2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)

Towards Robust Data-driven Parallel Loop Scheduling Using Bayesian Optimization

Khu-rai Kim
Electronics Engineering Dept.
Sogang University
Seoul, Republic of Korea
msca8h@sogang.ac.kr

Youngjae Kim
Computer Science and Engineering Dept.
Sogang University
Seoul, Republic of Korea
youkim@sogang.ac.kr

Sungyong Park
Computer Science and Engineering Dept.
Sogang University
Seoul, Republic of Korea
parksy@sogang.ac.kr

***Abstract*—Efficient parallelization of loops is critical to improving the performance of high-performance computing applications. Many classical parallel loop scheduling algorithms have been developed to increase parallelization efficiency. Recently, workload-aware methods were developed to exploit the structure of workloads. However, both classical and workload-aware scheduling methods lack what we call *robustness*. That is, most of these scheduling algorithms tend to be unpredictable in terms of performance or have specific workload patterns they favor. This causes application developers to spend additional efforts in finding the best suited algorithm or tune scheduling parameters. This paper proposes *Bayesian Optimization augmented Factoring Self-Scheduling* (BO FSS), a robust data-driven parallel loop scheduling algorithm. BO FSS is powered by *Bayesian Optimization* (BO), a machine learning based optimization algorithm. We augment a classical scheduling algorithm, *Factoring Self-Scheduling* (FSS), into a robust adaptive method that will automatically adapt to a wide range of workloads. To compare the performance and robustness of our method, we have implemented BO FSS and other loop scheduling methods on the OpenMP framework. A regret-based metric called *performance regret* is also used to quantify robustness. Extensive benchmarking results show that BO FSS performs fairly well in most workload patterns and is also very robust relative to other scheduling methods. BO FSS achieves an average of 4% performance regret. This means that even when BO FSS is not the best performing algorithm on a specific workload, it stays within a 4 percentage points margin of the best performing algorithm.**

***Keywords*-Parallel Loop Scheduling, Bayesian Optimization, Parallel Computing, OpenMP**

I. Introduction

Loop scheduling algorithms try to increase the efficiency of parallelization by exploiting the loop structures abundant in scientific applications. The importance of this class of algorithms has been quickly realized and led to the creation of various classical scheduling strategies [1]–[7]. Recent study [8] revealed that none of these scheduling algorithms is optimal in a general case. This is because most of these algorithms do not exploit information about the workload.

To address this issue, workload-aware scheduling algorithms such as the *History-aware Self-Scheduling* (HSS) [9] and the *Bin packing Longest Processing Time* (BinLPT) [10], [11] have emerged. While these workload-aware methods perform well when the imbalance pattern, or *workload-profile*, is known beforehand, the workload-profile may not always be available in practice. Also, their performance becomes limited when the number of tasks greatly outnumbers the computing units, which is typical in parallel applications. Moreover, both classical and workload-aware scheduling algorithms lack *robustness*, meaning that they do not perform consistently well on all types of workloads. With a lack of robustness, additional efforts should be made to find the best algorithm suited to a particular workload.

Recent advances in machine learning and optimization have offered new methods that can solve complex problems using data. Such problems include compiler optimization flag selection [12] and cloud configuration selection [13]. In this paper, we bridge the gap between classical and workload-aware scheduling algorithms using a new data-driven, adaptive strategy called *Bayesian Optimization augmented Factoring Self-Scheduling* (BO FSS). Using a machine learning based optimization algorithm called *Bayesian Optimization,* (BO) [14], we augment one of the classical scheduling algorithms called *Factoring Self-Scheduling* (FSS) [2] into a more robust loop scheduling algorithm that automatically adapts to the workloads. In particular, BO is notorious for its sensitivity to tunable hyperparameters due to its internal *Gaussian Process* (GP). Therefore, we use an *Auxiliary Particle Filter* (APF), a *Sequential Monte Carlo* (SMC) algorithm. APF marginalizes away hyperparameters reducing the pain of hyperparameter tuning.

We have implemented BO FSS as well as other classic scheduling algorithms such as *Chunk Self-Scheduling* (CSS) [1], *Factoring Self-Scheduling* (FSS) [2], *Trapezoid Self-Scheduling* (TSS) [3], *Tapering Self-Scheduling* (TAPER) [5], HSS and BinLPT, and integrated them into the OpenMP parallelism framework [15]. The performance of BO FSS is evaluated against HSS, BinLPT and other scheduling algorithms. To assess the robustness of our method, a regret-based metric [16], [17] called *performance regret* is used for quantifying robustness. Using this metric, we show that BO FSS is more robust than other scheduling methods on a wide range of workloads.

