SimAS: A simulation-assisted approach for the scheduling algorithm selection under perturbations

Ali Mohammed | Florina M. Ciorba

Department of Mathematics and Computer Science, University of Basel, Basel, Switzerland

Correspondence
Florina M. Ciorba, High Performance Computing Group, Department of Mathematics and Computer Science, University of Basel, Spiegelgasse 1, CH-4051 Basel, Switzerland.
Email: florina.ciorba@unibas.ch

Funding information
Swiss Platform for Advanced Scientific Computing, Grant/Award Number: 169123

Summary
Many scientific applications consist of large and computationally intensive loops. Dynamic loop self-scheduling (DLS) techniques are used to parallelize and to balance the load of such applications during execution. Load imbalance arises from variations in the loop iteration (or tasks) execution times, caused by problem, algorithmic, or systemic characteristics. Variations in systemic characteristics are referred to as perturbations. Our hypothesis is that no single DLS technique can achieve the absolute best performance under various perturbations on heterogeneous high-performance computing (HPC) systems. Therefore, the selection of the most efficient DLS technique is critical to achieve the best application performance. The goal of this work is to solve the algorithm selection problem for the scheduling of computationally intensive applications under perturbations. Existing work only considers perturbations caused by variations in the delivered computational speed of the HPC systems. However, perturbations in available network bandwidth or latency are inevitable on production HPC systems. A simulation-assisted scheduling algorithm selection (SimAS) approach is introduced herein as a novel control-theoretic-inspired approach to select DLS techniques dynamically that improve the performance of applications executing on heterogeneous HPC systems under perturbations. The present work examines the performance of seven applications on a heterogeneous HPC system under all the above system perturbations. SimAS is evaluated using native and simulative experiments. The performance results confirm the original hypothesis that motivates this work. The experimental evaluation shows that the SimAS-based DLS selection identifies the most efficient technique and improves application performance in most cases.

KEYWORDS
algorithm selection problem, load balancing, loop scheduling, performance, perturbations, simulation

1 | INTRODUCTION

Scientific applications are often characterized by large and computationally intensive parallel loops. The performance of these applications on high-performance computing (HPC) systems may degrade due to load imbalance caused by problem, algorithmic, or systemic characteristics. Application (problem or algorithmic) characteristics include the irregularity of the number of computations per loop iterations due to conditional statements, where systemic characteristics include variations in delivered computational speed of processing elements (PEs), available network bandwidth, or latency. Such variations are referred to as perturbations and can also be caused by other applications or processes that share the same resources or a temporary system fault or malfunction. Dynamic loop self-scheduling (DLS) is a widely used approach for improving the execution of computationally intensive loops in scientific applications. Self-scheduling refers to the dynamic assignment of the loop iterations (tasks) to free and requesting Pes. Computationally intensive loops are an ample source of parallel independent tasks and can be found in a large number of computationally intensive and computation-and-communication-intensive scientific applications. A wide range of DLS techniques...
exists and can be divided into nonadaptive and adaptive techniques. The nonadaptive DLS techniques include self-scheduling (SS), fixed size chunking (FSC), modified FSC (mFSC), guided SS (GSS), trapezoid SS (TSS), factoring (FAC), and weighted factoring (WF) among others. Adaptive DLS techniques include adaptive WF (AWF), its variants (AWF-B), chunk (AWF-C), batch-like (AWF-D), and chunk-like (AWF-E), as well as adaptive factoring (AF), among others.

An a priori optimal selection of the most appropriate DLS technique for a given application and system is not trivial and oftentimes infeasible, given the various sources of load imbalance and the different load balancing properties of the DLS techniques. This represents the algorithm selection problem in the context of scheduling. The goal of this work is to solve the algorithm selection problem for the scheduling of computationally intensive applications on HPC systems under perturbations. Earlier work studied the flexibility of DLS (taken as robustness to variable delivered computational speed) and the selection of the most robust DLS using machine learning with the SimGrid (SG) simulation toolkit. The selection of DLS techniques for synthetic time-stepping scientific workloads using reinforcement learning was also studied using simulative experimentation. We define simulative experiments as the counterpart of native experiments to describe performance results obtained from simulations rather than direct execution on computing systems, ie, native performance. The aforementioned work focuses on one source of perturbations, namely, the variation in the delivered computing speed, in time-stepping applications to learn from previous time steps. In addition, that approach is not directly applicable to noniterative applications. Scheduling solutions using static optimizations, eg, using evolutionary and genetic algorithms, cannot dynamically adapt to the perturbations encountered during execution. Modern HPC systems are often heterogeneous production systems typically shared by many users. Therefore, perturbations in the available network bandwidth and latency are unavoidable in such systems.

The study of the performance of scientific applications with DLS under perturbations revealed that the most robust DLS technique, identified as the DLS technique that results in the least variation of the application execution time under various perturbations, does not always achieve the best performance in all execution scenarios. Figure 1 shows the simulative performance of PSIA (cf, Section 4.1) on 696 cores of miniHPC (cf, Section 4.4) under perturbations (cf, Section 4.6). According to these results, GSS is the most robust DLS technique due to the minimal variation of its performance under perturbations (Figure 1A); however, it results in poor application performance under perturbations. Even the next most robust DLS technique, WF, is outperformed by SS and AWF-C in certain perturbation scenarios, as can be seen in Figure 1B. These results suggest that even if the most robust DLS technique is known a priori, which could be challenging, the application performance degrades in different execution scenarios due to perturbations. Therefore, a methodology for the dynamic selection of DLS techniques is needed to achieve the highest possible performance in all execution scenarios. In the present work, in an effort to select the most appropriate DLS dynamically for a given application and system under perturbations, the simulation-assisted scheduling algorithm selection (SimAS) approach is proposed. The performance of two scientific applications (PSIA and Mandelbrot) executed in single-sweep and time-stepping modes, and five synthetic workloads is studied on a heterogeneous HPC system using nonadaptive and adaptive DLS techniques, in the presence of perturbations. The synthetic workloads are used to cover a broader spectrum of application load imbalance profiles beyond what is encountered in practice.

* Direct experiments on real HPC systems.
The present work makes the following contributions. First, it extends and optimizes the Simulator in the Loop\textsuperscript{20} (SIL) approach into the SimAS\textsuperscript{1} approach for dynamically selecting a DLS technique during execution based on the application characteristics and the present (monitored or predicted) state of the computing system. Second, it extends a dynamic load balancing tool (DLB\_tool) from the literature\textsuperscript{21} for parallelizing scientific applications into Dynamic Loop Scheduling For Load Balancing\textsuperscript{1} (DLS4LB) with four more DLS techniques, namely SS, FSC, WF, and TSS.

