in (6), acquiring the next scheduling configuration θ_{t+1}.
5. At the subsequent execution of the workload, $t \leftarrow t+1$, and go back to 1.

Offline means the time we finish executing the workload, while *online* means the time we are executing the workload (runtime).

Implementation of the offline tuner. We implement the offline tuner as a separate program written in Julia [44], invoked by the user. When invoked, the tuner solves the inner optimization problem, and stores the results in disk. For solving the inner optimization problem, we use the *DIRECT* algorithm [45] implemented in the NLopt library [46]. For marginalizing the GP hyperparameters, we use the `AdvancedHMC.jl` implementation of NUTS [47].

Search space reparameterization. BO requires the domain of the parameter to be bounded. However, in the case of FSS, θ is not necessarily bounded. As a compromise, we reparameterized θ into a fixed domain such that

$$\underset{x}{\text{minimize}} \quad \mathbb{E}[\, T_{total}(S_{\theta(x)}) \,] \tag{21}$$

$$\textit{where} \quad \theta(x) = 2^{19\,x-10}, \; 0 < x < 1. \tag{22}$$

This also effectively converts the search space to be in a logarithmic scale. The reparameterized domain is chosen by empirically investigating feasible values of θ.

User interface. BO FSS can be selected by setting the `OMP_SCHEDULE` environment variable, or by the `OpenMP` runtime API as in Listing 1.

Listing 1
Selecting a scheduling algorithm

```
omp_set_schedule(BO_FSS); // selects BO FSS
```

1. Source code is available in https://github.com/Red-Portal/bosched

TABLE 1
Benchmark Workloads

Suite	Workload Profile	Characterization	# Tasks (N)	Application Domain	Benchmark Suite
`lavaMD`	Uniformative [1]	N-Body	8000	Molecular Dynamics	Rodinia 3.1
`stream.`	No	Dense Linear Algebra	65536	Data Mining	Rodinia 3.1
`kmeans`	Uniformative [2]	Dense Linear Algebra	494020	Data Mining	Rodinia 3.1
`srad v1`	Uniformative [1]	Structured Grid	229916	Image Processing	Rodinia 3.1
`nn`	Uniformative [1]	Dense Linear Algebra	8192	Data Mining	Rodinia 3.1
`cc-*`	Yes	Sparse Linear Algebra	N/A [3]	Graph Analytics	GAP
`pr-*`	Yes	Sparse Linear Algebra	N/A [3]	Graph Analytics	GAP

[1] Uniformly partitioned workload.
[2] Imbalance present only in domain boundaries.
[3] Input data dependent; number of vertices of the input graph.

Modification of the OpenMP ABI. As previously described, our system optimizes each loop in the workload independently. Naturally, our system requires the identification of the individual loops within the `OpenMP` runtime. However, we encounter a major issue: the current `OpenMP` ABI does not provide a way for such identification. Consequently, we have to modify the GCC 8.2 [23] compiler's `OpenMP` code generation and the `OpenMP` ABI. The modified GCC `OpenMP` ABI is shown in Listing 2. During compilation, a unique token for each loop is generated and inserted at the end of the `OpenMP` procedure calls. Using this token, we store and manage the state of each loop. Measuring the loop execution time is done by starting the system clock in `OpenMP` runtime entries such as `GOMP_parallel_runtime_start`, and stopping in exits such as `GOMP_parallel_end`.

Listing 2
Modified GCC OpenMP ABI

```
void GOMP_parallel_loop_runtime(void (*fn) (void *), void *
    data, unsigned num_threads, long start, long end, long
     incr, unsigned flags, size_t loop_id)
void GOMP_parallel_runtime_start(long start, long end, long
     incr, long *istart, long *iend, size_t loop_id)
void GOMP_parallel_end(size_t loop_id)
```

5 Evaluation

In this section, we first describe the overall setup of our experiments. Then, we compare the robustness of BO FSS against other scheduling algorithms. After that, we evaluate the performance of our BO augmentation scheme. Lastly, we directly compare the execution time.

5.1 Experimental Setup

System setup. All experiments are conducted on a single shared-memory computer with an AMD Ryzen Threadripper 1950X 3.4GHz CPU which has 16 cores and 32 threads with simultaneous multithreading enabled. It also has 1.5MB of L1 cache, 8MB of L2 cache and 32MB of last level cache. We use the Linux 5.4.36-lts kernel with two 16GB DDR4 RAM (32GB total). Frequency scaling is disabled with the `cpupower frequency-set performance` setting. We use the GCC 8.3 compiler with the `-O3`, `-march=native` optimization flags enabled in all of our benchmarks.

BO FSS setup. We run BO FSS for 20 iterations starting from 4 random initial points. All results use the best parameter found after the aforementioned number of iterations.