To summarize, the key contributions of this paper are as follows:

Corresponding author: Sungyong Park

- **Development of a robust, data-driven, and adaptiveloop scheduling algorithm**. We propose a novel loop scheduling algorithm that uses BO. The proposed algorithm performs well on a diverse range of workloads without significant variation in performance.
- **Application of Bayesian optimization for optimizing the performance of computer systems**. We apply BO to directly optimize the execution time of parallel loops. We show that our BO implementation performs well without any workload-specific tuning. The APF algorithm automatically marginalizes GP hyperparameters, reducing the effort spent in tuning the scheduling algorithm.
- **Implementation and comparison of parallel loop scheduling algorithms on realistic workloads**. We have implemented a variety of loop scheduling algorithms, including their BO augmented versions into the OpenMP framework. We analyze their performance and robustness on realistic workloads from the Rodinia 3.1 benchmark.

II. Background and Motivation

A. Background

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 *Computing Units* (CU). The most basic scheduling strategy called *Static Scheduling* (STATIC) equally divides the tasks by the number of CUs P in compile time. The execution time, or length, of the ith task, is denoted as T_i. Usually, a barrier is implied at the end of a loop. All the CUs must wait until all tasks have been computed, and if an imbalance is present across the tasks, some CUs may finish computation before other tasks. This results in inefficient parallelization. Execution time variance is abundant in practice because of control statements and inherent noise in modern computer systems [18].

Dynamic scheduling was introduced to solve the inefficiency caused by execution time variance of each task. 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 case where $K = 1$ is called *Self-Scheduling* (SS) [19]. The dynamic loop scheduling problem was mathematically formalized in [1], [20] and a review of the problem is done in [21].

B. Related Works

To improve the efficiency of dynamic scheduling, many classical algorithms were introduced such as CSS [1], FSS [2], TSS [3], BOLD [4], TAPER [5] and BAL [6]. However, most of these classic algorithms were derived in a limited context with strict statistical assumptions. Such an example is the *identically, independently distributed* (iid) assumption imposed on the workload. In this case, all the tasks are assumed to have a fixed mean execution time $\mathbb{E}[T_i] = \mu$ and variance $\mathbb{V}[T_i] = \sigma^2$. Even in the cases where the workload distribution is known in advance, or an inherent bias in the workload distribution is present, these algorithms cannot take much advantage.

To resolve this limitation, adaptive and workload-aware methods were developed starting from the *Adaptive Factoring Self-Scheduling* algorithm [7]. Recently, HSS [9] and BinLPT [10], [11] were developed. 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. Also, these methods tend to scale poorly with a large number of tasks.

Machine learning, on the other hand, has been applied to parallel scheduling just in a handful of cases. In [22], Wang and O'Boyle used compiler generated features to train classifiers that select the best-suited scheduling strategy for a workload. While this approach does not improve the effectiveness of the chosen scheduling algorithms, it has been experimentally shown that no scheduling algorithm dominates others in [8]. Thus, it is practical to automatically select an optimal scheduling strategy based on the workload. Recently, Khatami et al. in [23] used a logistic regression model for predicting the optimal chunk size for a scheduling strategy combining CSS and work-stealing. Their approach is limited, however, since a basic logistic regression model is a linear classifier.

The performance of classical scheduling algorithms varies greatly across workloads [8]. This means that it is necessary to choose a scheduling algorithm that is appropriate for a particular type of workload. This problem led to the development of machine learning systems that automatically recommend the appropriate algorithm [22]. Workload-aware methods such as the BinLPT and HSS are not free of this issue. Our motivation is to develop an adaptive scheduling algorithm that is less picky about the workload and quantify its pickiness, or in other words, its *robustness*.

