Automated Scheduling Algorithm Selection and Chunk Parameter Calculation in OpenMP

Ali Mohammed, Jonas H. Müller Korndörfer, Ahmed Eleliemy, and Florina M. Ciorba

Abstract—Increasing node and cores-per-node counts in supercomputers render scheduling and load balancing critical for exploiting parallelism. OpenMP applications can achieve high performance via careful selection of scheduling `kind` and `chunk` parameters on a per-loop, per-application, and per-system basis from a portfolio of advanced scheduling algorithms (Korndörfer *et al.*, 2022). This selection approach is time-consuming, challenging, and may need to change during execution. We propose **Auto4OMP**, a novel approach for automated load balancing of OpenMP applications. With Auto4OMP, we introduce three scheduling *algorithm selection methods* and an *expert-defined chunk parameter* for OpenMP's `schedule` clause's `kind` and `chunk`, respectively. Auto4OMP extends the OpenMP `schedule(auto)` and *chunk* parameter implementation in LLVM's OpenMP runtime library to automatically select a scheduling algorithm and calculate a chunk parameter during execution. Loop characteristics are inferred in Auto4OMP from the loop execution over the application's time-steps. The experiments performed in this work show that Auto4OMP improves applications performance by up to 11% compared to LLVM's `schedule(auto)` implementation and outperforms manual selection. Auto4OMP improves MPI+OpenMP applications performance by *explicitly* minimizing thread- and *implicitly* reducing process-load imbalance.

Index Terms—Automatic selection, algorithm selection problem, dynamic load balancing, self-scheduling, runtime library, OpenMP, multithreaded programming, shared-memory systems

1 Introduction

Scientific computing is the cornerstone of computational science. The ever-increasing computational needs of scientific applications drive the advancements in modern high performance computing (HPC) systems, which exhibit increased parallelism at multiple levels. According to Top500 list [2], the number of processing elements has increased from a few cores-per-node to tens or hundreds of cores-per-node in the last decade. In addition to increasing parallelism, non-uniform memory access (NUMA) effects and dynamic power management of modern HPC systems introduce performance variability among the (homogeneous) computing cores of a node.

The complex characteristics of modern HPC systems challenge scientific applications in terms of harnessing their full computational power. Code branches, conditional statements, and data access patterns also cause application performance variability. These software and hardware sources of variability typically lead to uneven computing nodes and cores finishing times, which is known as *load imbalance* and which may severely degrade applications performance on massively parallel HPC systems.

Scheduling and *load balancing* were identified as major challenges on the road to Exascale and beyond [3]. Scheduling algorithms typically balance the execution of work across parallel computing elements, i.e., in space. Nevertheless, load imbalance may still arise among cores of a node and among computing nodes due to performance variability in either application or system during execution, i.e., over time. Node-level load imbalance may significantly influence cross-node load imbalance [4], [5]. In this work, we concentrate on node-level scheduling and load balancing of OpenMP parallel loops.

OpenMP is the most widely- and successfully-used parallel programming paradigm for shared-memory architectures [6]. Parallel loop scheduling in OpenMP is governed by the `schedule(kind,chunk)` clause. The OpenMP standard specifies five options for `schedule(kind)`, three of which denote scheduling algorithms: `static`, `dynamic`, and `guided`, and two refer to convenience options: `runtime` and `auto`.

It has been shown that these three scheduling algorithms in OpenMP are insufficient for efficient scheduling of OpenMP parallel loops and that other scheduling algorithms deliver higher performance gains [1], [7], [8], [9]. However, those performance gains are achievable only via a careful (and expert) selection of scheduling algorithm and `chunk` parameter, on a *per-loop*, *per-time-step*, *per-application*, and *per-system* basis.

Choosing a loop scheduling algorithm among those available in a particular OpenMP runtime library is an instance of the *algorithm selection problem* [10]. Users must select one

• *Ali Mohammed is with the HPE's HPC/AI EMEA Research Lab (ERL), 4051 Basel, Switzerland. E-mail: ali.mohammed@hpe.com.*
• *Jonas H. Müller Korndörfer, Ahmed Eleliemy, and Florina M. Ciorba are with the Department of Mathematics and Computer Science, University of Basel, 4051 Basel, Switzerland. E-mail: {jonas.korndorfer, ahmed.eleliemy, florina.ciorba}@unibas.ch.*

Manuscript received 10 September 2021; revised 21 June 2022; accepted 29 June 2022. Date of publication 11 July 2022; date of current version 23 August 2022. This work was supported in part by the Swiss National Science Foundation in the context of the "Multi-level Scheduling in Large Scale High Performance Computers" (MLS) under Grant 169123, in part by the Swiss Platform for Advanced Scientific Computing (PASC) project "SPH-EXA2: Smoothed Particle Hydrodynamics at Exascale", by DAPHNE, and in part by European Union's Horizon 2020 research and innovation programme under Grant 957407.
(Corresponding author: Florina M. Ciorba.)
Recommended for acceptance by S. Chandrasekaran.
Digital Object Identifier no. 10.1109/TPDS.2022.3189270

scheduling algorithm among dozens of algorithms, each with various characteristics; they must also select a suitable chunk parameter among prohibitively numerous options, for each loop, for each application, and for each system on which an application executes. Increasing the number of scheduling choices may impede the usability of the library and may lead to sub-optimal choices or even *decision paralysis* [11].

The OpenMP standard offers, since version 3.0 [12], `auto` as a `kind` parameter for the `schedule()` clause to avoid the scheduling algorithm choice paralysis. With `auto`, the scheduling decision is delegated to the compiler/runtime implementation [13]. The idea behind `auto` is to give the implementation the freedom of choosing any possible space-time assignment of iterations in a loop construct to the team of threads of the parallel region encapsulating the loop, while remaining standard-compliant. In this sense, OpenMP `auto` is a descriptive rather than prescriptive scheduling option. Currently, most OpenMP implementations do not fully leverage OpenMP `auto` to improve application performance. The implementation of `auto` in the GNU OpenMP runtime library [14] maps to `static` scheduling. The `auto` implementation into LLVM's OpenMP runtime library, compatible with Clang and Intel compilers, maps to an optimized implementation of guided self-scheduling. By introducing the `auto` scheduling option, OpenMP was ahead of its time in addressing the algorithm selection problem, given that only few scheduling algorithm choices were available in any OpenMP standard-compliant compiler/runtime library.