Baseline scheduling algorithms. We compare BO FSS against the FSS [7], CSS [6], TSS [8], GUIDED [48], TAPER [10], BinLPT [16], HSS [14] algorithms. Apart from the HSS and BinLPT algorithms, the details of the implementation of CSS, TSS and TAPER tend to vary. We organized the details of our implementations of these algorithms in Table 4, which is at the end of our paper. The implementation details of FSS has already been shown in Section 2.2. We use the implementation of BinLPT and HSS provided by the authors of BinLPT[2]. For the FSS and CSS algorithms, we estimate the statistics of each workloads (μ, σ) beforehand from 64 executions. The scheduling overhead parameter h is estimated using the method described in [49]. We use the default STATIC and GUIDED implementations of the `OpenMP` 4.5 framework using the `static` and `guided` scheduling flags. For the TSS and TAPER schedules, we follow the heuristic versions suggested in their original works, denoted as TRAP1 and TAPER3, respectively.

Benchmark workloads. The workloads considered in our experiments are summarized in Table 1. We select workloads from the Rodinia 3.1 benchmark suite [26] (`lavamd`, `streamcluster`, `kmeans`, `srad v1`) where the STATIC scheduling method performs worse than other dynamic scheduling methods. We also include workloads from the GAP benchmark suite [27] (`cc`, `pr`) where the load is predictable from the input graph.

Workload-profile availability. We characterize the workload-profile availability of each workload in the *Workload-Profile* column in Table 1. For workloads with homogeneous tasks (`lavaMD`, `stream.`, `srad v1`, `nn`), static imbalance does not exist. Most of the imbalance is caused during runtime, deeming a workload-profile uniformative. On the other hand, the static imbalance of the `kmeans` workload is revealed during execution, not before execution. We thus consider the workload-profile to be effectively unavailable.

TABLE 2
Input Graph Datasets

Dataset	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\mathbf{deg}^-(\mathbf{v})$[1], $\mathbf{deg}^+(\mathbf{v})$[2]		
			mean	**std**	**max**
journal [50]	4.0M	69.36M	17, 17	43, 43	15k, 15k
wiki [51]	3.57M	45.01M	13, 13	33, 250	7k, 187k
road [52]	24.95M	57.71M	2, 2	1, 1	9, 9
skitter [53]	1.70M	22.19M	13, 13	137, 137	35k, 35k

[1] In-degree of each vertex.
[2] Out-degree of each vertex.

Input graph datasets. We organize the graph datasets used for the workloads from the GAP benchmark suite in Table 2, acquired from [54]. $|\mathcal{V}|$ and $|\mathcal{E}|$ are the vertices and edges in

2. Retrieved from https://github.com/lapesd/libgomp

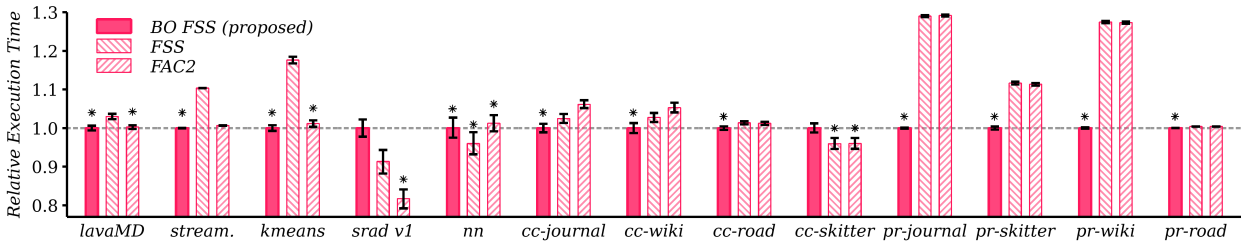


Fig. 5. Execution time comparison of BO FSS, FSS and FAC2. We estimate the mean execution time from 256 executions. The error bars show the 95% bootstrap confidence intervals. The results are normalized by the mean execution time of BO FSS. The methods with the lowest execution time are marked with a star (*). Methods *not* significantly different with the best performing method are also marked with a star (Wilcoxon signed rank test, 1% null-hypothesis rejection threshold).

the graphs, respectively. The load of each task (T_i) in the `cc` and `pr` workloads is proportional to the in-degree and out-degree of each vertex [55], respectively. We use this degree information for the workload-profile. Among the datasets considered, `wiki` has the most extreme imbalance while `road` has the least imbalance [55].

Workload-robustness measure. To quantify the notion of workload-robustness, we use the *minimax regret* measure [25]. The minimax regret quantifies robustness by calculating the opportunity cost using an algorithm, computed as

$$\mathcal{R}(S,w) = \frac{U(S,w) - \min_{S\in\mathcal{S}} U(S,w)}{\min_{S\in\mathcal{S}} U(S,w)} \times 100 \quad (23)$$

$$\mathcal{R}(S) = \max_{w\in\mathcal{W}} \mathcal{R}(S,w) \quad (24)$$

where $U(S,w)$ is the utility of the scheduling algorithm S on the workload w, and $\mathcal{W}$ is our set of workloads. We choose $U_w(S,w)$ to be the relelative execution time normalized by that of the best performing algorithm. Note that among different robustness measures, the minimax regret is very pessimistic [24], emphasizing the worst-case performance.