C. Motivation

While the HSS and BinLPT strategies fully incorporate information about the workload distribution, they have significant drawbacks. An accurate workload-profile must be available before execution. In situations where the workload distribution is not known, extensive benchmarking is required to acquire a precise workload-profile. Another major drawback of both the HSS and BinLPT methods is that they tend to perform poorly when the number of tasks largely surpasses the number of CUs. This phenomenon can be seen in Figure 1, where the execution times of different scheduling algorithms are compared against HSS and BinLPT by varying their parameter. Unlike in the *hotspot* suite where the tasks count is small, the HSS and BinLPT perform poorly

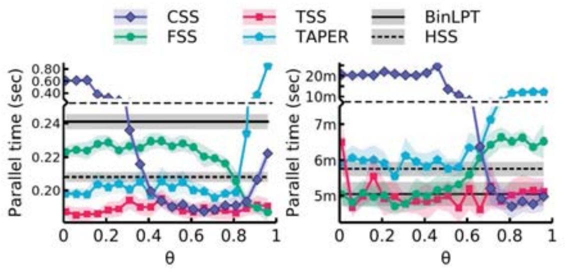

Figure 1: Visualization of the parameter space (θ) of scheduling algorithms compared to HSS and BinLPT. (Left) *kmeans* benchmark suite ($N = 494020$). (Right) *hostpot* benchmark suite ($N = 4096$). N is the number of tasks. A description of θ for each scheduling algorithm is provided in Table I. The colored region is the 95% confidence interval.

against other scheduling algorithms on the *kmeans* suite. In our terms, this makes these algorithms not robust enough since their performance can vary depending on the workload. Additionally, the workload-profile imposes a $O(N)$ runtime memory overhead for each loop. If the number of tasks is huge, this overhead may not be negligible.

Classical scheduling algorithms such as CSS, FSS, TSS, and TAPER have tunable parameters. When appropriately tuned, the performance of these algorithms can sometimes surpass that of HSS and BinLPT. This can also be seen in the visualization of the parameter spaces (θ) shown in Figure 1. We denoted the parameter as θ. For the case of the *kmeans* benchmark where the task number is large, all four scheduling algorithms easily surpass the performance of HSS and BinLPT by a significant margin. Even in the *hotspot* suite where the number of tasks is small, which is in favor of HSS and BinLPT, appropriately tuning θ in some algorithms achieves similar or better performance. This means that the robustness of these classical algorithms can be increased against both workloads with a large and small number of tasks.

In summary, the performance potential of classical scheduling algorithms points towards the possibility of creating a novel robust scheduling algorithm.

III. Design of Bayesian Optimization Augmented Factoring Self-Scheduling

A. Factoring Self-Scheduling

FSS was originally developed by Hummel et al. [2] extending the CSS algorithm. The chunk size is annealed in multiple *batches* according to (1). At the ith batch, P chunks of size K_i are allocated. R_i is the number of remaining tasks at the ith batch.

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

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

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

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

The parameter θ in (2) is crucial to the performance of FSS. The analysis in [24] concluded that $\theta = \sigma/\mu$ results in the best performance under strict statistical assumptions. However, we argue that determining θ by solving an optimization problem yields a better, more consistent solution.

B. Formulation as an Optimization Problem

The optimization problem is formulated as follows. First, let us denote the total execution time of a loop as $T(\mathcal{S}, \theta, P, N)$ in (5) where $\mathcal{S}$ is the chosen scheduling strategy. The overhead caused by σ and h is denoted as $f(\mathcal{S}, \theta, \sigma, h, P, N)$. The observed total execution time $\hat{T}$ in (6) contains observation noise ϵ. This noise term represents various real-life variations present in modern computer systems such as *Translation Lookaside Buffer* (TLB) misses, cache misses and network contentions. We assume ϵ to follow a Gaussian distribution. This assumption will be justified in Section III-C.

$$T(\mathcal{S}, \theta, P, N) = \frac{N}{P}\mu + f(\mathcal{S}, \theta, \sigma, h, P, N) \tag{5}$$

$$\hat{T} = T(\mathcal{S}, \theta, P, N) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma_\epsilon^2) \tag{6}$$

Our goal is to minimize the imbalance term f from the observations of $\hat{T}$. Since μ is assumed to be fixed, T is proportional to f. Thus, optimizing T is equivalent to optimizing the execution time imbalance. Assuming the noise to have zero mean, we can optimize T by optimizing the expectation of $\hat{T}$. The problem is stated as (7)

$$\underset{\theta}{\text{minimize}} \;\; \mathbb{E}[\,\hat{T}(\mathcal{S}, \theta, P, N)\,] \tag{7}$$

Since we do not have an exact model of $f(\mathcal{S}, \theta, \sigma, h, P, N)$, we cannot use gradient-based methods. To make things worse, most optimization algorithms perform poorly in the presence of noise. These difficulties motivates us to use BO for solving this optimization problem.