In this work, we leverage the existence of `auto` as a scheduling option in OpenMP and extend its implementation in the LLVM OpenMP runtime library with an *`expert chunk`* parameter (see Section 3.1) and algorithm selection methods (see Section 3.3). The automatic selection methods select a scheduling algorithm from the Auto4OMP portfolio (see Section 3.2), a subset of the portfolio of loop scheduling algorithms presented in prior work [1]. We refer to these extensions collectively as Auto4OMP.

Auto4OMP, introduces three novel *algorithm selection methods*: `RandomSel` , `ExhaustiveSel` , and `ExpertSel` to address the scheduling algorithm selection problem in time-stepping OpenMP loops. The three selection methods leverage application and system information obtained during execution for the automated and dynamic selection of scheduling algorithm during execution. Auto4OMP also introduces the `expert chunk` parameter calculated based on the number of loop iterations and number of threads. The `expert chunk` can be used with all scheduling algorithms either on their own or with those selected by the algorithm selection methods in Auto4OMP .

We conducted an extensive performance analysis campaign with Auto4OMP on five scientific applications (ALYA [15], GROMACS [16], Mandelbrot [17], SPEC OMP 2012 352.nab [18], and SPHYNX [19]) executed on three shared-memory systems with variable core architectures and counts. The results show that *Auto4OMP* improves performance by up to 11% compared to LLVM's `schedule (auto)` implementation and outperforms manual selection. Auto4OMP also improves performance of large scale MPI +OpenMP applications executing on multiple nodes, by explicitly minimizing thread- and implicitly minimizing process-load imbalance [5].

Significance. Auto4OMP is a unique opportunity to fulfil the true potential of the `schedule(auto)` option, in general, and of its implementation in LLVM's OpenMP runtime library, in particular. Auto4OMP unburdens the user of the scheduling algorithm and `chunk` parameter selection problem at application-, loop-, time-step-, and system-level, and enables *automated selection of scheduling algorithms* based on performance information available exclusively during execution. By design, Auto4OMP does not require any user input or intervention in pre- or post-processing nor any changes to the application. By consequence, Auto4OMP is a significant step towards adaptive execution of OpenMP loops, supporting performance portability of OpenMP applications.

Contributions. This work is the first to leverage the potential of OpenMP's `schedule(auto)` and makes the following contributions, collectively called Auto4OMP:

1) Introduces the `expert chunk` parameter, designed as a practical solution that works without requiring detailed knowledge of the target loops characteristics.
2) Introduces three new methods for automated scheduling algorithm selection in OpenMP.
3) Implements the selection methods and `expert chunk` parameter in the LLVM OpenMP runtime library as extensions[1] to `schedule(auto)`.

The remainder of this work is organized as follows. The work related to Auto4OMP is discussed in Section 2. Section 3 presents the design, implementation, and usage of Auto4OMP, including details of the algorithm selection methods and the `expert chunk` parameter. The results of a comprehensive performance evaluation and their analysis are presented and discussed in Section 4. The work is concluded in Section 5, outlining limitations and directions for future work.

2 Related Work

Numerous loop scheduling algorithms were implemented in various OpenMP runtime libraries and made publicly available for OpenMP users. For instance, LB4OMP [1], an extended version of the LLVM OpenMP runtime library[2] (RTL), supports various dynamic loop scheduling (DLS) algorithms, including fixed size chunking (`FSC`) [20], factoring (`FAC`) [21], the practical variant of factoring (`FAC2`), tapering (`TAP`) [22], the practical variant of weighted factoring (`WF2`) [23], `BOLD` [24], adaptive weighted factoring (`AWF`), its variants (`AWF-B,C,D,E`) [25], adaptive factoring (`AF`) [26], `mFAC` and `mAF` (versions of `FAC` and `AF` with less overhead). Such a variety of many scheduling algorithms may bring users to face decision paralysis [11]. In addition, it may even be impractical to manually select the highest performing scheduling algorithm and a suitable chunk parameter for a given application (in each time-step, for each loop within a time-step) and system. Therefore, the scheduling algorithm selection problem has been recognized and addressed in the

1. Auto4OMP https://zenodo.org/record/583414
2. https://github.com/unibas-dmi-hpc/LB4OMP

literature via decision trees [27], exhaustive offline search [28], simulation-assisted selection [29], [30], and machine learning-based selection [31], [32].

Banicescu *et al.* [31] and Boulmier *et al.* [32] proposed and evaluated the use of reinforcement learning (RL) algorithms. Both approaches were tested for scheduling various MPI applications [31], [32], in direct experiments with a single application and a single system [31] or in simulation [32], using the SimGrid-SMPI [33] simulator.

The main methodological difference between the present and prior work [31], [32], [34] is the usage of an *expert-based* scheduling algorithm selection in OpenMP. The advantage of an expert-based selection approach is considering previous knowledge and expertise about the load balancing capabilities of the scheduling algorithms in solving the algorithm selection problem.

The above approaches [31], [32], [34] tested the selection methods only with simulation, synthetic applications, or single application-system configurations. In contrast, in this work, we evaluate the proposed selection methods via native experiments considering five scientific applications and three shared-memory systems.

Zhang *et al.* [27] targeted OpenMP applications executing on SMPs and organize the threads on two levels: physical cores and hyperthreads within a physical core. They proposed three empirical methods for the selection of the ideal number of threads and scheduling algorithms from a set that includes five scheduling algorithms. With each empirical method, once a scheduling algorithm is selected during execution, it will persist until execution completes. In the present work, the selection of scheduling algorithms dynamically adapts to performance variation during the execution of an application. Our approach does not require any pre-computed information, and the information collected during execution only consists of thread execution times, with negligible overhead.