In addition, DLS4LB supports SimAS as the 14th option to select DLS techniques dynamically during execution. Third, it confirms via systematic native and simulative experimentation the original hypothesis that no single DLS ensures the best performance in all execution scenarios considered. It evaluates the performance of two applications (PSIA and Mandelbrot) and five synthetic workloads with SimAS under perturbations and shows that the dynamic selection of the most efficient DLS technique improves performance versus selecting the most robust one.

This work is structured as follows. Section 2 contains a brief review of the selected DLS techniques, the SG simulation toolkit, as well as the work related to the performance of scheduling scientific applications with DLS in the presence of perturbations. The proposed SimAS approach is conceptually described in Section 3. The factorial design of experiments, together with details about the DLS techniques and SimAS implementation into the DLS4LB, the HPC system characteristics, and the perturbations injected in native and simulative experiments are presented in Section 4. The analysis of the applications performance under perturbations is presented in Section 5. The work concludes and outlines potential future work in Section 6.

This paper extends the authors’ prior work.\textsuperscript{20} Specifically, an additional real-world application (Mandelbrot set) is used (in Section 4.1) to evaluate the performance of thirteen DLS under perturbations. Two versions of PSIA and Mandelbrot applications are considered, single-sweep and time-stepping (Section 4.1). DLB\_tool\textsuperscript{21} is extended with four more DLS techniques (described in Section 4.2) and is used in this work to parallelize and load balance the real world applications (see Section 4.1). In addition, the proposed SimAS approach is also integrated with DLS4LB (as clarified in Section 4.3). The performance of the applications of interest using DLS under perturbations is examined in native as well as simulative experiments (see Section 5.1). The native experiments confirm the conclusions drawn from the simulative experiments for the performance of applications under perturbations and in the evaluation of the proposed approach SimAS.

2 | BACKGROUND AND RELATED WORK

Scheduling. The aim of scheduling is to achieve a balanced load execution among parallel PEs with minimum scheduling overhead. A loop iteration is a parallel independent task that represents the smallest unit of work to schedule and execute. Loop scheduling techniques can be divided into static and dynamic. In static loop scheduling, the loop iterations are divided and assigned to PEs before execution; both division and assignment remain fixed during execution. This work considers static (block) scheduling, denoted STATIC, each PE being assigned a chunk size equal to the number of iterations $N$ divided by the number of PEs $P$. STATIC incurs minimum scheduling overhead, compared to dynamic scheduling, and may lead to load imbalance for nonuniformly distributed tasks and/or on perturbed systems.

In DLS, free and requesting PEs are assigned loop iterations during execution. The DLS techniques can be categorized into nonadaptive and adaptive techniques.

The nonadaptive DLS techniques considered in this work are: SS,\textsuperscript{22} FSC,\textsuperscript{3} mFSC,\textsuperscript{4} GSS,\textsuperscript{5} TSS,\textsuperscript{6} FAC,\textsuperscript{7} and WF.\textsuperscript{8} While STATIC represents one scheduling extreme, SS represents the other scheduling extreme. In SS, the size of each chunk is one loop iteration (one task). This yields a high load balance with potentially very large scheduling overhead. FSC assigns loop iterations in chunks of fixed sizes, where the chunk size depends on the standard deviation of loop iteration execution times $\sigma$ as an indication of its variation and the incurred scheduling overhead $h$. FSC requires this information ($h$ and $\sigma$) to be known before the execution to calculate the chunk size. mFSC alleviates the requirement of recalculating $h$ and $\sigma$, and calculates a fixed chunk size that results in a number of chunks equal to that produced by FAC (described below). GSS assigns loop iterations in chunks of decreasing sizes, where the size of a chunk is equal to the number of remaining unscheduled loop iterations $R$ divided by the number of PEs $P$. Similar to GSS, TSS assigns chunks of loop iterations in decreasing sizes. Unlike GSS, the chunk sizes decrease linearly, to ease the chunk calculation operation and to minimize the scheduling overhead.

FAC employs a probabilistic modeling of loop characteristics that takes into account the mean of iterations execution time $\mu$ and their standard deviation $\sigma$) to calculate batch sizes that maximize the probability of achieving a load balanced execution. A PE's chunk size is equal to the batch size divided by $P$. When $\mu$ and $\sigma$ are unavailable, a practical implementation of FAC assigns half of the remaining loop iterations $R$ to a batch. WF divides a batch of iterations into unequally-sized chunks, proportional to the relative PE speeds (called weights). The PE weights need to be determined prior to the execution and are assumed not to change during execution. This work considers the practical implementations of FAC and WF. All nonadaptive DLS techniques account for variations in the iteration execution times due to application characteristics.

The adaptive DLS techniques monitor the performance of the application during execution and adapt the chunk calculation accordingly. Adaptive DLS techniques include AWF\textsuperscript{9} and its variants\textsuperscript{10}. AWF-B, AWF-C, AWF-D, AWF-E, and AF,\textsuperscript{11} among others. AWF is designed for time-stepping applications. It improves WF by adapting the relative weights of PEs during execution by monitoring their performance in each time step. AWF-B relieves the time-stepping requirement in AWF and measures the performance after each batch to update the PE weights.

\textsuperscript{1}https://github.com/unibas-dmi-hpc/SimAS
\textsuperscript{2}https://github.com/unibas-dmi-hpc/DLS4LB
AFW-C is similar to AWF-B where weight updates are performed upon the completion of each chunk, instead of a batch. AWF-D is similar to AWF-B and considers the total chunk time (equal to the sum of the iteration execution times in the chunk plus the associated overhead of the PE to acquire the chunk) and all the bookkeeping operations to calculate and update the PE weights. AWF-B and AWF-C only consider the chunk iterations execution times. AWF-E is similar to AWF-C by updating the PE weights on every chunk. Yet, AWF-E is also similar to AWF-D by also considering the total chunk time. Unlike FAC, AF estimates the values of $\mu$ and $\sigma$ during execution and updates them based on the measured performance of the PEs on the executed loop iterations.

Loop scheduling in simulation. SG14 is a versatile event-based simulation toolkit designed for the study of the behavior of large-scale distributed systems. It provides ready-to-use application programming interfaces (APIs) to represent applications and computing systems through different interfaces: MSG (SG-MSG), SimDag (SG-SD), and SMPI (SG-SMPI). SG uses a simple and fast CPU computation model and verified, more complex network models,23 which render it well suited for the study of computationally intensive parallel and distributed scientific applications.