C. Solution using Bayesian Optimization

BO is a gradient-free black-box optimization algorithm. BO has been shown to be successful in optimizing complex real-life systems that are often noisy and non-convex [12], [13], [25], [26].

The BO algorithm is explained in Algorithm 1. BO first fits a *surrogate model*, in which we used the *Gaussian Process* (GP) algorithm. GPs assume the target function to be a sequence of jointly Gaussian random variables with covariance between arbitrary points given by a covariance kernel function $k(x_i, x_j)$. Under the Gaussian assumption, we can predict the mean $\mu(x|\mathcal{D})$ of a point. The uncertainty of the prediction is quantified as prediction variance

Algorithm 1 Bayesian Optimization

```
1) Initially sample enough observations and form a dataset
   $\mathcal{D}_0 = \{(x_0, y_0), (x_0, y_0), \dots, (x_N, y_N)\}$
2) Run Bayesian Optimization.
   Until stopping criterion is met. $i = 0, 1, \dots T$
   a) Fit surrogate model from observation data.
      Fit $\mathcal{GP}_i$ from $\mathcal{D}_i$
   b) Optimize acquisition function.
      $x_i = \arg\min_x \alpha(x|\mathcal{GP}_i)$
   c) Evaluate objective function.
      $y_i \leftarrow f(x_i)$
   d) Add observation to dataset.
      $\mathcal{D}_{i+1} \leftarrow \mathcal{D}_i \cup (x_i, y_i)$
3) Return point with lowest predicted mean.
   $\arg\min_x \mu(x|\mathcal{D}_T)$.
```

$\sigma^2(x|\mathcal{D})$. We used the *Squared-Exponential Covariance Kernel* for computing the covariance.

GPs can naturally express the noise present in program execution time. This is done by adding a small constant σ^2_ϵ to the diagonal of the covariance matrix. In the GP setting, this is equivalent of assuming Gaussian noise, which justifies our assumption about execution time noise in Section III-B. We describe a process for automatically estimating σ^2_ϵ in Section III-D.

$$\alpha_{LCB}(x|\,\mathcal{GP}_t\,) = \mu(x|\mathcal{D}_t) - \sqrt{\beta_t \sigma^2(x|\mathcal{D}_t)} \tag{8}$$

$$\beta_t = 2\log(\frac{t^2\pi^2}{6\delta}),\ \ \delta = 0.1 \tag{9}$$

After approximating the execution time resulting from scheduling parameters, it is important to decide which value of the scheduling parameter we will explore next. *Exploring* a point that has high uncertainty might reveal points with good performance that were previously unknown. Points that are already known to result in good performance could also be *exploited*. The *acquisition function* is in charge of solving this decision problem often called the *exploration-exploitation tradeoff*. We chose the *Lower Confidence Bound* (LCB) acquisition function [27]. From its formulation in (8), we can see that the LCB policy balances the influence of the objective value $\mu(x|\mathcal{D})$ and the uncertainty $\sigma^2(x|\mathcal{D})$ by a constant β_t. By optimizing the acquisition function, BO explores the parameter space and gradually improves the performance of BO FSS.

D. Marginalization of Bayesian Optimization Hyperparameters Using Sequential Monte Carlo

GPs have multiple hyperparameters that determine its performance. These hyperparameters are directly related to the attributes of the workload. Since real-life workloads are very diverse, it is important to automatically handle these parameters to prevent the need for manual tuning. Such parameters include the characteristic length l of the covariance kernel, objective function mean μ_f, objective function variance σ^2_f and the noise variance σ^2_ϵ. We denote these parameters with the parameter vector θ. Instead of hand-tuning θ, it is possible to estimate it by maximizing the likelihood of the GP, $p(\mathcal{D}|\theta)$. Instead, we chose to take the Bayesian way of threating θ by marginalizing it away. This is done by computing the intractable integral in (10).

$$\mathbb{E}_\theta[\,\mu(x|\mathcal{D})\,] = \int \mu(x|\theta,\mathcal{D})p(\theta|\mathcal{D})d\theta \tag{10}$$

$$\approx \sum_i^N \frac{w_i}{\sum_i^N w_i}\mu(x|\theta_i,\mathcal{D}) \tag{11}$$

Since computing the likelihood of a GP is expensive as it requires inverting a matrix, there have been approaches that exploit the iterative nature of BO by using *Sequential Monte Carlo* (SMC) methods [14], [28]–[30]. We used the *Auxiliary Particle Filter* (APF), a SMC method that approximates (10) using the finite sum (11). Using APF, we automatically infer the characteristics of the workload. As a detailed description of APF is out of our scope, readers interested in particle based SMC methods are pointed to [31].