Thoman *et al.* [35] proposed a selection method using polyhedral model [36] which takes into account compiler analysis and a load-aware runtime system to select the highest performing scheduling algorithm only among the OpenMP standard-compliant algorithms. Their approach requires certain information (compiler analysis) to be collected prior to the execution. Auto4OMP selection methods do not require any information to be collected prior to the execution.

Sreenivasan *et al.* [28] proposed a framework for identifying the highest performing scheduling algorithm in OpenMP loops considering `static` or `dynamic`, and `chunk` parameter values: 1, 8, or 16 iterations for a given application-loop-system combination. Their framework automatically generates multiple instances of the same applications with different `schedule(kind,chunk)` clauses and number of threads configuration. The selection is then conducted offline through exhaustive experimentation of the automatically generated instances of the application. Our approach does not need offline experimentation as the algorithm selection process is dynamic and adaptive during the execution.

Mohammed *et al.* [30] proposed a simulation-assisted selection of scheduling algorithms for MPI applications during execution. They used simulation, with SimGrid [33], to predict a loop' performance with different scheduling algorithms before it was executed, and selected the scheduling algorithm that the simulation predicted would achieve the highest performance. In their approach, the simulator required significant amounts of information, such as the number of floating-point operations of each loop iteration, to be available before the execution. Our approach employs direct execution and requires no prior information as input.

All these approaches [27], [28], [30], [31], [32], [34], [35] either require user intervention for (semi-automated) preprocessing, information from hardware counters, or are not fully adaptive during execution to react to performance variability during execution. In contrast, Auto4OMP is fully dynamic and adaptive, offered as a lightweight solution for automated scheduling algorithm selection for OpenMP applications with parallel loops.

3 AUTO4OMP DESIGN AND IMPLEMENTATION

Auto4OMP is designed to address the *scheduling algorithm selection problem* for OpenMP applications. It leverages the existence of `auto` as a scheduling option in OpenMP and extends its implementation in LB4OMP [1], an extended LLVM OpenMP runtime library.

3.1 Expert Chunk Parameter

The `chunk` parameter was introduced in the OpenMP standard to minimize the scheduling overhead and to improve data locality. Nevertheless, the use of the `chunk` parameter bears different meanings among scheduling algorithms. For `schedule(static,chunk)` and `schedule (dynamic, chunk)`, the `chunk` parameter denotes the amount of iterations that the threads receive per work request. For the other algorithms, the `chunk` parameter works as a *threshold*, such that when the chunk sizes calculated by a scheduling algorithm fall below this threshold, they are simply given the value of the `chunk` parameter. The rationale is to reduce the overhead of assigning very small chunk sizes to threads. Therefore, declaring a proper `chunk` parameter is expected to improve performance as threads require fewer scheduling rounds to complete a loop than without this threshold.

The `chunk` parameter leading to the highest performance for a given loop is a choice between an empirical observation based on extensive experimentation and an expert choice. To illustrate the impact of the `chunk` parameter choice, Fig. 1 shows the execution time of loop $L1$ from SPHYNX (a hydrodynamics simulation code) [37] on a miniHPC-Broadwell node (see Section 4 for more details) scheduled with the standard OpenMP scheduling algorithms, `STATIC`, `SS=dynamic`, and `GSS=guided`, with decreasing `chunk` parameter values in the interval $[N/(2P),..,1]$ loop iterations, where $N = 1,000,000$ denotes the number of loop iterations and $P = 20$ the number of OpenMP threads. We see that the performance follows a bathtub curve, with the `chunk` parameter value leading to highest performance, regardless of scheduling `kind`, around the middle of the interval, namely at 48 loop iterations in this example.

We examined the performance of several applications by varying the `chunk` parameter value, the scheduling algorithms, loops, and computing systems. Similar performance was observed in experiments with other applications [1],

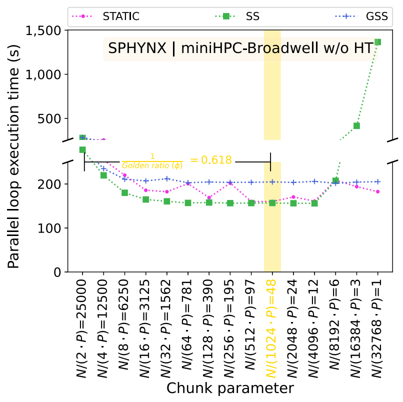

Fig. 1. Performance of SPHYNX 's $L1$ loop on a miniHPC-Broadwell node without hyperthreading (see Table 2). Using the golden ratio (Eq. (1)), the expert `chunk` parameter is 48 iterations.

the results of which can be found online.[3] These experiments led to identifying the `expert chunk`, derived as Eq. (1), as a practical value that improves performance in a large majority cases. The `expert chunk` parameter is a practical method that uses the golden ratio $\phi = 1.618$ [38] in the interval $[N/(2P), ..., 1]$ (see Eq. (1) and Fig. 1) to arrive at a chunk parameter value that leads to high performance, using only N, P, and ϕ.

$$\textbf{chunk}_{\textbf{expert}} = \left\lfloor \frac{N}{2^f \times 2P} \right\rfloor, \text{ where } f = \left\lfloor log_2\left(\frac{N}{P}\right) \times \frac{1}{\phi} \right\rfloor \tag{1}$$

3.2 Auto4OMP Portfolio

We define three inclusion characteristics for scheduling algorithms to build a portfolio of scheduling algorithms for Auto4OMP. Selection methods in Auto4OMP use this portfolio (a subset of scheduling algorithms in LB4OMP) to help reduce the search space and selection cost.

C1. *A scheduling algorithm should not require user input or profiling information prior to loop execution.* The rationale is that Auto4OMP should work seamlessly and should not require any effort from the user or additional input, similar to OpenMP standard's `schedule(auto)`.

C2. *A scheduling algorithm should use lightweight synchronization mechanisms to reduce the scheduling overhead.*The rationale is to avoid experimentation with inefficient synchronization implementations, which degrade overall performance.

C3. *A scheduling algorithm can adapt its decisions during any time-step but should not require further time-steps to adapt an application's performance.*The rationale is that Auto4OMP examines each loop's performance at the end of every time-step to update the algorithm selection. Therefore, a scheduling algorithm that requires more than a single time-step to adapt the scheduling of that loop is not be suitable for a portfolio targeted for automatic selection of scheduling algorithms.