5.2 Evaluation of Workload-Robustness

Table 3 compares the minimax regrets of different scheduling algorithms with that of BO FSS. Each entry in the table is the regret subject to the workload and scheduling algorithm ($\mathcal{R}(S,w)$). The final row is the regret subject to the scheduling algorithm ($\mathcal{R}(S)$). BO FSS achieves the lowest overall regret of 22%p. In contrast, both static and dynamic scheduling methods achieve similar level of regret. This observation is on track with the previous findings [13]; none of the classic scheduling methods dominate each other.

Remember that we select workloads where STATIC performs poorly. Our robustness analysis thus only holds for comparing dynamic and workload-aware scheduling methods.

Remarks. The results for workload-robustness using the minimax regret metric show that BO FSS achieves significantly lower levels of regret compared to other scheduling methods. As a result, BO FSS performs consistently well. Even when BO FSS does not perform the best, its performance is within an acceptable range.

5.3 Evaluation of Bayesian Optimization Augmentation

A fundamental part of the proposed method is that BO FSS improves the performance of FSS by tuning its internal

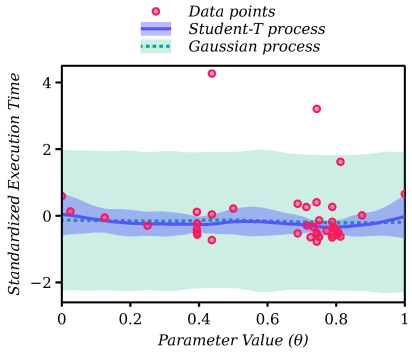


Fig. 6. Parameter space and surrogate model fit on the `srad v1` workload. The colored regions are the 95% predictive confidence intervals of the GP (green region) and Student-T process (blue region). The red circles are the data points used to fit both surrogate models.

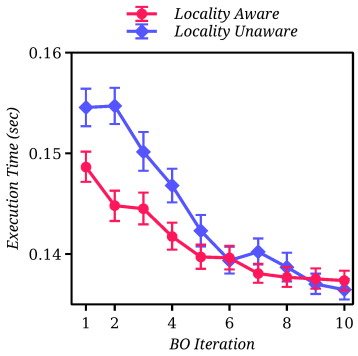


Fig. 7. Convergence plot of the locality-unaware GP and the locality-aware GP on the `skitter` workload. We ran 10 iterations BO 30 times from beginning to end, and computed the 95% boostrap confidence intervals.

parameter. In this section, we show whether our BO augmentation improves the performance of FSS compared to those of its original counterpart and its heuristic variant FAC2. We run BO FSS, FSS, and FAC2 on workloads with both high and low static imbalances. The results are shown in Figure 5. Overall, we can see that BO FSS consistently outperforms FSS and FAC2 with the exception of `srad v1` and `cc-skitter`. On workloads with high imbalance such as `pr-journal` and `pr-wiki`, the execution time improvements are as high as 30%.

Performance degradation on `srad v1`. Interestingly, BO FSS does not perform well on two workloads: `srad v1` and `cc-skitter`. While the performance difference in `cc-skitter` is marginal, the difference in `srad v1` is not. This phe-

TABLE 3
Minimax Regret of Scheduling Algorithms

Workload	*Ours*	Static	Workload-Aware		Dynamic					
	BO FSS	**STATIC**	**HSS**	**BinLPT**	**GUIDED**	**FSS**	**CSS**	**FAC2**	**TRAP1**	**TAPER3**
`lavaMD`	**0.00**	17.55	N/A	N/A	7.25	3.00	0.36	0.25	10.33	42.64
`stream.`	**0.00**	10.79	N/A	N/A	2.39	10.36	1.25	0.68	2.00	2.45
`kmeans`	**0.00**	23.02	N/A	N/A	8.01	17.62	1.50	1.17	2.30	6.41
`srad v1`	22.34	10.92	N/A	N/A	16.75	11.74	26.03	**0.00**	16.43	17.61
`nn`	4.76	5.06	N/A	N/A	**0.00**	0.55	7.00	6.06	4.39	5.14
`cc-journal`	**0.00**	2.88	66.98	196.63	11.94	2.47	2.98	6.15	3.65	0.66
`cc-wiki`	**0.00**	6.94	58.57	154.31	10.37	2.77	6.58	5.29	7.88	5.27
`cc-road`	**0.00**	8.57	81.88	251.71	7.19	1.37	1.55	1.23	1.97	1.71
`cc-skitter`	5.28	2.28	61.69	129.08	3.57	1.03	1.05	1.06	0.73	**0.00**
`pr-journal`	**0.00**	29.66	5.52	66.89	42.93	29.01	29.07	29.17	29.33	28.81
`pr-wiki`	15.30	45.20	**0.00**	42.26	85.34	46.99	47.28	46.82	46.53	46.87
`pr-road`	**0.00**	0.32	41.65	138.32	6.60	0.41	0.42	0.42	0.40	0.41
`pr-skitter`	**0.00**	11.51	23.21	68.91	29.97	11.66	11.21	11.34	12.06	11.26
$\mathcal{R}(S)$	**22.34**	45.20	81.88	251.71	85.34	46.99	47.28	46.83	46.53	46.87