Various studies have used SG to evaluate the performance of applications with DLS techniques in different scenarios.13,15 To attain high trustworthiness in the performance results obtained with SG, the implementation of the nonadaptive DLS techniques in SG-SD has been verified24 by reproducing the results presented in the work that introduced factoring.7 The accuracy of the performance results obtained by simulative experiments against native experiments has also recently been quantified.25 The present work employs the SG-SD interface to study the performance of scientific applications on heterogeneous HPC systems under perturbations.

Related work. Scheduling of applications on large HPC systems involves many sources of uncertainties, eg, task execution times and perturbations in the computing system. Therefore, many studies have focused on robust schedules that maintain certain performance requirements despite fluctuations in the behavior of the system.26 Robust scheduling of tasks with uncertain execution and communication times was studied16,27 using a multiobjective evolutionary algorithm and using dynamic scheduling, respectively. Moreover, the flexibility, defined as robustness to perturbations of DLS techniques, was examined28 in an effort to select the most flexible technique using machine learning. However, a robust scheduling technique may not always guarantee the best performance in all possible execution scenarios and for all application parameters17 (eg, problem size and data distribution). Thus, dynamically selecting the best performing DLS technique is of paramount importance, given the broad spectrum of available DLS techniques, each with unique properties. Selecting the best performing DLS technique for time-stepping applications, using reinforcement learning was introduced15 by adapting to the variations in the delivered computational speed during previous time steps. In addition, machine learning and decision trees were used to select the best performing DLS technique dynamically from a portfolio of DLS techniques13 and for multithreaded applications parallelized with OpenMP29 or with Charm++.30

Scheduling solutions based on optimization techniques, such as genetic and evolutionary algorithms, cannot adapt to perturbations during execution. None of the aforementioned efforts considered perturbations in network bandwidth or latency. This work complements the previous efforts by studying the performance of scientific applications using nonadaptive and adaptive DLS techniques under different perturbation scenarios (variations in delivered computational speed, network bandwidth, and network latency) on an HPC system. A new approach, namely, SimAS is introduced, to select DLS techniques dynamically that improve the performance of applications on HPC systems under multiple sources of perturbations known mostly during execution.

3 | SimAS: SIMULATION-ASSISTED SCHEDULING ALGORITHM SELECTION

SimAS is inspired by control theory, where a controller (scheduler) is used to achieve and maintain a desired state (load balance) of the system (parallel loop execution). Its concept is motivated by the well-known control strategy, model predictive control (MPC).31 The MPC predicts the performance of the system with different control signals to optimize system performance. As shown in Figure 2, a call to a loop simulator is inserted inside a typical scheduling loop. SimAS leverages loop simulators to predict the performance of the application with various DLS techniques. The system monitor and estimator components read the system state during the execution and update the computing system representation accordingly to feed the simulator with the current perturbations on the system. SimAS examines the predicted performance by the simulator with different DLS techniques and selects the DLS technique that achieves the best performance in the current execution scenario. The above steps may be repeated several times during the execution of the loop, and the SimAS call frequency can be aligned with the occurrence of perturbations (monitored or predicted). The main idea of the SimAS approach (inspired by MPC) is to use the simulator (system model in MPC terms) to test different DLS techniques (control signals in MPC terms) on the loop execution (the system in MPC terms), before actually employing the selected DLS in the executing application.

The advantage of SimAS is that it leverages the use of already developed state-of-the-art simulators to predict the performance dynamically during execution. The prediction accuracy of a simulator is strongly influenced by the representation of both the applications and the systems in simulation as well as by the subsystem models it comprises.25 Since loop simulators predict the performance of load imbalanced computationally-intensive loops, the influence of the memory subsystem (eg, complex memory hierarchy) on their performance is minimal. Therefore, application performance can accurately be predicted via simulation. For instance, the percent error between native and simulative executions for a given application (PSIA18) using the SG-SD interface was found to be between 0.95% and 2.99%.25 The percent error is calculated as $\%E = (1 - \frac{T_{\text{sim}}}{T_{\text{native}}}) \times 100$, where $T_{\text{native}}$ and $T_{\text{sim}}$ are the native and simulative performance, respectively. Moreover, it was found that the performance simulations with SG captures the native applications performance features and identifies the most efficient DLS technique for
PSIA and Mandelbrot applications. It is expected that the accuracy and speed of the simulators employed by SimAS will improve as simulators in general are continuously being developed and refined.

The cost of frequent calls to SimAS can be amortized by launching parallel SimAS instances on dedicated resources to derive predictions for various DLS techniques. In addition, this cost can entirely be mitigated by asynchronously calling SimAS, i.e., the application does not block (nor wait) until SimAS simulations complete.

The system monitor and estimator components can be implemented with a number of system monitoring tools, such as collect. Such tools can periodically be instantiated to measure PE and network loads and to update the system representation in the simulator for the next call to SimAS. The measured chunk execution times can also be used to estimate the current PE computational speeds. The implementation details of the loop simulator and SimAS are described in Section 4. The PE loads can be estimated and predicted using autoregressive integrated moving average.

4 | DESIGN OF EXPERIMENTS

In this work, we employ a factorial design of experiments, due to the numerous parameters and values to explore. The design of the factorial experiments is presented in the following (cf. Table 1), along with details of the DLS techniques and SimAS implementation, the implemented loop simulator, the computing system under test and its injected perturbations in native and simulative experiments.

4.1 | Applications

This work considers two real-world applications (executed as single-sweep applications and as time-stepping applications) and five synthetic (single-sweep) workloads.

Real applications.

1. **PSIA**. Parallel spin-image algorithm (PSIA), is a computationally intensive application from computer vision. PSIA is embarrassingly parallel application and algorithmically load imbalanced where the computational effort of a loop iteration depends on the input data. The performance of PSIA has been studied in prior work and was enhanced for execution on a heterogeneous cluster by using nonadaptive DLS techniques. The total number of parallel loop iterations in PSIA is 400,000.

2. **Mandelbrot**. This application computes the Mandelbrot set and generates its image. The program is based on one of the codes available online. The application is parallelized such that the calculation of the value at every single pixel of a 2D image is a loop iteration, which is performed in parallel. The application is modified to compute the function $f(z) = z^2 + c$ instead of $f(z) = z^4 + c$ to increase the number of computations per task. The size of the generated image is $512 \times 512$ pixels resulting in $2^{18}$ parallel loop iterations.

3. **PSIA_TS**. This application is similar to PSIA. Unlike PSIA, PSIA_TS is executed in time steps. It simulates applying spin-image transformations to an object in motion (a video), where at each time-step a certain number of spin-images (4000) is created. PSIA_TS is executed for 10 time steps.