To build the Auto4OMP portfolio, we apply the three inclusion characteristics to the 21 scheduling algorithms from LB4OMP [1], which comprises 15 dynamic and adaptive scheduling algorithms: `FSC`, `FAC`, `mFAC`, `FAC2`, `mFAC2`, `WF2`, `TAP`, `BOLD`, `AWF`, `AWF-B`, `AWF-C`, `AWF-D`, `AWF-E`, `AF`, `mAF`, the 3 scheduling algorithms in the OpenMP standard: `STATIC` or `static`, self-scheduling (`SS`) or `dynamic`, and guided self-scheduling (`GSS`) or `guided`, as well as 2 algorithms that are not specified in the OpenMP standard: trapezoidal self-scheduling (`TSS`) and `Static Steal`. We also consider `schedule(auto)` from LLVM's OpenMP runtime library, which defaults to `guided_analytical_-chunked` (`GAC`), which is a variation of `GSS`.

3. https://doi.org/10.5281/zenodo.6309015

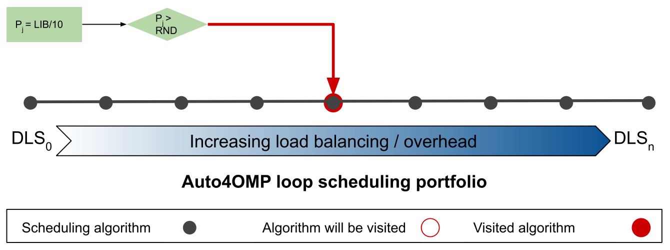
P_j = LIB/10
P_j > RND
DLS0
Increasing load balancing / overhead
DLSn
Auto4OMP loop scheduling portfolio
Scheduling algorithm ●
Algorithm will be visited ○
Visited algorithm ●

Fig. 2. `RandomSel` scheduling algorithm selection method. `RandomSel` randomly selects a new scheduling algorithm if P_j is greater than RND.

From these 21 scheduling algorithms, we include 12 in Auto4OMP's portfolio, according to the three inclusion criteria. The 12 scheduling algorithms are *ordered* in increasing order of their scheduling overhead and load balancing capacity: [4] `STATIC`, `SS`, `GSS`, `GAC`, `TSS`, `Static Steal`, `mFAC2`, `AWF-B`, `AWF-C`, `AWF-D`, `AWF-E`, and `mAF`. Auto4OMP employs by default the `expert chunk` parameter for each algorithm selected from its portfolio, unless the user overrides this option (clarification in Section 3.4.1).

3.3 Automated Selection Methods

3.3.1 `RandomSel`

This method selects a scheduling algorithm in a random fashion, without searching for the highest performing option. `RandomSel` , illustrated in Fig. 2, defines the jump probability P_j, indicating the probability to *change* the selected scheduling algorithm in the interval DLS_0 = `STATIC` and DLS_n = `mAF`. Upon each loop execution instance (time-step), `RandomSel` randomly generates a number RND, between 0 and 1. If P_j is higher than RND, a scheduling algorithm will randomly be selected from the Auto4OMP portfolio. Otherwise, the currently selected scheduling algorithm will be kept. P_j is calculated as $LIB/10$, where LIB denotes the percent load imbalance metric [39] calculated as in Eq. (2), and 10 is an arbitrary constant. Therefore, if the current LIB is greater than 10%, `RandomSel` is guaranteed to select a new scheduling algorithm. $LIB = 10$ was empirically found to be a suitable threshold to represent high load imbalance as observed in loop executions, where possible load balancing performance gains surpass the cost of changing the scheduling algorithm.

$$LIB = \left(1 - \frac{\texttt{mean} \text{ of thread finishing times}}{\texttt{max} \text{ of thread finishing times}}\right) \times 100. \tag{2}$$

`RandomSel` has the advantage of directly selecting a scheduling algorithm without the need for trying several

4. Scheduling overhead and scheduling capacity of scheduling algorithms can be inferred from their derivation, implementation, and performance benchmarking. `STATIC` has the lowest overhead and load balancing capacity while `mAF` represents the other extreme.

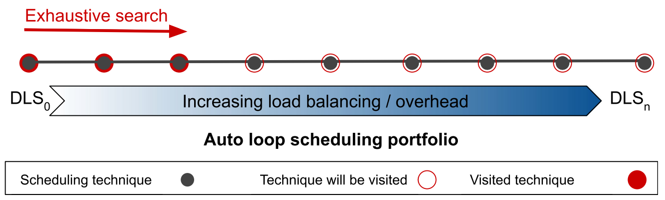
Exhaustive search
DLS0
Increasing load balancing / overhead
DLSn
Auto loop scheduling portfolio
Scheduling technique
Technique will be visited
Visited technique

Fig. 3. `ExhaustiveSel` scheduling algorithm selection method. `ExhaustiveSel` visits all the scheduling algorithms in the Auto4OMP portfolio, one algorithm per time-step. Then, it selects the scheduling algorithms that resulted in the shortest loop execution time.

algorithms before making a selection. The disadvantage is that the random selection may result in poor performance and high LIB. Thanks to P_j, a poorly performing scheduling algorithm will be changed in the next loop execution instance.

3.3.2 `ExhaustiveSel`

This method conducts an exhaustive search, i.e., trying all algorithms in the Auto4OMP portfolio, one algorithm per time-step, before selecting the algorithm that achieves the shortest loop execution time (see Fig. 3, where DLS$_0$= `STATIC` and DLS$_n$ = `mAF`). This comes at the cost of a number of trials (over time-steps) proportional to the length of the Auto4OMP portfolio. `ExhaustiveSel` does not repeat the same loop time-step with all scheduling algorithms in the Auto4OMP portfolio. Instead, it changes the selected scheduling algorithm each time the loop is executed and keeps a record of the shortest execution time and the corresponding selected algorithm. This way, trials of various scheduling algorithms progress application execution. After testing all algorithms in Auto4OMP portfolio, `ExhaustiveSel` selects the one that achieved the shortest execution time. `ExhaustiveSel` has the advantage of finding the highest performing scheduling algorithm for a loop. The caveat is comparing the loop execution times of different scheduling algorithms over different time-steps, i.e., comparing a loop's execution time with `STATIC` at time-step 0 with the loop's execution time with `SS` at time-step 1. If the loop execution time varies greatly across time-steps, this may lead to sub-optimal selection.