nomenon is due to the large deviations in the execution time measurements as shown in Figure 6. That is, large outliers near $\theta = 0.4$ and $\theta = 0.8$ cause to deviate the GP prediction (green line), reducing the Gaussianity of the noise. Since GPs assume the noise to be Gaussian, they are not well suited for this kind of situation. A possible remedy is to use the *Student-T process* [37], [38], shown as the blue line. In Figure 6, the Student-T process is much less affected by outliers, resulting in a tighter fit.

Comparison of Gaussian Process Models. We now compare the simple GP construction in Section 3.2 and the locality-aware GP construction in Section 3.3. We equip BO with each of the models, and run the autotuning process beginning to end 30 times. The convergence results are shown in Figure 7. We can see that the locality-aware construction converges much quickly. Note that the shown results are averages. In the individual results, there are cases where the locality-unaware version completely fails to converge within a given budget. We thus suggest to use the locality-aware construction whenever possible. It achieves consistent results at the expense of additional computation during tuning.

Remarks. Apart from `srad v1`, BO FSS performs better than FSS and FAC2 on most workloads. This indicates that the Gaussian assumption works fairly well in most cases. We can conclude that our BO augmentation improves the performance of FSS on both workloads with high and low static imbalances. Our interest is now to see how this improvement compares against other scheduling algorithms.

5.4 Evaluation on Workloads Without Static Imbalance

This section compares the performance of BO FSS against dynamic scheduling methods on workloads where a workload-profile is unavailable or uniformative. The benchmark results are shown in Figure 8. Out of the 5 workloads considered, BO FSS outperforms all other methods on 3 out of 5 workloads. On the `nn` workload, the difference between all methods is insignificant. As discussed in Section 5.3, BO FSS performs poorly on the `srad v1` workload. Note that the same tuning results are used both for Section 5.3 and this experiment. Thus, with proper treatment of outliers, it is possible that BO FSS could have shown better results.

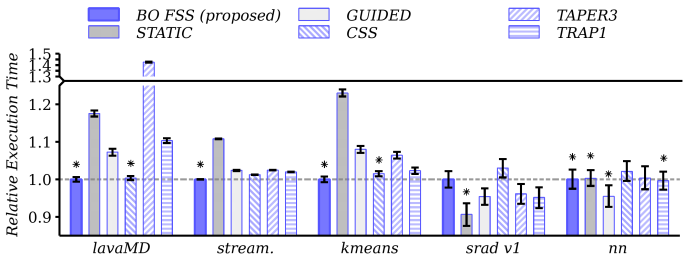


Fig. 8. Execution time comparison of BO FSS against dynamic scheduling methods. We estimate the mean execution time from 256 executions. The error bars show the 95% bootstrap confidence intervals. The results are normalized by the mean execution time of BO FSS. The methods with the lowest execution time are marked with a star (*). Methods *not* significantly different with the best performing method are also marked with a star (Wilcoxon signed rank test, 1% null-hypothesis rejection threshold).

Remarks. Compared to other dynamic scheduling methods, BO FSS achieves more consistent performance. However, because of the turbulence in the tuning process, BO FSS performs poorly on `srad v1`. It is thus important to ensure that BO FSS correctly converges to a critical point before applying it.

5.5 Evaluation on Workloads With Static Imbalance

This section evaluates the performance of BO FSS against workload-aware methods using workloads with a workload-profile. The evaluation results are shown in Figure 9. Except for the `pr-wiki` workload, BO FSS dominates all considered baselines. Because of the large number of tasks, both the HSS and BinLPT algorithms do not perform well on these workloads. Meanwhile, the STATIC and GUIDED strategies are very inconsistent in terms of performance. On the `pr-wiki` and `pr-journal` workloads, both methods are nearly 30% slower than BO FSS. This means that these algorithms lack workload-robustness unlike BO FSS.

On the `pr-wiki` workload which has the most extreme level of static imbalance, HSS performs significantly better. As discussed in Section 2.3, HSS has a very large critical section, resulting in a large amount of scheduling overhead. However, on the `pr-wiki` workload, the inefficiency caused by load imbalance is so extreme compared to the inefficiency caused by the scheduling overhead, giving HSS a relative advantage.