4. **Mandelbrot_TS**. This is the time-stepping version of Mandelbrot application. At each time step, the generated Mandelbrot set image at time $t$ is zoomed in by 5% on the center of the image to generate the image at $t + 1$. Mandelbrot_TS is executed for 10 time steps. The workload per time step is reduced compared to Mandelbrot (single-sweep) such that the execution time of 10 time steps of Mandelbrot_TS is comparable to the execution time of the single-sweep version. This is desirable for the purpose of native experimentation given the large set of experiments performed (see Table 1), to avoid extremely long execution times.

https://github.com/CaptGreg/SenecaOOP345-attic/blob/master/parallel-pgm/mpi/mandelbrot-mpi-dynamic.c
Synthetic workloads.

Five synthetic workloads are examined in this work. Each of the five synthetic workloads contains 400,000 parallel loop iterations. The number of floating point operations (FLOP count) in each loop iteration is assumed to follow five different probability distributions, namely: constant, uniform, normal, exponential, and gamma probability distributions. The probability distribution parameters used to generate these FLOP counts are also given in Table 1. The synthetic workloads are used to stress test the performance and usefulness of the proposed approach and to cover a broader spectrum of application load imbalance profiles than what may be encountered in practice.15,37

4.2 Loop scheduling

Thirteen loop scheduling techniques are used to assess the performance of the above nine applications under various execution scenarios. These techniques represent a wide range of static and dynamic loop scheduling approaches. The DLS techniques can further be distinguished into five adaptive and seven nonadaptive techniques.

| TABLE 1 | Details used in the design of the factorial experiments |
| Factors | Values | Properties |
| --- | --- | --- |
| Applications | PSIA | [5.9 \cdot 10^7 \ldots \ 6.6 \cdot 10^7] FLOP per iteration |
| | Mandelbrot | [5.9 \cdot 10^5 \ldots \ 2.6 \cdot 10^8] FLOP per iteration |
| | PSIA, TS (time-stepping) | [5.9 \cdot 10^7 \ldots \ 6.5 \cdot 10^7] FLOP per iteration |
| | Mandelbrot, TS (time-stepping) | [5.9 \cdot 10^5 \ldots \ 2.6 \cdot 10^8] FLOP per iteration |
| | Constant | 2.3 \cdot 10^8 FLOP per iteration |
| | Uniform | [10^3 \ldots \ 7 \cdot 10^9] FLOP per iteration |
| | Normal | μ = 9.5 \cdot 10^6 FLOP, σ = 7 \cdot 10^7 FLOP, [6 \cdot 10^8 \ldots \ 1.3 \cdot 10^9] FLOP per iteration |
| | Exponential | λ = 1/3 \cdot 10^9 FLOP, [9.48 \cdot 10^2 \ldots \ 4.5 \cdot 10^3] FLOP per iteration |
| | Gamma | k = 2, θ = 10^8 FLOP, [4.1 \cdot 10^6 \ldots \ 2.7 \cdot 10^9] FLOP per iteration |
| Problem size | N = 400,000 iterations, all applications except for PSIA, Mandelbrot, PSIA, TS, and Mandelbrot, TS on 128 miniHPC cores under all perturbations |
| Loop scheduling | STATIC | Static |
| | SS, FSC, mFSC, GSS, TSS, FAC, WF | Dynamic nonadaptive |
| Computing system | minHPC (heterogeneous HPC cluster) | Intel Broadwell and Intel Xeon Phi KNL nodes, relative core weights44 = 0.817 and 0.183, respectively P = 128 heterogeneous (4 \times 16 Broadwell + 1 \times 64 KNL) cores | |
| | | P = 416 heterogeneous (22 \times 16 Broadwell + 1 \times 64 KNL) cores |
| Perturbations | Nominal conditions | np (no perturbations) |
| | PE availability | pea-cm (constant mild): μ = 75%, σ = 0% |
| | | pea-cs (constant severe): μ = 25%, σ = 0% |
| | | pea-em (exponential mild): μ = 78%, σ = 24 \cdot 10^{-3}% |
| | | pea-es (exponential severe): μ = 31%, σ = 89 \cdot 10^{-3}% |
| Bandwidth | bw-cm (constant mild): μ = 1 \cdot 10^4 byte/s, σ = 0 byte/s |
| | | bw-cs (constant severe): μ = 1 \cdot 10^2 byte/s, σ = 0 byte/s |
| | | bw-em (exponential mild): μ = 1.1 \cdot 10^4 byte/s, σ = 9 \cdot 10^4 byte/s |
| | | bw-es (exponential severe): μ = 23 \cdot 10^2 byte/s, σ = 19 \cdot 10^2 byte/s |
| Latency | lat-cm (constant mild): μ = 1 \cdot 10^{-2}s, σ = 0s |
| | | lat-cs (constant severe): μ = 1s, σ = 0s |
| | | lat-em (exponential mild): μ = 1.2 \cdot 10^{-2}s, σ = 1.5 \cdot 10^{-2}s |
| | | lat-es (exponential severe): μ = 2.9s, σ = 1.8s |
| Combined | all-cm (constant mild): pea-cm, bw-cm, and lat-cm |
| | | all-cs (constant severe): pea-cs, bw-cs, and lat-cs |
| | | all-em (exponential mild): pea-em, bw-em, and lat-em |
| | | all-es (exponential severe): pea-es, bw-es, and lat-es |
| Experimentation | Native | PSIA, Mandelbrot, PSIA, TS, and Mandelbrot, TS on 128 miniHPC cores under targeted perturbations |
| | Simulative | PSIA and Mandelbrot on 128 miniHPC cores under all perturbations |

aCore weight relative to the total speed of a system of one core of each type.

bAvailable in the companion research report.36
The DLS4LB library is used to parallelize the applications with DLS. DLS4LB extends the dynamic load balancing tool (DLB_tool\textsuperscript{21}) with additional four DLS techniques. DLB_tool originally contained the implementation of nine loop scheduling techniques, namely STATIC, mFSC, GSS, FAC, AWF-B, AWF-C, AWF-D, AWF-E, and AF. DLS4LB supports four additional DLS techniques, namely SS, FSC, TSS, and WF.

DLS4LB employs MPI two-sided communications for work distribution among processes and implements a master-worker execution model, where the master is responsible for handling work requests from workers. In addition, the master acts also as a worker, and checks for outstanding work requests with a certain adjustable frequency. DLS4LB is designed to parallelize an application with minimum changes. Listing 1 shows, in green font color, the lines needed to be added to the application code to parallelize it.