Once selected, the scheduling algorithm that achieved the shortest loop execution time is employed for scheduling the loop iterations over the remaining time-steps. The LIB (see Eq. (2)) keeps being calculated after every loop execution instance. If the loop execution becomes highly imbalanced with the selected scheduling algorithm, the exhaustive search is re-triggered to select a more suitable algorithm.

3.3.3 `ExpertSel`

This method directly selects a suitable scheduling algorithm per loop execution instance, like `RandomSel`. In contrast to `RandomSel`, `ExpertSel` uses fuzzy logic [40] and expert rules to make this selection. `ExpertSel` benefits from performance information collected during execution to make an *expert-based selection* of scheduling algorithms.

In `ExpertSel`, the first instance of the loop is executed with `STATIC` to collect initial information, i.e., loop execution time (T_{par}) and load imbalance (LIB). Two fuzzy systems are designed and illustrated in Fig. 4. The first one is

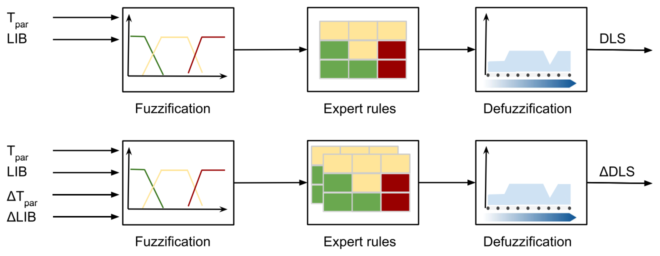

Fig. 4. `ExpertSel` scheduling algorithm selection method. The top fuzzy system selects a suitable initial scheduling algorithm, while the bottom fuzzy system decides how to change the selected scheduling algorithm based on T_{par} and LIB and their variations ΔT_{par} and ΔLIB.

used on the second execution instance to select, based on the initial T_{par} and LIB, an initial scheduling algorithm. The second one is used in later execution instances when the change of loop execution time (ΔT_{par}) and load imbalance (ΔLIB) can be calculated, using the difference between T_{par} and LIB values on the current and previous time-step, respectively.

Fuzzy logic has the advantage of reducing the complexity of problems that deal with uncertainties. It provides a simple approach to formulate expert knowledge as a set of rules, that help decide on actions based on uncertain (non-crisp) inputs. The selection process using fuzzy logic [41] includes three steps: 1. Fuzzification; 2. Evaluation of expert rules; and 3. Defuzzification.

Fuzzification: It is the process of classifying the input or output according to its value. Fuzzification encapsulates the uncertainty of the absolute "crisp" values and gives them a meaning that can be later used to reason about in the expert rules. The inputs are classified according to membership functions (see Fig. 5). Based on its value, the input is assigned a membership value for each membership function of that input. For example, an LIB value of 1.5% would be considered 0.5 `Low` and 0.5 `Moderate` load imbalance. The trapezoidal shapes are commonly used for the membership functions as they simplify calculations of fuzzification/defuzzification and represent the overlap (uncertainty) of crisp to fuzzy conversion. Values used to define the membership functions were defined based on our expertise and were tuned during the design of the fuzzy system together with the fuzzy rules such that `ExpertSel` reacts to load imbalance and loop execution time changes adequately. As such, `ExpertSel` encodes our scheduling expertise to enable OpenMP of automated scheduling algorithm selection.

Expert Rules: These are rules that represent how an expert would select an output given certain inputs. These rules use the fuzzy inputs and outputs, i.e., the membership values. The evaluation of each rule gives a truth value per rule. Expert rules use the fuzzy operators (`and` and `or`) to combine inputs and calculate the output truth value. The fuzzy operators, `and` and `or`, are equivalent to `MIN` and `MAX`, respectively. We evaluate the rules using `MIN-MAX`, i.e., we combine the membership values of the inputs using the `MIN` operator to calculate the truth value of each rule (inference step). Then, all truth values of a single fuzzy output are combined using `MAX` operator (composition step).

Table 1 shows the expert rules to select a scheduling algorithm for a loop at the beginning of the execution. The rule at the top leftmost green cell of the table is translated to *"If T_{par} is `Short`and LIB is `Low`, then DLS is `Simple`"*.

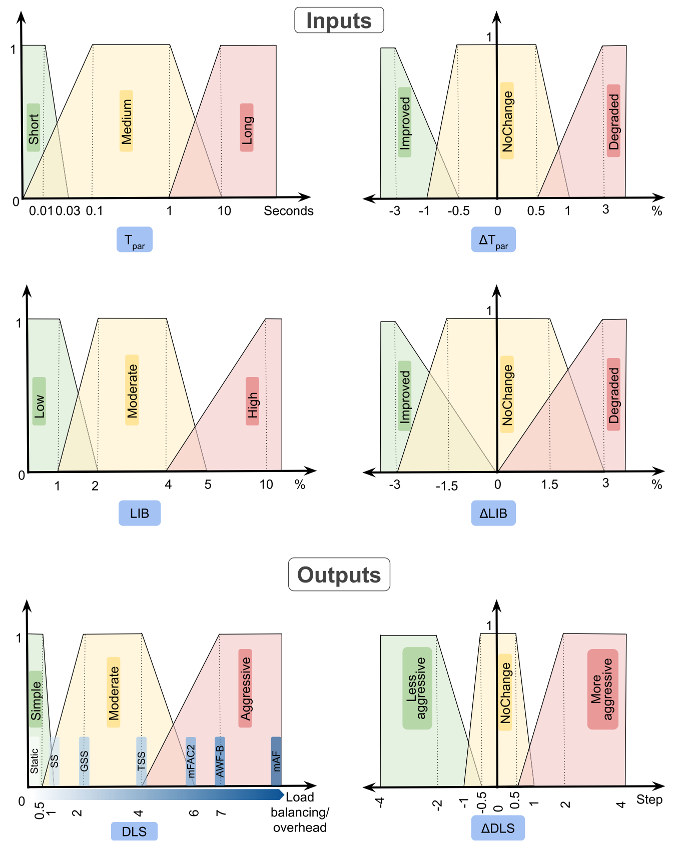

Fig. 5. Fuzzy membership functions for inputs and outputs used for the fuzzification process of `ExpertSel` .