E. Implementation

We implemented our BO FSS scheduling algorithm on the GCC implementation of the OpenMP 4.5 framework [15]. To compare the performance of BO FSS against other scheduling algorithms, we also implemented the CSS, FSS, TSS and TAPER scheduling algorithms, and their augmented versions. The scheduling algorithm can be selected by setting the `OMP_SCHEDULE` environment variable, or by the OpenMP runtime API as in the Listing 1. We followed a similar implementation with the GCC version of the GUIDED schedule using *compare-and-swap* (CAS) synchronization.

Listing 1: Selecting a scheduling algorithm

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

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)
```

Our method explained in Section III-B requires the identification of the individual loops in the OpenMP runtime. A major issue we encountered is that the current OpenMP ABI does not provide a way for such identification. We instead had to modify the GCC 8.2 [32] compiler's code generating backend and the OpenMP ABI. The modified GCC OpenMP ABI is listed 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 identification token, we can store and manage the state of each loop. Measurements of loop execution time are 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`.

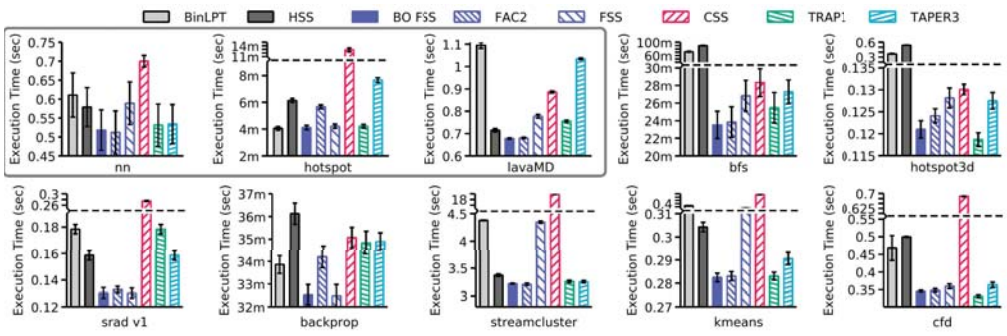

Figure 2: Benchmark results on the Rodinia 3.1 workloads. The error bars show the 95% confidence intervals. Workloads with a relatively small task count are grouped in the grey rectangle.

We implemented the BO algorithm described in Section III-C with the GP algorithm described in III-D. For solving the subproblem of optimizing α_{LCB}, we experimented with multiple optimization algorithms and settled with the *Covariance Matrix Adaptation Evolution Strategy* (CMA-ES) algorithm [33]. Both the BO and GP algorithms were implemented from scratch using C++, the Blaze linear algebra library [34] and CMA-ESpp library [35] for CMA-ES.

IV. Evaluation

A. Experiment Setup and Performance Metric

All experiments were conducted on a single shared-memory node with an AMD Ryzen Threadripper 1950X 3.4GHz CPU which has 16 cores (32 threads). The GCC 8.3 compiler was used with `-O3`, `-march=native` optimization flags enabled.

In order to evaluate BO FSS from different perspectives, we also augmented other classic scheduling algorithms such as the CSS, TSS, and TAPER, apart from FSS. These schemes were used as baselines for comparison marked with a BO prefix. The internal parameters of these algorithms are summarized in Table I.

Heuristic versions of the FSS, TSS, TAPER algorithms initially introduced in their original works are also included for evaluation. These are denoted FAC2, TAPER, TRAP1 respectively. Statistics of the workloads μ, σ, and the workload-profile were acquired by executing multiple profiling runs. The parameter h required by CSS was calculated according to [36]. BO augmented scheduling strategies were trained for 50 iterations starting from 30 warmup iterations. The workloads used for experiments are from the Rodinia 3.1 benchmark [37]. The characteristics of the benchmark workloads are summarized in Table II. In the case of TSS, (which has two tunable parameters K_f and K_l), the original authors suggested $K_f = N/2P$, $K_l = 1$ as a rule of thumb. Following this recommendation, we optimized $\theta = \delta$ where δ is the difference between subsequent tasks.

The performance of previously developed scheduling algorithms tends to vary greatly across workloads. This lack of robustness complicates the process of developing parallel applications since searching for the best-suited scheduling strategy is necessary. To quantify the robustness of scheduling algorithms, we propose to use a regret-based robustness measure [16], [17]. We call this the *performance regret*, denoted in (12). Regret is computed by the latency relative to the best performing scheduling algorithm on a workload. Denoting the set of scheduling algorithms as $\{\mathcal{S}\}$ and the set of workloads as $\mathcal{W}$, the average regret and the standard deviation of regret are computed as (13). A robust scheduling algorithm achieves consistently low regret overall, hence minimizing the regret mean $\mu_{\mathcal{R}}$ and standard deviation $\sigma_{\mathcal{R}}$. For example, an algorithm that performs the best on all workloads will measure as $\mu_{\mathcal{R}} = 0, \sigma_{\mathcal{R}} = 0$.