4.3 | Simulation-assisted scheduling Algorithm Selection

DLS4LB is extended to support the SimAS approach as the 14th option in DLS4LB. Taking the same approach of DLS4LB of minimal application code changes, an application can use the SimAS approach by inserting only two function calls, shown in red font color, in Listing 1. SimAS\_setup function sets up the main data structure SimAS\_info that holds important information, such as the number of PEs, the number of loop iterations, the path to the simulator, the FLOP file that contains the FLOP count per loop iteration, and the platform file that describes the computing system. In addition, SimAS\_setup asynchronously starts the simulation of the application performance immediately with a portfolio of DLS techniques in parallel. SimAS\_setup sets the scheduling technique to a default DLS (WF in this work), to allow the application to start and avoid delaying the application execution.

SimAS\_update checks (every 5 seconds in this work) if the simulation is finished, and selects the DLS technique that allows the application to finish the largest number of tasks in the shortest time by manipulating the DLS\_info structure; otherwise, it will keep the selected DLS technique unchanged. SimAS\_update reruns the simulation again if 50 seconds (the SimAS calling frequency) have passed since the simulator was previously called or every new time step for time-stepping applications. SimAS\_update prevents the start of a new instance of the simulator unless the earlier one is completed or the number of remaining unscheduled iterations is less than or equal the number of Pes.

4.3.1 | SimAS improvements

Several measures were taken in this work to mitigate the overhead of simulation during execution, such as running the simulation in parallel, asynchronously to the application, to avoid stopping the application execution. A default DLS technique (namely, WF) was used until the simulation completes, based on its high performance in the simulative experiments. SimAS checks the completion of the simulation at certain (adjustable) periods to reduce the overhead of these checks. DLS techniques with poor performance on heterogeneous computing systems were excluded from the DLS portfolio provided to SimAS to reduce the number of needed simulations and the search space of the SimAS. Simulations launched by SimAS were executed on the 4 cores per node that are not used by the application and left for operating system, network, and other processes (see Section 4.4). Therefore, running the simulations does not perturb the executing application.

Listing 1  Dynamic load balancing with SimAS support using the DLS4LB library

```c
#include <mpi.h>
#include "DLS4LB.h"
#include "SimAS.h"

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);

Scheduling method = SimAS_setup(SimAS_info, P, N, h, sigma, sim_path, FLOP_file, platform_file);
DLS_setup(MPI_COMM_WORLD, DLS_info);
DLS_startLoop(DLS_info, N, scheduling_method);
while(not DLS_terminated(DLS_info))
{
    SimAS_update(DLS_info, SimAS_info);
    DLS_startChunk(DLS_info, Cstart, Csize);
    Compute_iterations(Cstart, size);
    DLS_endChunk(DLS_info);
}
DLS_endLoop(DLS_info);
```
4.4 Computing system in native and simulative experiments

The native experiments were conducted on miniHPC, a research and teaching cluster at the Department of Mathematics and Computer Science at the University of Basel, Switzerland. It consists of 26 compute nodes: 22 nodes each with dual socket Intel Xeon E5-2640 v4 (20 cores) configuration and 4 nodes each with one Intel Xeon Phi Knights Landing 7210 processor (64 cores). All nodes are interconnected with Intel Omni-Path fabrics in a nonblocking two-level fat-tree topology. To ensure the uninterrupted execution of applications on the allocated CPU cores, only 16 out of the 20 cores per node (8 out of 10 per socket) are used. The other 4 cores per node are left for the operating system, network load, and other nonapplication-related processes.

4.5 Simulation details

Applications in simulation. LoopSim, an SG-SD-based simulator, is used to simulate the applications of interest, where the loop iterations in the application code are represented as tasks. To represent the computational effort associated with an application's loop iterations, the number of floating point operations (FLOP) of each loop iteration is counted using PAPI counters. The FLOP count per iteration is then read by LoopSim during execution to simulate the computation per iteration. All DLS techniques supported by the DLS4LB are also implemented in LoopSim and tasks are assigned to free and requesting simulated cores, similar to the native execution.

The pseudocode of LoopSim is presented in Listing 2. LoopSim reads in the number of iterations (tasks), start task ID, the path to the file that contains the FLOP count per loop iteration, the path to the computing system representation (see below), the selected scheduling technique, and the maximum simulated time. The simulator reads the data and simulates the loop execution using the selected DLS technique. It then outputs the simulated time and the number of tasks executed in this simulated time. This information is read by the SimAS, which compares different DLS techniques based on this information and selects the DLS technique that results in the shortest execution time and largest number of finished tasks.

Computing system in simulation. A computing system is represented in SG via an XML file denoted as platform file. SG registers each processor core for its representation as a host in the platform file. The computational speed of a processor core is estimated by measuring a loop execution time and dividing it by the total number of floating point operations included in the loop. A Xeon core was found to be four

---

Listing 2  SG-SD loop simulator

```
#include <simdag.h>
#include <DLS_scheduling.h>

// read input
read_input(num_tasks, FLOP_file, start_task_ID, platform_file, DLS_t, max_sim_t);

// create tasks that represent loop iterations
Task_array = create_tasks(num_tasks, FLOP_file);
scheduled_tasks = 0;
while(executed_tasks < num_tasks) && (get_sim_time() < max_sim_t)
{
  idle_processes = get_idle_processes();
  foreach(idle_process in idle_processes)
  {
    // read and update finished tasks
    executed_tasks += get_finished_tasks(idle_process);
    // send work request to master
    send_work_request(idle_process, master);
    chunk = calculate_chunk(Task_array, num_tasks, scheduled_tasks, DLS_t);
    // assign work to worker
    send_work(master, idle_process);
    scheduled_tasks += chunk;
  }
  // resume simulation until a task is finished, i.e., a process is idle
  simulate_execution(platform_file);
}
print("simulated_time:" + get_sim_time());
print("finished_tasks:" + executed_tasks);
```
times faster than a Xeon Phi core as indicated by the relative core weights (cf. Table 1). The network bandwidth and latency represented in the platform file are calibrated with the SG calibration procedure.\footnote{http://simgrid.gforge.inria.fr/contrib/smpi-calibration-doc/}

## 4.6 | Injected perturbations

Three different categories of perturbations are considered in this work, namely delivered computational speed, available network bandwidth, and available network latency. Two intensities are considered, mild and severe, for each category. Two scenarios are considered for each intensity, where the value of the delivered computational speed is either constant or exponentially distributed.

All perturbations (cf. Table 1) are considered to occur periodically, with a period of 100 seconds where the perturbations affect the system only during 50% of the perturbation period. The network (bandwidth and latency) perturbations commence with the application execution, while the delivered computational speed perturbations begin 50 seconds after the start of the application. Another perturbation scenario is created by combining all perturbations from the other individual categories.