The rules for intelligent subsequent changes of the scheduling algorithm selection are shown in Algorithm 1. These rules depend on four inputs, compared to just two as those in Table 1.

Algorithm 1: Expert rules for changing the scheduling algorithm selection during execution

```
// Rules for $\Delta DLS$ Less aggressive
if ($\Delta T_{par}$ is Degraded) and ($\Delta LIB$ is Improved) or
if ($\Delta T_{par}$ is Degraded) and ($\Delta LIB$ is NoChange) or
if ($\Delta T_{par}$ is NoChange) and ($\Delta LIB$ is Improved) or
if ($T_{par}$ is Short) or
if ($\Delta T_{par}$ is Degraded) and ($LIB$ is Low)
then $\Delta DLS$ is Less aggressive
// Rules for $\Delta DLS$ NoChange
if ($\Delta T_{par}$ is NoChange and ($\Delta LIB$ is NoChange) or
if ($LIB$ is Low) or
if ($\Delta T_{par}$ is Improved )
then $\Delta DLS$ is NoChange
// Rules for $\Delta DLS$ More aggressive
if ($LIB$ is High) and ($T_{par}$ is Long) or
if ($\Delta T_{par}$ is Degraded ) and ($\Delta LIB$ is Degraded)
or
if ($\Delta T_{par}$ is NoChange and ($\Delta LIB$ is Degraded)
then $\Delta DLS$ is More aggressive
```

Defuzzification: It is the process of converting the output membership values to a "crisp" output again to select a scheduling algorithm [42]. We use Centroid [43], the most common defuzzification method, to calculate the center of gravity of the membership functions for the output fuzzy membership values. After combining the rules and determining the output membership value in each fuzzy set, the membership functions are cut at their membership values. Then, the areas under the trapezoids (membership functions) are calculated by integration. These areas are normalized to identify the contribution of each fuzzy set in the output. The normalized areas are then multiplied by the center of their respective membership functions and are summed together to obtain a crisp output.

TABLE 1
Expert Rules for Algorithm Selection for a Loop

T_{par} \ LIB	**Low**	**Moderate**	**High**
Short	Simple	Simple	Simple
Medium	Simple	Moderate	Moderate
Long	Simple	Moderate	Aggressive

3.4 Implementation

The three main functions: `init`, `next`, `finish`, are responsible for scheduling loop iterations, in the file `kmp_dispatch.cpp`, as illustrated in Fig. 6.

Upon OpenMP parallel loop initialization, each thread calls the `__kmp_dispatch_init_algorithm` function (`init` in Fig. 6) inside the `kmp_dispatch.cpp` file. This function initializes the structures needed for the selected scheduling algorithm and calls `__kmp_dispatch_next_algorithm` (`next` in Fig. 6). Most code changes related to Auto4OMP are in the `init` function to change the *chunk parameter* or the selected scheduling algorithm.

The logic of the chunk calculation of all DLS algorithms is in the `__kmp_dispatch_next_algorithm` function, which is called every time a thread attempts to obtain work. Finally, the threads call `__kmp_dispatch_finish` (`finish` in Fig. 6) to reset variables or free allocated memory.

3.4.1 *Expert chunk parameter*

The `expert chunk` parameter is controlled by the newly introduced environment variable **`KMP_Expert_Chunk`**. A newly implemented function, `expertChunk`, calculates the `expert chunk` parameter (using Eq. (1)) and sets it for this loop if the variable is set to 1. The `expertChunk` is called from the `init` function before initializing the selected scheduling algorithm. If automated scheduling algorithm selection

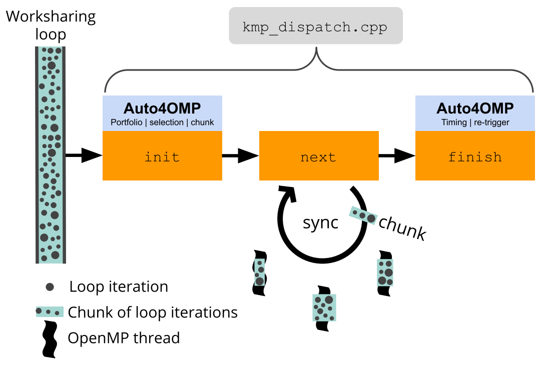

Fig. 6. Loop scheduling workflow of the standard LLVM OpenMP RTL augmented with Auto4OMP.

TABLE 2
Design of Factorial Experiments for the Performance Evaluation of Auto4OMP