Table I: Parameters of considered baselines

Scheduling Alg.	Parameter	θ
CSS [1]	h/σ	h/σ
TAPER [5]	$\alpha\sigma/\mu$	$\alpha\sigma/\mu$
TSS [3]	K_f, K_l [1]	δ

[1] K_f, K_l is the size of the first and last chunk.

Table II: Benchmark workloads

Suite	Characterization	N	Application Domain
kmeans	Linear Algebra	494020	Data Mining
lavamd	N-Body	8000	Molecular Dynamics
nn	Linear Algebra	8192	Data Mining
streamcluster	Linear Algebra	65536	Data Mining
hotspot	Structured Grid	4096	Physics Simulation
hotspot3D	Structured Grid	209715	Physics Simulation
cfd	Unstructured Grid	193474	Fluid Dynamics
bfs	Graph Traversal	1000000	Graph Alorithms
srad	Structured Grid	229916	Image Processing
backprop	Unstructured Grid	1048592	Deep Learning

$$\mathcal{R}(\mathcal{S}) = \frac{T(\mathcal{S}, \theta, P, N) - \min_{i \in \{\mathcal{S}\}} \{T(\mathcal{S}_i, \theta, P, N)\}}{\min_{i \in \{\mathcal{S}\}} \{T(\mathcal{S}_i, \theta, P, N)\}} \times 100 \tag{12}$$

$$\mu_{\mathcal{R}} = \mathbb{E}_{\mathcal{W}}[\, R(\mathcal{S}) \,], \quad \sigma_{\mathcal{R}} = \mathbb{V}_{\mathcal{W}}[\, R(\mathcal{S}) \,] \tag{13}$$

Table III: Performance regret of scheduling algorithms

Benchmark	HSS	BinLPT	BO FSS	BO CSS	BO TAPER	BO TSS	CSS	FAC2	FSS	TAPER3	TRAP1
backprop	11.2	4.2	0.2	4.9	6.7	4.6	7.9	5.3	**0.0**	7.4	7.2
bfs	308.7	224.5	6.9	12.3	19.6	**0.0**	28.8	8.4	22.0	24.0	15.8
cfd	51.1	41.6	4.5	1.9	2.4	11.5	107.5	5.3	8.8	10.1	**0.0**
hotspot	364.0	215.1	4.1	**0.0**	3.3	18.0	11.9	6.8	10.2	9.7	2.1
hotspot3D	58.5	4.9	6.5	**0.0**	43.2	6.0	232.0	46.4	8.7	97.8	8.9
kmeans	7.7	29.6	**0.0**	0.3	3.3	0.5	91.0	0.2	17.9	2.9	0.2
lavamd	5.4	61.5	**0.0**	8.4	0.2	4.1	30.7	0.3	14.7	52.6	11.4
nn	27.4	34.4	14.1	13.5	9.0	**0.0**	54.1	12.7	29.6	17.5	16.9
srad	126.0	41.1	3.1	3.2	8.0	6.7	117.4	5.0	3.0	25.5	**0.0**
streamcluster	5.2	36.6	0.4	9.9	3.2	0.7	519.3	**0.0**	35.6	1.5	1.6
$\mu_{\mathcal{R}}$	96.5	69.4	**4.0**	5.4	9.9	5.2	120.1	9.0	15.1	24.9	6.4
$\sigma_{\mathcal{R}}$	125.5	77.0	**4.2**	4.9	12.3	5.5	147.6	13.0	10.8	28.2	6.3

B. Performance Evaluation

Performance Analysis: Figure 2 shows the comparison of BO FSS with other scheduling algorithms. Overall, BO FSS performed consistently well. It ended up being the best or stayed within the confidence interval of the best performing algorithm in 8 out of 10 workload cases. For example, BO FSS was outperformed only in *hotspot3D* and *cfd* by TRAP1, which is the heuristic version of TSS. This is because the parameter found in the space of FSS did not perform better than TRAP1 on those specific workloads. However, among the FSS algorithm family (FSS, BO FSS, FAC2), BO FSS performed the best. This shows the fact that the performance of BO FSS is bounded to that of FSS. However, within the space of FSS, BO can find the best parameter that minimizes the relative performance loss. This minimization of regret is what we claim to be the key to robustness.

The grey rectangle demarks the workloads with a relatively small task count N. This includes *nn*, *lavaMD*, and *hotspot*. On these workloads, both BinLPT and HSS performed reasonably well since they prefer a small task count. However, the performance was inconsistent compared to that of BO FSS. Because the parameter space of FSS allows good performance even with small tasks, when appropriately tuned, the performance of FSS and FAC2 can be made more consistent. As a result, BO FSS outperformed BinLPT and HSS despite their preference.

Robustness Analysis: To quantify the robustness of BO FSS, we compared the performance regret against other algorithms. We included other BO augmented algorithms to show that BO FSS is the most consistent among BO augmented algorithms. The performance regrets of the experimented algorithms are shown in Table III. The bottom row is the mean and standard deviation of regret. Compared to other methods, BO FSS achieved the lowest regret mean and standard deviation. This means that BO FSS performs consistently well on a wide range of workloads without fluctuating. Compared to other BO augmented algorithms, this can be interpreted as the FSS algorithm having the widest, most flexible parameter space allowing for aggressive workload adaptation