**Perturbations in simulative experiments.** All perturbations are enacted in SG during simulation via the availability, bandwidth, latency, and platform files to represent perturbations in delivered computational speed, network available bandwidth, and network latency, respectively.

**Perturbations in native experiments.** A program (CPU burner) is launched in parallel and pinned on the same processor cores as the application to induce perturbations on the PE availability in native execution. The program is executed periodically every 100 s and is only active during a fraction of this period that corresponds to the required PE availability perturbation (75% or 25%).

For injecting perturbations in the link latency, the all MPI communication functions are intercepted using the MPI profiling interface (PMPI), and certain delays are inserted to simulate longer communication latencies. The duration of the injected delays is proportional to the selected perturbation intensity, mild or severe, and is randomly drawn from the selected perturbation distribution, constant or exponential. Similar to PE perturbations, the injected delays are enacted periodically every 100 s and are only active over half the length of this period. The values of the injected delays are listed in Table 1. Given that the applications of interest are computationally intensive and the communicated data size between application’s processes is minimal, perturbations in the network bandwidth do not have a significant effect on the application performance, as can be seen from the simulative experiments below. Therefore, perturbations in the network bandwidth are excluded from native experimentation.

A combined perturbations scenario is created for the native execution by combining PE availability perturbations and network latency perturbations. As both perturbation distributions (constant and exponential) have a comparable effect on the performance, where the impact of constantly distributed perturbations is more evident, only the constant distribution of perturbations is considered in the native experiments.

## 5 | EXPERIMENTAL EVALUATION AND DISCUSSION

The performance results of the execution of the applications with different loop scheduling techniques under different execution scenarios are illustrated and discussed. We also show the need and importance of the proposed SimAS approach.

### 5.1 | Performance of scientific applications under perturbations

**Simulative experiments.** The simulative performance results of the two real applications, PSIA and Mandelbrot, under perturbations are shown in Figure 3. One can note that STATIC, GSS, TSS, and FAC perform poorly on heterogeneous systems. This is due to the fact that these techniques do not account for different computational powers in different Pes. However, SS, FSC, mFSC, WF, and the adaptive techniques significantly improve the performance under no perturbations.

Under perturbations, WF cannot accommodate the variability in the system due to perturbations as PE weights are constant, especially to perturbations in the delivered computational speed of the PEs. The performance of FSC and mFSC is, in general, better than that of STATIC, GSS, TSS, and FAC. However, FSC and mFSC are highly affected by the perturbations in the PE availability. SS is resilient to perturbations in the delivered computational speed of the PEs. However, it is significantly influenced by the network latency variations, as can be seen in Figure 3 (lat-cs and lat-es).

Perturbations in the network bandwidth show a minimal influence on performance, as the PEs only communicate loop iteration indices to calculate the start index and the size of the next chunk. Therefore, the communicated messages are small. The bandwidth perturbations are, thus, not selected for subsequent more targeted native experiments under perturbations. However, network latency perturbations show a significant effect on the performance of applications.

The adaptive techniques perform comparably, with a slight advantage for AWF-E as can be seen in Figure 3b (all-cm and all-es). However, in certain cases, such as lat-cm, lat-em, and all-em for in Mandelbrot on 128 cores (Figure 3B), AWF-B and AWF-D perform significantly poorer than all other techniques. This is due to the high variation of loop iteration execution times of the Mandelbrot that results in one rank obtaining a large chunk that delays the application execution and results in a misestimation of PE weights. In general, WF results in the best
Simulative performance results of PSIA and Mandelbrot without (denoted with np) and with (the rest) perturbations using SimAS and other 13 loop scheduling techniques on miniHPC. The heatmaps show percent performance improvement normalized with respect to STATIC in the np scenario (baseline case without any perturbations and baseline load balancing method). White, red, and blue boxes denote baseline (= 100%), degraded (> 100%), and improved performance (< 100%), respectively. Tables show the corresponding DLS techniques dynamically selected by SimAS during execution as percentage of the total execution time. A, PSIA simulative performance on 128 cores; B, Mandelbrot simulative performance on 128 cores.
FIGURE 4 Native performance results of PSIA and Mandelbrot without (denoted with np) and with (the rest) perturbations using SimAS and other 13 loop scheduling techniques on miniHPC. Percent performance improvement normalized with respect to STATIC in np scenario (baseline case without any perturbations and baseline load balancing method). White, red, and blue denote baseline (=100%), degraded (>100%), and improved performance (<100%), respectively. Each table shows the DLS techniques dynamically selected by SimAS as percentage of the total execution time and the percent of execution time spent in SimAS calls. A, PSIA native performance on 128 cores; B, Mandelbrot native performance on 128 cores

application performance with SimAS degraded in certain cases due to the nonpreemptive scheduling implementation. Although the technique with the best performance is selected upon a new call to SimAS, the execution of already scheduled loop iterations cannot be preempted to be resumed with the newly selected DLS. The code of the native application is instrumented with timers around the SimAS calls to measure its overhead. The results show that the overhead of calling SimAS is less than 0.5% of the application execution time (see Figure 4).

To show the applicability of the SimAS approach for improving the performance of time-stepping scientific applications, time-stepping versions of PSIA and Mandelbrot are also executed under perturbations with and without SimAS. In PSIA_TS and Mandelbrot_TS, SimAS starts a new simulation at the beginning of each time step. A default DLS technique is used, WF in these experiments or the DLS from the most recent time-step, until the simulations are finished. This represents another use case of SimAS with time-stepping applications, which is frequently encountered in scientific applications.

The results of the time-stepping applications are shown in Figure 5. Similar to the single-sweep versions of PSIA and Mandelbrot, SimAS improved the performance of applications in most of the cases. One can note that no single DLS technique always achieves the best performance, therefore, a dynamic selection of DLS technique according to the current perturbations in the system is needed. The overhead of calling SimAS is in general below 0.5% of the execution time, except for PSIA_TS for which the overhead is at most 2.7%. This is due to the short execution time of PSIA_TS compared to its nontime-stepping version.

The nonpreemptive scheduling approach of the DLS techniques significantly impacted the performance of applications with SimAS. The execution of already scheduled chunks of loop iterations is not preempted to be resumed with the newly selected DLS. As shown in Figure 4A, even though SimAS selected DLS techniques with shorter execution times in the case of lat-cs with PSIA application on 128 cores, the execution time with SimAS was even longer than that of SS, which was not selected by SimAS.