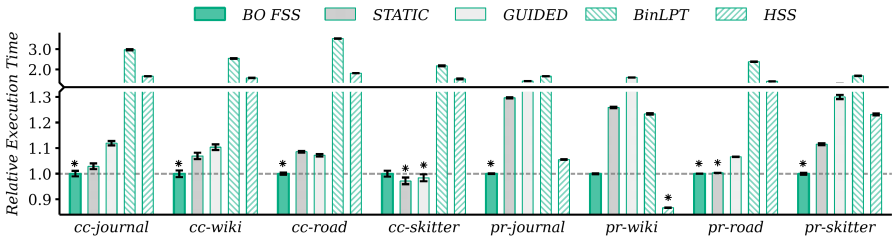


Fig. 9. Execution time comparison of BO FSS against workload-aware methods. We estimate the mean execution time from 256 executions. The error bars show the 95% bootstrap confidence intervals. The results are normalized by the mean execution time of BO FSS. The methods with the lowest execution time are marked with a star (*). Methods *not* significantly different with the best performing method are also marked with a star (Wilcoxon signed rank test, 1% null-hypothesis rejection threshold).

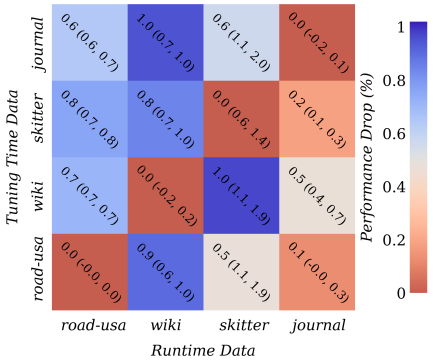


Fig. 10. Effect of mismatching the data used for tuning BO FSS and the data used for execution. The rows are the data used for tuning of BO FSS, while the columns are the data used for execution. The numbers represent the percentage slowdown relative to the matched case. Colder colors represent higher slowdown (worst performance).

Does the input data affect performance?. BO FSS's performance is tightly related to the individual property of each workload. It is thus interesting to ask how much the input data of the workload affects the behavior of BO FSS. To analyze this, we interchange the data used to tune BO FSS and the data used to measure the performance. If the input data plays an important role, the discrepancy between the *tuning time data* and the *runtime data* would degrade the performance. The corresponding results are shown in Figure 10 where the entries are the percentage increase in execution time relative to the *matched case*. Each row represents the dataset used for tuning, while each column represents the dataset used for execution. The anti-diagonal (bottom left to top right) is the case when the data is matched. The maximum amount of degradation is caused when we use `skitter` for tuning and `wiki` during runtime. Also, the case of using `journal` for tuning and `wiki` during runtime significantly degrades the performance. Overall, the `wiki` and `road` datasets turned out to be the pickiest about the match. Since both `wiki` and `road` resulted in high degradation, the amount of imbalance in the data does not determine how important the match is. However, judging from the fact that the degradation is at most 1%, we can conclude that BO FSS is more sensitive to the workload's algorithm rather than its input data.

Remarks. Compared to the workload-aware methods, BO FSS performs the best except for one workload which has the most amount of imbalance. Excluding this extreme case, the performance benefits of BO FSS is quite large. We also evaluate the sensitivity of BO FSS on perturbations to the workload. Results show that BO FSS is not affected much by changes in the input data of the workload.

5.6 Overhead Analysis

BO FSS has specific duties, both online and offline. When online, BO FSS loads the precomputed scheduling parameter θ_i, measures the loop execution time and stores the pair (θ_i, τ_i) in the dataset $\mathcal{D}_t$. A storage memory overhead of $O(T)$, where T is the number of BO iterations, is required to store $\mathcal{D}_t$. This is normally much less than the $O(N)$ memory requirement, where N is the number of tasks, imposed by workload-aware methods. When offline, BO FSS runs BO using the dataset $\mathcal{D}_t$ and determines the next scheduling parameter θ_{i+1}. Because most of the actual work is performed offline, the online overhead of BO FSS is almost identical to that of FSS. The offline step is relatively expensive due to the computation complexity of GPs. Fortunately, BO FSS converges within 10 to 20 iterations for most cases. This allows the computational cost to stay within a reasonable range.

6 Related Works

Classical dynamic loop scheduling methods. To improve the efficiency of dynamic scheduling, many classical algorithms are introduced such as CSS [6], FSS [7], TSS [8], BOLD [9], TAPER [10] and BAL [11]. However, most of these classic algorithms are derived in a limited theoretical context with strict statistical assumptions. Such an example is the *i.i.d.* assumption imposed on the workload.

Adaptive and workload-aware methods. To resolve this limitation, adaptive methods are developed starting from the *adaptive factoring self-scheduling* algorithm [12]. Recently, workload-aware methods including HSS [14] and BinLPT [15], [16] are introduced. These scheduling algorithms explicitly require a workload-profile before execution and exploit this knowledge in the scheduling process. On the flip side, this requirement makes these methods difficult to use in practice since the exact workload-profile may not always be available beforehand. In contrast, our proposed method is more convenient since we only need to measure the execution time of a loop. Also, the overall concept of our method is more flexible; it is possible to plug in our framework to any parameterized scheduling algorithm, directly improving its robustness.