The BO augmented algorithms showed improvements in robustness as much as 115 percentage points over their original counterparts. Mainly, the CSS algorithm after applying BO augmentation (BO CSS) performed quite competitively with a regret of 5.4. This contradicts the previous belief that decreasing chunk size schemes such as FSS, TSS, and TAPER, should outperform CSS, which is a constant chunk size scheme [1], [2]. Not restricted to CSS, It is apparent that scheduling algorithms behave differently after properly adapting to the workload. This observation potentially opens the door to future parameterized scheduling algorithms designed with BO augmentation in mind. These algorithms should form a new, robust class of scheduling methods.

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 $\hat{T}_i$ and stores the pair $(\theta_i, \hat{T}_i)$ in the dataset $\mathcal{D}$. A storage memory overhead of $O(T)$, where T is the number of BO iterations, is required to store $\mathcal{D}$. This is normally much less than the $O(N)$ memory requirement, where N is the number of tasks, imposed by other workload-aware methods. The number of tasks N tends to grow as parallel applications scale with more data. When offline, BO FSS runs BO using the dataset $\mathcal{D}$ 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 $O(T^3)$ computation complexity of GPs. Fortunately, we experienced that BO FSS converges within 50 iterations for most cases. This lets the computational cost to stay within a reasonable range.

V. 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 online measurements. We implemented our method on the OpenMP framework and quantified its performance plus robustness. BO FSS has consistently performed well on a wide range of real workloads, showing that it is robust compared to other loop scheduling algorithms.

ACKNOWLEDGEMENT

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2014–0–00035, Research on High Performance and Scalable Manycore Operating System) and Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7080245)

REFERENCES

[1] 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. [Online]. Available: https://doi.org/10.1109/tse.1985.231547

[2] 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. [Online]. Available: http://doi.acm.org/10.1145/135226.135232

[3] 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.

[4] 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. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0743731597914118

[5] 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. [Online]. Available: http://doi.acm.org/10.1145/143095.143134

[6] H. Bast, "On Scheduling Parallel Tasks at Twilight," *Theory of Computing Systems*, vol. 33, no. 5-6, pp. 489–563, Dec. 2000. [Online]. Available: http://link.springer.com/10.1007/s002240010013

[7] 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. [Online]. Available: http://ieeexplore.ieee.org/document/1015661/

[8] 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. [Online]. Available: https://arxiv.org/abs/1809.03188

[9] 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. [Online]. Available: https://doi.org/10.1109/icpp.2006.49

[10] 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. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01596427

[11] 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. [Online]. Available: http://doi.wiley.com/10.1002/cpe.5170

[12] 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. [Online]. Available: https://projecteuclid.org/euclid.ba/1533866666

[13] 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. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard

[14] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. d. 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.