Factors		Values		Properties
Applications		ALYA		N = 16, 677, 401 | T = 200 | Total loops = 133 | Modified loops = 12
		GROMACS		N = 3, 316, 460 | T = 10, 001 | Total loops = 90 | Modified loops = 1
		Mandelbrot		N = 262, 144 | T = 200 | Total loops = 3 | Modified loops = 3
		SPEC OMP 2012 352.nab		N = 44, 794 | T = 1, 002 | Total loops = 13 | Modified loops = 7
		SPHYNX Evrard collapse		N = 1, 000, 000 | T = 200 | Total loops = 37 | Modified loops = 2 — #instances/time-steps $L0 = 2$ — #instances/time-steps $L1 = 1$
Selection	Approaches	LLVM `schedule(auto)`	`GAC`	A variant of guided self-scheduling
		Manual selection*	`ManualBest`	A non-automatic selection of scheduling algorithms for each loop with the default `chunk` parameter. It could be determined only after exhaustive experimentation.
			`Oracle`	Ideal combination of the best choices of DLS algorithm selection across application loops, time-steps, and `chunk` parameter. Calculated theoretically after exhaustive experimentation of all scheduling algorithms.
		Auto4OMP**	`RandomSel` `ExhaustiveSel` `ExpertSel`	**Automated** scheduling algorithm selection across application loops and time-steps
	Chunk parameter	Default		Chunk size =N/P for static and 1 for all other scheduling algorithms
		Auto4OMP	`Expert chunk`	a point at $\frac{1}{Golden\ ratio(1.618)} = 0.618$ on the curve between $N/(xP)$ and 1, with x increasing in steps of 2
Computing nodes		miniHPC-Broadwell		Intel Xeon E5-2640 v4 (2 sockets, 10 cores each) P = 20 without hyperthreading, Pinning:`OMP_PLACES` = cores `OMP_PROC_BIND` = close
		Piz Daint-Haswell		Intel Xeon E5-2690 v3 (1 socket, 12 cores) P = 12 without hyperthreading, Pinning:`OMP_PLACES` = cores `OMP_PROC_BIND` = close
		miniHPC-Cascade-Lake		Intel Xeon Gold 6258R (2 sockets, 28 cores each) P = 56 without hyperthreading, Pinning: `OMP_PLACES` = cores `OMP_PROC_BIND` = close

$*$ *Selects a scheduling algorithm among: {`STATIC`, `SS`, `GSS`, `TSS`, `Static Steal`, `FAC2`, `mFAC2`, `AWF`, `AWF-B`, `AWF-C`, `AWF-D`, `AWF-E`, `AF`, `mAF`, `GAC`.*
$**$ *Selects a scheduling algorithm from Auto4OMP's portfolio: {`STATIC`, `SS`, `GSS`, `GAC`, `TSS`, `Static Steal`, `mFAC2`, `AWF-B`, `AWF-C`, `AWF-D`, `AWF-E`, `mAF`}. Employs the `expert chunk` for the selected scheduling algorithm.*

is active, then the `expert chunk` parameter is also set by default. The user can override this behavior by explicitly exporting `KMP_Expert_Chunk=0` when using automated scheduling algorithm selection. However, disabling the expert chunk will influence the selection methods, in ways that deserve further study and are planned as future work.

3.4.2 Loop Information

It is crucial to save certain information about the loop across its execution instances, since an application usually contains many loops that are executed many times (mostly as time-steps). Therefore, we use a hash-map data structure to hold loop information, such as current selected scheduling algorithm, previous scheduling algorithm, number of search trials, `chunk` parameter value, current loop execution time, and previous loop execution time. Whenever a loop is encountered for the first time, the data structure is created and initialized by the `init` function, if the automated scheduling selection is activated.

3.4.3 Activation of Automated Selection

`auto` is already a valid scheduling option for OpenMP's `schedule` clause and can be exported using the `OMP_-SCHEDULE` environment variable. To distinguish between the three proposed automated selection methods: `Random-Sel` , `ExhaustiveSel` , and `ExpertSel` , we also read an additional parameter when setting `OMP_SCHEDULE=auto, method`.

The `method` parameter is used to select one of the automated selection methods above, i.e., setting `OMP_SCHEDULE` to `auto,2` selects `RandomSel` , whereas `auto,1` selects `GAC` scheduling (the original LLVM OpenMP runtime implementation). If the `method` parameter does not map to a valid selection method, i.e., outside the range $2 - 4$, Auto4OMP defaults to LLVM's `auto` scheduling, `GAC`.

Upon the selection of a valid automated selection option, the automated selection is activated, which: (1) Enables special timers to measure loop execution time and load imbalance (see below). (2) Initializes the loop information data structure whenever encountering a new loop; (3) Begins the search of the highest performing scheduling algorithm for this loop; (4) Enables the `expert chunk` parameter and sets it by default. Steps (2) and (3) are only performed by the first thread to avoid data races.

3.4.4 Timing and Re-Trigger

Timers are placed around the loop region, which starts in the `init` and ends in the `finish` functions in Fig. 6, to estimate loop execution time T_{par}. The timers use lightweight timing calls to read the time stamp register (rdtscp) and divide it by the CPU frequency read from system information to estimate the execution time. Also, the finishing time of each thread is measured to calculate the load imbalance LIB of the loop execution (see Eq. (2)). T_{par} and LIB are recorded in the loop information structure in the `finish` function of the loop.

If the LIB is higher than the previous execution instance, by a certain threshold, i.e., 10% in the current implementation, the automated search for the highest performing scheduling algorithm is re-triggered again for `Exhausti-veSel` .

4 Performance Evaluation

We designed a set of performance evaluation experiments (a total of $2'700$), described in Table 2, to test the following hypotheses:

H.1 Auto4OMP achieves high performance and provides the smallest variation of performance across application-system pairs.
H.2 Auto4OMP *adapts* to the various scheduling needs of applications when they execute on different systems.
H.3 The use of the `expert chunk` parameter improves application performance at no additional cost.
H.4 Auto4OMP *adapts* to the various scheduling needs of applications *across time-steps for each loop* to achieve high performance.
H.5 Auto4OMP *adapts* to the various scheduling needs of applications' various loops within *a single time-step* (and across time-steps).
H.6 Reducing OpenMP thread-level load imbalance improves overall performance of hybrid process+thread parallel applications (MPI+OpenMP).