Machine learning based approaches. Machine learning has yet to see many applications in parallel loop scheduling. In [56], Wang and O'Boyle use compiler generated features to train classifiers that select the best-suited scheduling strategy for a workload. This approach contrasts with ours since it

does not improve the effectiveness of the chosen scheduling algorithm. Recently, Khatami et al. in [57] use a logistic regression model for predicting the optimal chunk size for a scheduling strategy, combining CSS and work-stealing. Similarly, Laberge et al. [58] propose a machine-learning based strategy for accelerating linear algebra applications. These supervised-learning based approaches are limited in the sense that they are not yet well understood: their performance is dependent on the quality of the training data. It is unknown how well these approaches *generalize* across workloads from different application domains. In fact, quantifying and improving generalization is still a central problem in supervised learning as a whole. Our method is free of these issues since we directly optimize the performance for a target workload.

7 Conclusion

In this paper, we have presented BO FSS, a data-driven, adaptive loop scheduling algorithm based on BO. The proposed approach automatically tunes its performance to the workload using execution time measurements. Also, unlike the scheduling algorithms that are inapplicable to some workloads, our approach is generally applicable. We implemented our method on the `OpenMP` framework and quantified its performance as well as robustness on realistic workloads. BO FSS has consistently performed well on a wide range of real workloads, showing that it is robust compared to other loop scheduling algorithms. Our approach motivates the development of computer systems that can automatically adapt to the target workload.

Acknowledgments

The authors would like to thank Pedro Henrique Penna for the helpful discussions about the BinLPT scheduling algorithm. We thank Myoung Suk Kim for comments about our statistical analysis. We also thank Rover Root for helpful comments about the scientific computation workloads considered in this work.

References

[1] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, Jan.-March/1998.

[2] J. Regier, K. Pamnany, K. Fischer, A. Noack, M. Lam, J. Revels, S. Howard, R. Giordano, D. Schlegel, J. McAuliffe, R. C. Thomas, and Prabhat, "Cataloging the Visible Universe Through Bayesian Inference at Petascale," *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 44–53, 2018.

[3] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, and e. al, "Exascale Deep Learning for Climate Analytics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[4] A. G. Baydin, L. Shao, W. Bhimji, L. Heinrich, L. Meadows, J. Liu, A. Munk, S. Naderiparizi, B. Gram-Hansen, G. Louppe, M. Ma, X. Zhao, P. Torr, V. Lee, K. Cranmer, Prabhat, and F. Wood, "Etalumis: Bringing probabilistic programming to scientific simulators at scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Denver Colorado: ACM, Nov. 2019, pp. 1–24.

[5] D. Durand, T. Montaut, L. Kervella, and W. Jalby, "Impact of Memory Contention on Dynamic Scheduling on NUMA Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 11, pp. 1201–1214, Nov. 1996.

[6] C. P. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 10, pp. 1001–1016, Oct. 1985.

[7] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A Method for Scheduling Parallel Loops," *Commun. ACM*, vol. 35, no. 8, pp. 90–101, Aug. 1992.

[8] T. H. Tzen and L. M. Ni, "Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 87–98, Jan. 1993.

[9] T. Hagerup, "Allocating Independent Tasks to Parallel Processors: An Experimental Study," *Journal of Parallel and Distributed Computing*, vol. 47, no. 2, pp. 185–197, Dec. 1997.

[10] S. Lucco, "A Dynamic Scheduling Method for Irregular Parallel Programs," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, ser. PLDI '92. New York, NY, USA: ACM, 1992, pp. 200–211.

[11] H. Bast, "On Scheduling Parallel Tasks at Twilight," *Theory of Computing Systems*, vol. 33, no. 5-6, pp. 489–563, Dec. 2000.

[12] I. Banicescu and V. Velusamy, "Load Balancing Highly Irregular Computations with the Adaptive Factoring," in *Proceedings of the 16th International Parallel and Distributed Processing Symposium*. Ft. Lauderdale, FL: IEEE, 2002, p. 12 pp.

[13] F. M. Ciorba, C. Iwainsky, and P. Buder, "OpenMP Loop Scheduling Revisited: Making a Case for More Schedules," in *Proceedings of the IWOMP 2018: Evolving OpenMP for Evolving Architectures*. Springer, 2018, pp. 21–36.

[14] A. Kejariwal, A. Nicolau, and C. D. Polychronopoulos, "History-Aware Self-Scheduling," in *Proceedings of the 2006 International Conference on Parallel Processing (ICPP'06)*. IEEE, 2006.

[15] P. H. Penna, M. Castro, P. Plentz, H. Cota de Freitas, F. Broquedis, and J.-F. Méhaut, "BinLPT: A Novel Workload-Aware Loop Scheduler for Irregular Parallel Loops," in *Proceedings of the Simpósio Em Sistemas Computacionais de Alto Desempenho*, Campinas, Brazil, Oct. 2017.

[16] P. H. Penna, A. T. A. Gomes, M. Castro, P. D.M. Plentz, H. C. Freitas, F. Broquedis, and J.-F. Méhaut, "A Comprehensive Performance Evaluation of the BinLPT Workload-Aware Loop Scheduler," *Concurrency and Computation: Practice and Experience*, Feb. 2019.