[15] "OpenMP Application Programming Interface," OpenMP Architecture Review Board, Tech. Rep. Version 4.5, Nov. 2015. [Online]. Available: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[16] 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. [Online]. Available: http://doi.wiley.com/10.1002/2017EF000649

[17] L. J. Savage, "The Theory of Statistical Decision," *Journal of the American Statistical Association*, vol. 46, no. 253, pp. 55–67, Mar. 1951. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/01621459.1951.10500768

[18] 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.

[19] 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)*, K. Hwang, S. M. Jacobs, and E. E. Swartzlander, Eds. IEEE, 1986, pp. 528–535.

[20] Bast, Hannah, "Provably Optimal Scheduling of Similar Tasks," Ph.D Thesis, Universität des Saarlandes, Saarbrücken, 2000. [Online]. Available: http://hdl.handle.net/11858/00-001M-0000-000F-332B-D

[21] K. K. Yue and D. J. Lilja, “Parallel Loop Scheduling for High Performance Computers,” in *High Performance Computing*, ser. Advances in Parallel Computing, J. J. Dongarra, G. R. Joubert, L. Grandinetti, and J. Kowalik, Eds. North-Holland, 1995, vol. 10, pp. 243 – 264. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S092754520680016X

[22] Z. Wang and M. F. O’Boyle, “Mapping Parallelism to Multi-cores: 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. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504189

[23] 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. [Online]. Available: https://arxiv.org/abs/1711.01519

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

[25] G. Kochanski, D. Golovin, J. Karro, B. Solnik, S. Moitra, and D. Sculley, “Bayesian Optimization for a Better Dessert,” in *Proceedings of the NIPS Workshop on Bayesian Optimization (BayesOpt’17)*, 2017.

[26] R.-R. Griffiths and J. M. Hernández-Lobato, “Constrained Bayesian Optimization for Automatic Chemical Design,” in *Proceedings of the NIPS Workshop on Bayesian Optimization (BayesOpt’17)*, 2017.

[27] N. Srinivas, A. Krause, S. M. Kakade, and M. W. Seeger, “Information-Theoretic Regret Bounds for Gaussian Process Optimization in the Bandit Setting,” *IEEE Transactions on Information Theory*, vol. 58, no. 5, pp. 3250–3265, May 2012. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6138914

[28] R. B. Gramacy and N. G. Polson, “Particle Learning of Gaussian Process Models for Sequential Design and Optimization,” *Journal of Computational and Graphical Statistics*, vol. 20, no. 1, pp. 102–118, Jan. 2011. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1198/jcgs.2010.09171

[29] Y. Wang and B. Chaib-draa, “A Marginalized Particle Gaussian Process Regression,” in *Proceedings of the Neural Information Processing Systems (NIPS’12)*, 2012, pp. 1187–1195. [Online]. Available: https://papers.nips.cc/paper/4850-a-marginalized-particle-gaussian-process-regression.pdf

[30] K. R. Dalbey and L. P. Swiler, “Gaussian Process Adaptive Importance Sampling,” *International Journal for Uncertainty Quantification*, vol. 4, no. 2, pp. 133–149, 2014.

[31] O. Cappe, S. J. Godsill, and E. Moulines, “An Overview of Existing Methods and Recent Advances in Sequential Monte Carlo,” *Proceedings of the IEEE*, vol. 95, no. 5, pp. 899–924, May 2007.

[32] F. S. F. , “GCC, the GNU Compiler Collection,” Jul. 2018. [Online]. Available: https://gcc.gnu.org/

[33] N. Hansen, “The CMA Evolution Strategy: A Comparing Review,” in *Towards a New Evolutionary Computation*, J. A. Lozano, P. Larrañaga, I. Inza, and E. Bengoetxea, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, vol. 192, pp. 75–102. [Online]. Available: http://link.springer.com/10.1007/3-540-32494-1_4

[34] K. Iglberger, G. Hager, J. Treibig, and U. Rüde, “Expression Templates Revisited: A Performance Analysis of Current Methodologies,” *SIAM Journal on Scientific Computing*, vol. 34, no. 2, pp. C42–C69, Jan. 2012. [Online]. Available: http://epubs.siam.org/doi/10.1137/110830125

[35] A. Fabisch, “CMA-ESpp,” 2011. [Online]. Available: https://github.com/AlexanderFabisch/CMA-ESpp

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

[37] 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. [Online]. Available: http://ieeexplore.ieee.org/document/5650274/