For time-stepping applications, the effect of frequently switching the DLS technique and the nonpreemption overhead is much less than for single-sweep applications. Therefore, the performance of time-stepping applications with SimAS under perturbations is better than that of the single-sweep versions of the same applications as shown in Figure 5A and Figure 5B. It is planned to study the preemption of scheduled (yet not executed) loop iterations in the future, to switch, without further delay, from one DLS to another during execution.

5.2 Discussion

Even though the applications considered are computationally-intensive and only communicate loop indices with the master, perturbations in network latency had a significant impact on performance. The implementation choice of the scheduling techniques, such as STATIC, implemented in an SS fashion, led to degrading its performance in scenarios with network perturbations.

Selecting the most performing DLS technique before execution might not deliver the best performance, as perturbations in the HPC system are unknown a priori. For instance, the best DLS technique for Mandelbrot that could be identified before execution, ie, in np execution scenario, is SS, which is outperformed by SimAS in lat-cs and pea+lat-cs in Figure 4B. Similar change in the best DLS technique can be seen from the results
FIGURE 5 Native performance results of PSIA_TS and Mandelbrot_TS without (denoted with np) and with (the rest) perturbations using SimAS and other thirteen loop scheduling techniques on miniHPC. The heatmaps show percent performance improvement normalized with respect to STATIC in the np scenario (baseline case without any perturbations and baseline load balancing method). White, red, and blue boxes denote baseline (= 100%), degraded (> 100%), and improved performance (< 100%), respectively. Tables show the DLS techniques dynamically selected by SimAS as percentage of the total execution time and the percentage of execution time spent in calls to SimAS. A, PSIA_TS native performance on 128 cores; B, Mandelbrot_TS native performance on 128 cores.

of Mandelbrot_TS in Figure 5B. Since there is no high load imbalance in the PSIA or PSIA_TS, there is no high variation in the performance of different DLS techniques. Since the best DLS technique cannot be known before execution, SimAS improved the performance by dynamically selecting the DLS with the best performance based on the simulation predictions.

In general, the DLS techniques are designed to be efficient. However, in certain cases, efficiency prevents robustness due to the reduced tolerance of the efficient techniques to uncertain events. Uncertainty is ineradicable and it manifests in HPC systems as perturbations. Perturbations due to non fatal errors and system interference significantly degrade applications performance on large scale HPC systems.1,39,40 This highlights the importance of the careful choice of DLS techniques for each application, system size, and execution scenario. The dynamic selection of DLS techniques ensures that each DLS technique is employed where and when its the most efficient.

The SimAS approach can proactively select the best suited DLS before any perturbations manifest in the system, whenever perturbations can be predicted in advance. SimAS leverages state-of-the-art simulators to select the most efficient DLS technique dynamically for an application under perturbations.

6 | CONCLUSIONS AND FUTURE WORK

SimAS is introduced for the dynamic selection of DLS techniques under perturbations. The performance of two real applications and five synthetic workloads was studied under perturbations and insights into the resilience of the DLS techniques to perturbations are provided. The performance results confirm the hypothesis that no single DLS technique can achieve the best performance in all the considered execution scenarios. Furthermore, native DLS experiments under system-induced perturbations showed that even the computationally intensive applications could be significantly affected by perturbations in the network characteristics. The implementation choice of scheduling techniques, such as STATIC implemented in a SS manner, led to the degradation of its performance under network perturbations. Using the SimAS approach improved the performance of applications in most experiments. SimAS leverages state-of-the-art simulators to select the most efficient DLS technique dynamically for an application under perturbations.

The scheduling of scientific applications has traditionally been approached without preemption. However, operating systems perform preemption of the processes and threads that execute a certain application task. We believe that application level scheduling can further improve performance if the scheduling strategies employ task preemption. It is planned in the future to experiment with preempting scheduled yet not executed loop iterations upon a change in the selected DLS technique by the SimAS approach. Empirical selection rules to select DLS techniques under perturbations can be created from examining pre-production executions of applications or simulations under various perturbations. Such a set of rules can replace the role of simulator in SimAS. The most efficient DLS technique can then be dynamically selected by SimAS using this empirical set of rules.
ACKNOWLEDGMENTS

This work was supported by the Swiss Platform for Advanced Scientific Computing (PASC) under project “SPH-EXA: Optimizing Smooth Particle Hydrodynamics for Exascale Computing” and by the Swiss National Science Foundation in the context of the “Multi-level Scheduling in Large Scale High Performance Computers (MLS)” (grant 169123). The authors gratefully acknowledge Ahmed Eleliemy from the University of Basel for sharing an initial implementation of the PSIA application. We also gratefully acknowledge the anonymous reviewers for their constructive comments which significantly improved the quality of this work.

ORCID

Ali Mohammed https://orcid.org/0000-0002-8465-0398
Florina M. Ciorba https://orcid.org/0000-0002-7773-4499