[17] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the Human Out of the Loop: A Review of Bayesian Optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, Jan. 2016.

[18] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 469–482.

[19] B. Letham, B. Karrer, G. Ottoni, and E. Bakshy, "Constrained Bayesian Optimization with Noisy Experiments," *Bayesian Analysis*, vol. 14, no. 2, pp. 495–519, Aug. 2018.

[20] V. Dalibard, M. Schaarschmidt, and E. Yoneki, "BOAT: Building Auto-Tuners with Structured Bayesian Optimization," in *Proceedings of the 26th International Conference on World Wide Web - WWW '17*. Perth, Australia: ACM Press, 2017, pp. 479–488.

[21] K.-r. Kim, Y. Kim, and S. Park, "Towards Robust Data-Driven Parallel Loop Scheduling Using Bayesian Optimization," in *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. Rennes, FR: IEEE, Oct. 2019, pp. 241–248.

[22] K. Swersky, J. Snoek, and R. P. Adams, "Freeze-Thaw Bayesian Optimization," *arXiv:1406.3896 [cs, stat]*, Jun. 2014.

[23] GCC, "GCC, the GNU Compiler Collection," Jul. 2018.

[24] C. McPhail, H. R. Maier, J. H. Kwakkel, M. Giuliani, A. Castelletti, and S. Westra, "Robustness Metrics: How Are They Calculated, When Should They Be Used and Why Do They Give Different Results?" *Earth's Future*, vol. 6, no. 2, pp. 169–191, Feb. 2018.

[25] L. J. Savage, "The theory of statistical decision," *Journal of the*

TABLE 4
Details of Considered Baselines

Types	Chunk Size Equation	Parameter Setting
CSS [6]	$K = \left(\frac{h}{\sigma}\frac{\sqrt{2}N}{P\sqrt{\log P}}\right)^{2/3}$	h, σ, μ (*measured values*)
TAPER [10]	$v_\alpha = \alpha\frac{\sigma}{\mu}$, $x_i = \frac{R_i}{P} + \frac{K_{\min}}{2}$, $R_{i+1} = R_i - K_i$ $K_i = \max(K_{\min}, x_i + \frac{v_a^2}{2} - v_\alpha\sqrt{2x_i + \frac{v_\alpha^2}{4}})$	$v_\alpha = 3, K_{\min} = 1$
TSS [8]	$\delta = \frac{K_f - K_l}{N-1}$, $K_0 = K_f$ $K_{i+1} = \max(K_i - \delta, K_l)$	$K_f = \frac{N}{2P}$, $K_l = 1$,

American Statistical Association, vol. 46, no. 253, pp. 55–67, 1951.

[26] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, Liang Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*. Atlanta, GA, USA: IEEE, Dec. 2010, pp. 1–11.

[27] S. Beamer, K. Asanović, and D. Patterson, "The GAP Benchmark Suite," *arXiv:1508.03619 [cs]*, May 2017.

[28] P. Tang and P. C. Yew, "Processor Self-Scheduling for Multiple-Nested Parallel Loops," in *Proceedings of the International Conference on Parallel Processing (ICPP'86)*. IEEE, Dec. 1986, pp. 528–535.

[29] Bast, Hannah, "Provably Optimal Scheduling of Similar Tasks," Ph.D Thesis, Universität des Saarlandes, Saarbrücken, 2000.

[30] K. K. Yue and D. J. Lilja, "Parallel Loop Scheduling for High Performance Computers," in *High Performance Computing*, ser. Advances in Parallel Computing. North-Holland, 1995, vol. 10, pp. 243–264.

[31] L. E. Flynn and S. F. Hummel, "Scheduling Variable-Length Parallel Subtasks," IBM Research T. J. Watson Research Center, Tech. Rep., Feb. 1990.

[32] J. C. Spall, "An overview of the simultaneous perturbation method for efficient optimization," *Johns Hopkins apl technical digest*, vol. 19, no. 4, pp. 482–492, 1998.

[33] Z. Wang and S. Jegelka, "Max-value Entropy Search for Efficient Bayesian Optimization," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. JMLR.org, 2017, pp. 3627–3635.

[34] V. S. Adve and M. K. Vernon, "The Influence of Random Delays on Parallel Execution Times," in *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '93. New York, NY, USA: ACM, 1993, pp. 61–73.

[35] C. E. Rasmussen and Z. Ghahramani, "Occam's Razor," in *Advances in Neural Information Processing Systems 13*. MIT Press, 2001, pp. 294–300.

[36] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*, ser. Adaptive Computation and Machine Learning. Cambridge, Mass: MIT Press, 2006.

[37] R. Martinez-Cantin, K. Tee, and M. McCourt, "Practical Bayesian optimization in the presence of outliers," *arXiv:1712.04567 [cs, stat]*, Dec. 2017.

[38] A. Shah, A. G. Wilson, and Z. Ghahramani, "Bayesian Optimization using Student-t Processes," in *NIPS Workshop on Bayesian Optimisation*, 2013.

[39] D. Duvenaud, J. Lloyd, R. Grosse, J. Tenenbaum, and G. Zoubin, "Structure Discovery in Nonparametric Regression through Compositional Kernel Search," in *Proceedings of the 30th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 28. Atlanta, Georgia, USA: PMLR, Jun. 2013, pp. 1166–1174.

[40] J. M. Henrández-Lobato, M. W. Hoffman, and Z. Ghahramani, "Predictive Entropy Search for Efficient Global Optimization of Black-box Functions," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, pp. 918–926.

[41] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian Optimization of Machine Learning Algorithms," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 2951–2959.

[42] M. D. Hoffman and A. Gelman, "The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo," *Journal of Machine Learning Research*, vol. 15, no. 47, pp. 1593–1623, 2014.

[43] I. Sobol', "On the distribution of points in a cube and the approximate evaluation of integrals," *USSR Computational Mathematics and Mathematical Physics*, vol. 7, no. 4, pp. 86–112, Jan. 1967.

[44] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[45] D. R. Jones, C. D. Perttunen, and B. E. Stuckman, "Lipschitzian optimization without the Lipschitz constant," *Journal of Optimization Theory and Applications*, vol. 79, no. 1, pp. 157–181, Oct. 1993.

[46] S. G. Johnson, *The NLopt Nonlinear-Optimization Package*, 2011.

[47] H. Ge, K. Xu, and Z. Ghahramani, "Turing: A language for flexible probabilistic inference," in *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*, 2018, pp. 1682–1690.

[48] C. D. Polychronopoulos and D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1425–1439, Dec. 1987.

[49] J. M. Bull, "Measuring Synchronisation and Scheduling Overheads in OpenMP," in *In Proceedings of the First European Workshop on OpenMP*, 1999, pp. 99–105.

[50] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group Formation in Large Social Networks: Membership, Growth, and Evolution," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 44–54.

[51] D. Gleich, "Wikipedia-20070206," 2007.

[52] C. Demetrescu, A. Goldberg, and D. Johnson, "9th DIMACS Implementation Challenge - Shortest Paths," 2006.

[53] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations," in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, ser. KDD '05. New York, NY, USA: Association for Computing Machinery, 2005, pp. 177–187.

[54] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[55] S. Bak, Y. Guo, P. Balaji, and V. Sarkar, "Optimized Execution of Parallel Loops via User-Defined Scheduling Policies," in *Proceedings of the 48th International Conference on Parallel Processing*. Kyoto Japan: ACM, Aug. 2019, pp. 1–10.

[56] Z. Wang and M. F. O'Boyle, "Mapping Parallelism to Multicores: A Machine Learning Based Approach," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 75–84.

[57] Z. Khatami, L. Troska, H. Kaiser, J. Ramanujam, and A. Serio, "HPX Smart Executors," *Proceedings of the Third International Workshop on Extreme Scale Programming Models and Middleware: ESPM2'17*, 2017.

[58] g. laberge, S. Shirzad, P. Diehl, H. Kaiser, S. Prudhomme, and A. S. Lemoine, "Scheduling Optimization of Parallel Linear Algebra Algorithms Using Supervised Learning," in *2019 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*. Denver, CO, USA: IEEE, Nov. 2019, pp. 31–43.

Khu-rai Kim (Student Member, IEEE) is working towards his B.S. degree with the Department of Electronics Engineering, Sogang University, Seoul, South Korea.

His research interests lie in the duality of machine learning and computer systems, including parallel computing, compiler runtime environments, probabilistic machine learning and Bayesian inference methods.

Youngjae Kim (Member, IEEE) received the B.S. degree in computer science from Sogang University, South Korea, in 2001, the MS degree in computer science from KAIST, in 2003, and the PhD degree in computer science and engineering from Pennsylvania State University, University Park, Pennsylvania, in 2009.

He is currently an associate professor with the Department of Computer Science and Engineering, Sogang University, Seoul, South Korea. Before joining Sogang University, Seoul, South Korea, he was a R&D staff scientist with the US Department of Energy's Oak Ridge National Laboratory (2009–2015) and as an assistant professor at Ajou University, Suwon, South Korea (2015–2016). His research interests include operating systems, file and storage systems, parallel and distributed systems, computer systems security, and performance evaluation.

Sungyong Park (Member, IEEE) received the B.S. degree in computer science from Sogang University, Seoul, South Korea, and both the MS and PhD degrees in computer science from Syracuse University, Syracuse, New York.

He is currently a professor with the Department of Computer Science and Engineering, Sogang University, Seoul, South Korea. From 1987 to 1992, he worked for LG Electronics, South Korea, as a research engineer. From 1998 to 1999, he was a research scientist at Bellcore, where he developed network management software for optical switches. His research interests include cloud computing and systems, high performance I/O and storage systems, parallel and distributed system, and embedded system.