REFERENCES

1. Ashraf RA, Engelmann C. Analyzing the impact of system reliability events on applications in the Titan supercomputer. Paper presented at: 2018 IEEE/ACM 8th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS); 2018; Dallas, TX.
2. Asanovic K, Bodik R, Catanzaro BC, et al. The Landscape of Parallel Computing Research: A View From Berkeley. Technical Report. UCB/EECS-2006-183. Berkeley, CA: EECS Department, University of California at Berkeley; 2006.
3. Kruskal CP, Weiss A. Allocating independent subtasks on parallel processors. IEEE Trans Softw Eng. 1985;SE-11(10):1001-1016.
4. Banicescu I, Ciorba FM, Srivastava S. Performance optimization of scientific applications using an autonomic computing approach. In: Scalable Computing and Communications: Theory and Practice. Hoboken, NJ: John Wiley & Sons; 2013:437-466.
5. Polychronopoulos CD, Kuck DJ. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. IEEE Trans Comput. 1987;100(12):1425-1439.
6. Tzen TH, Ni LM. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. IEEE Trans Parallel Distrib Syst. 1993;4(1):87-98.
7. Hummel SF, Schönberg E, Flynn LE. Factoring: A method for scheduling parallel loops. Commun ACM. 1992;35(8):90-102.
8. Hummel SF, Schmidt J, Uma RN, Wein J. Load-sharing in heterogeneous systems via weighted factoring. In: Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures; 1996; Padua, Italy.
9. Banicescu I, Velusamy V, Devaprasad J. On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring. Cluster Computing. 2003;6(3):215-226.
10. Cariño RL, Banicescu I. Dynamic load balancing with adaptive factoring methods in scientific applications. J Supercomput. 2008;44(1):41-63.
11. Banicescu I, Liu Z. Adaptive factoring: a dynamic scheduling method tuned to the rate of weight changes. In: Proceedings of the High Performance Computing Symposium; 2000; Washington, DC.
12. Rice JR. The algorithm selection problem. Adv Comput. 1976;15:65-118.
13. Sukhija N, Malone B, Srivastava S, Banicescu I, Ciorba FM. Portfolio-based selection of robust dynamic loop scheduling algorithms using machine learning. In: Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium Workshops; 2014; Phoenix, AZ.
14. Casanova H, Giersch A, Legrand A, Quinson M, Suter F. Versatile, scalable, and accurate simulation of distributed applications and platforms. J Parallel Distrib Comput. 2014;74(10):2899-2917.
15. Boulmier A, Banicescu I, Ciorba FM, Abdennadher N. An autonomic approach for the selection of robust dynamic loop scheduling techniques. In: Proceedings of the 16th International Symposium on Parallel and Distributed Computing; 2017; Innsbruck, Austria.
16. Canon L-C, Jeannot E. Evaluation and optimization of the robustness of DAG schedules in heterogeneous environments. IEEE Trans Parallel Distrib Syst. 2010;21(4):532-546.
17. Mohammed A, Ciorba FM. A study of the performance of scientific applications with dynamic loop scheduling under perturbations. Paper presented at: Poster at 2018 Platform for Advanced Scientific Computing Conference (PASC); 2018; Basel, Switzerland.
18. Eleliemy A, Mohammed A, Ciorba FM. Efficient generation of parallel spin-images using dynamic loop scheduling. In: Proceedings of the 19th IEEE International Conference on High Performance Computing and Communications Workshops; 2017; Bangkok, Thailand.
19. Mandelbrot BB. Fractal aspects of the iteration of z → az (1-z) for complex Λ and z. Ann NY Acad Sci. 1980;357(1):249-259.
20. Mohammed A, Ciorba FM. SIL: An approach for adjusting applications to heterogeneous systems under perturbations. In: Euro-Par 2018: Parallel Processing Workshops: Euro-Par 2018 International Workshops, Turin, Italy, August 27-28, 2018, Revised Selected Papers. Cham, Switzerland: Springer; 2018.
21. Cariño RL, Banicescu I. A tool for a two-level dynamic load balancing strategy in scientific applications. Scalable Comput Pract Exp. 2007;8(3):249-261.
22. Tang P, Yew PC. Processor self-scheduling for multiple-nested parallel loops. In: Proceedings of the 1986 International Conference on Parallel Processing: August 19-22, 1986. Washington, DC: IEEE Computer Society Press; 1986.
23. Velho P, Legrand A. Accuracy study and improvement of network simulation in the SimGrid framework. In: Proceedings of the 2nd International Conference on Simulation Tools and Techniques; 2009; Rome, Italy.
24. Mohammed A, Eleliemy A, Ciorba FM. Performance reproduction and prediction of selected dynamic loop scheduling experiments. In: Proceedings of the 2018 International Conference on High Performance Computing and Simulation; 2018; Orléans, France.
25. Mohammed A, Eleliemy A, Ciorba FM, Kasielke F, Banicescu I. Experimental verification and analysis of dynamic loop scheduling in scientific applications. In: Proceedings of the 17th International Symposium on Parallel and Distributed Computing; 2018; Geneva, Switzerland.
26. Ali S, Maciejewski AA, Siegel HJ, Kim JK. Measuring the robustness of a resource allocation. IEEE Trans Parallel Distrib Syst. 2004;15(7):630-641.
27. Yang Y, Casanova H. RUMR: Robust scheduling for divisible workloads. In: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing; 2003; Seattle, WA.
28. Sukhija N, Banicescu I, Srivastava S, Ciorba FM. Evaluating the flexibility of dynamic loop scheduling on heterogeneous systems in the presence of fluctuating load using SimGrid. In: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum; 2013; Cambridge, MA.
29. Zhang Y, Voss M, Rogers ES. Runtime empirical selection of loop schedulers on hyperthreaded SMPs. In: Proceedings of the 19th International Parallel and Distributed Processing Symposium; 2005; Denver, CO.
30. Menon H, Chandrasekar K, Kale LV. POSTER: Automated load balancer selection based on application characteristics. In: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming; 2017; Austin, TX.
31. Rawlings JB. Tutorial overview of model predictive control. IEEE Control Syst Mag. 2000;20(3):38-52.
32. Mohammed A, Eleliemy A, Ciorba FM, Kasielke F, Banicescu I. An approach for realistically simulating the performance of scientific applications on high performance computing systems. Future Gener Comput Syst. 2019.
33. Ciorba FM. The importance and need for system monitoring and analysis in HPC operations and research. In: Richling S, Baumann M, Heuveline V, eds. Proceedings of the 3rd bwHPC-Symposium. Heidelberg 2016. Heidelberg, Germany; 2017:10. https://doi.org/10.11588/heibooks.308.418
34. Mehrotra R, Banicescu I, Srivastava S, Abdelwahed S. A power-aware autonomic approach for performance management of scientific applications in a data center environment. In: Handbook on Data Centers. New York, NY: Springer; 2015.
35. Montgomery DC. Design and Analysis of Experiments. Hoboken, NJ: John Wiley & Sons; 2017.
36. Mohammed A, Ciorba FM. SimAS: A simulation-assisted approach for the scheduling algorithm selection under perturbations. 2019. http://arxiv.org/abs/1912.02050
37. Penna PH, Castro M, Freitas HC, Broquedis F, Méhaut JF. Design methodology for workload-aware loop scheduling strategies based on genetic algorithm and simulation. Concurrency Computat Pract Exper. 2017;29(22). http://doi.org/10.1002/cpe.3933
38. Browne S, Dongarra J, Garner N, Ho G, Mucci P. A portable programming interface for performance evaluation on modern processors. Int J High Perform Comput Appl. 2000;14(3):189-204.
39. Gottscho M, Shoaib M, Govindan S, Sharma B, Wang D, Gupta P. Measuring the impact of memory errors on application performance. IEEE Comput Archit Lett. 2016;16(1):51-55.
40. Vergara Larrea VG, Joubert W, Brim MJ, et al. Scaling the Summit: deploying the world’s fastest supercomputer. In: High Performance Computing: ISC High Performance 2019 International Workshops, Frankfurt, Germany, June 16-20, 2019, Revised Selected Papers. Cham, Switzerland: Springer; 2019.

How to cite this article: Mohammed A, Ciorba FM. SimAS: A simulation-assisted approach for the scheduling algorithm selection under perturbations. Concurrency Computat Pract Exper. 2020;32:e5648. https://doi.org/10.1002/cpe.5648