TABLE 3
H1. Comparison Prior Work (no or Manual Selection) and Auto4OMP in Terms of Performance Degradation (%) Relative to `Oracle`

Selection / App-Sys pair	Prior work (no or manual selection)			Auto4OMP				`Oracle exec. time`
	`Static Steal`	`GAC (LLVM-auto)`	`ManualBest`	`SS,expert chunk`	`RandomSel`	`ExhaustiveSel`	`ExpertSel`	
ALYA-Piz Daint-Haswell	11.48%	2.44%	2.03%	2.93%	0.29%	0.23%	0.89%	11,492s
GROMACS-miniHPC-Broadwell	22.11%	0.44%	0.11%	1.42%	1.18%	0.56%	0.67%	1,954s
GROMACS-Piz Daint-Haswell	52.12%	0.27%	0.17%	5.26%	0.75%	0.49%	1.10%	22,248s
Mandelbrot-miniHPC-Broadwell	0.36%	0.38%	0.14%	0.02%	0.54%	0.36%	0.87%	1,503s
Mandelbrot-Piz Daint-Haswell	0.20%	0.37%	0.09%	0.13%	0.41%	0.37%	0.45%	8,394s
Mandelbrot-miniHPC-Cascade-Lake	0.46%	0.45%	0.35%	0.31%	1.20%	1.64%	1.42%	1,503s
SPEC OMP 2012 352.nab-miniHPC-Broadwell	4.64%	3.08%	0.20%	2.91%	2.83%	0.96%	2.42%	825s
SPEC OMP 2012 352.nab-Piz Daint-Haswell	3.84%	2.33%	1.59%	2.58%	2.30%	1.99%	2.13%	1,191s
SPHYNX-miniHPC-Broadwell	22.01%	10.36%	5.43%	0.02%	13.48%	1.58%	0.55%	3,863s
SPHYNX-Piz Daint-Haswell	38.81%	12.38%	3.04%	0.11%	15.89%	1.65%	0.50%	4,918s
SPHYNX-miniHPC-Cascade-Lake	26.08%	11.02%	2.45%	0.29%	13.71%	1.19%	1.13%	1,321s

ALYA did not execute on miniHPC-Broadwell and miniHPC-Cascade-Lake nodes (with and without linking with Auto4OMP). GROMACS and SPEC OMP 2012 352.nab do not scale with the given problem size to the higher core count of the miniHPC-Cascade-Lake node.
The color gradient in the cells highlights (column-wise) *the incurred performance variation of each scheduling algorithm or selection method across different applications and systems. Lower performance degradation, hence, lighter and fewer gradient shades, is better (e.g.,* `ExpertSel`*).*

The performance experiments are conducted on five parallel applications selected to cover five different scientific domains: ALYA [15] (computational mechanics, multiphysics), GROMACS [16] (molecular dynamics), Mandelbrot [17] (mathematics), SPEC OMP 2012 352.nab [18] (molecular modeling), and SPHYNX [19] (astrophysics). These applications comprise numerous OpenMP parallel loops (see Table 2). To decide which loops to modify, we used the `profiling` functionality in LB4OMP [1] to identify the most time-consuming and load imbalanced loops of the applications. GROMACS is an exception from this selection as we intentionally chose a strongly memory-bound and well load balanced loop to show the possible overhead of using the selection methods proposed here. Dynamic scheduling naturally raises locality issues for memory-bound loops similar to the selected loop in GROMACS. This poses an extra challenge to the selection methods proposed here. The applications were executed on three multicore systems: miniHPC-Broadwell, Piz Daint-Haswell, and miniHPC-Cascade-Lake. All applications and Auto4OMP were compiled with the Intel compiler version 19.0.1.144.

Each experiment was repeated 5 times. We collected the execution time of every modified loop (latest finishing thread), and the total execution of the applications. We denote with T the number of time-steps, with L the number of loops where we modified the `schedule` clause, and with T_ol the time spent by the application outside of loops.

4.1 Auto4OMP in OpenMP Applications

As comparison baseline, we use the *ground truth* (`Oracle`) understood as the calculated highest achievable performance by a perfect selection of the highest performing scheduling algorithm (with or without `expert chunk`) for each loop, time-step, application, and system. `Oracle` can only be determined posthoc by assessing the performance of the application, loops, and time-steps executing on each system individually with all scheduling algorithms.

Another selection method, **`ManualBest`** (Table 2), i.e., the manual selection of the highest performing scheduling algorithms for each loop for an application-system pair. `ManualBest` is only recognizable after all experiments are completed and does not consider different scheduling algorithms per time-step. `ManualBest` represents a user's best effort to improve an application's performance by examining the performance of all available scheduling algorithms. An exhaustive search over suitable chunk parameter values would be infeasible for every scheduling algorithm, loop, time-step, application, and system. For simplicity, `ManualBest` only uses the default chunk parameter.

Performance plots of the full set of results can be found online.[5] Table 3 presents an overview of the performance achieved in prior work, i.e., LLVM `schedule(auto)` (`GAC`), `ManualBest`, and `Static Steal` (to include a non self-scheduling method), versus the proposed selection algorithm methods in Auto4OMP and `SS` with `expert chunk` to show the advantage of `expert chunk` without automatic algorithm selection. In Table 3, the percentages (%) represent the performance degradation relative to `Oracle` (ideal selection). The color gradient highlights (column-wise) the performance variation across different applications and systems that each scheduling algorithm or selection method achieved. Columns with a wide range of red shades indicate that *the respective scheduling algorithm or selection method does not offer high performance across different applications and systems.*

`ExhaustiveSel` and `ExpertSel` incur the least performance variability across all applications and systems, causing in the worst case 1.99% and 2.42% of performance degradation compared to `Oracle`, both for SPEC OMP 2012 352.nab-Piz Daint-Haswell and miniHPC-Broadwell, respectively.

`SS` with the proposed `expert chunk` parameter and `ManualBest` incur moderate performance variability across all application-system pairs, causing worst performance degradation of 5.26% to GROMACS executing on Piz Daint-Haswell, and 5.43% to SPHYNX on miniHPC-Broadwell, respectively. `GAC` and `RandomSel`, exhibit up to 12% *to* 15% performance degradation across different application-system pairs. `Static Steal`, which employs work stealing to balance the load across threads, shows high performance variability (up to 52%) across considered application-system pairs. However `Static Steal` provides high load balance via work stealing, it incurs poor performance for certain applications due to the high cost of load balancing (stealing) surpassing its performance gain.

In general, Table 3 shows that `ExpertSel` and `ExhaustiveSel` consistently achieved performance close to `Oracle` in most cases. In certain cases, e.g. with ALYA and SPHYNX, `ExpertSel` and `ExhaustiveSel` outperform `ManualBest`, which requires extensive offline experimentation. Overall, the

5. https://doi.org/10.5281/zenodo.6309015

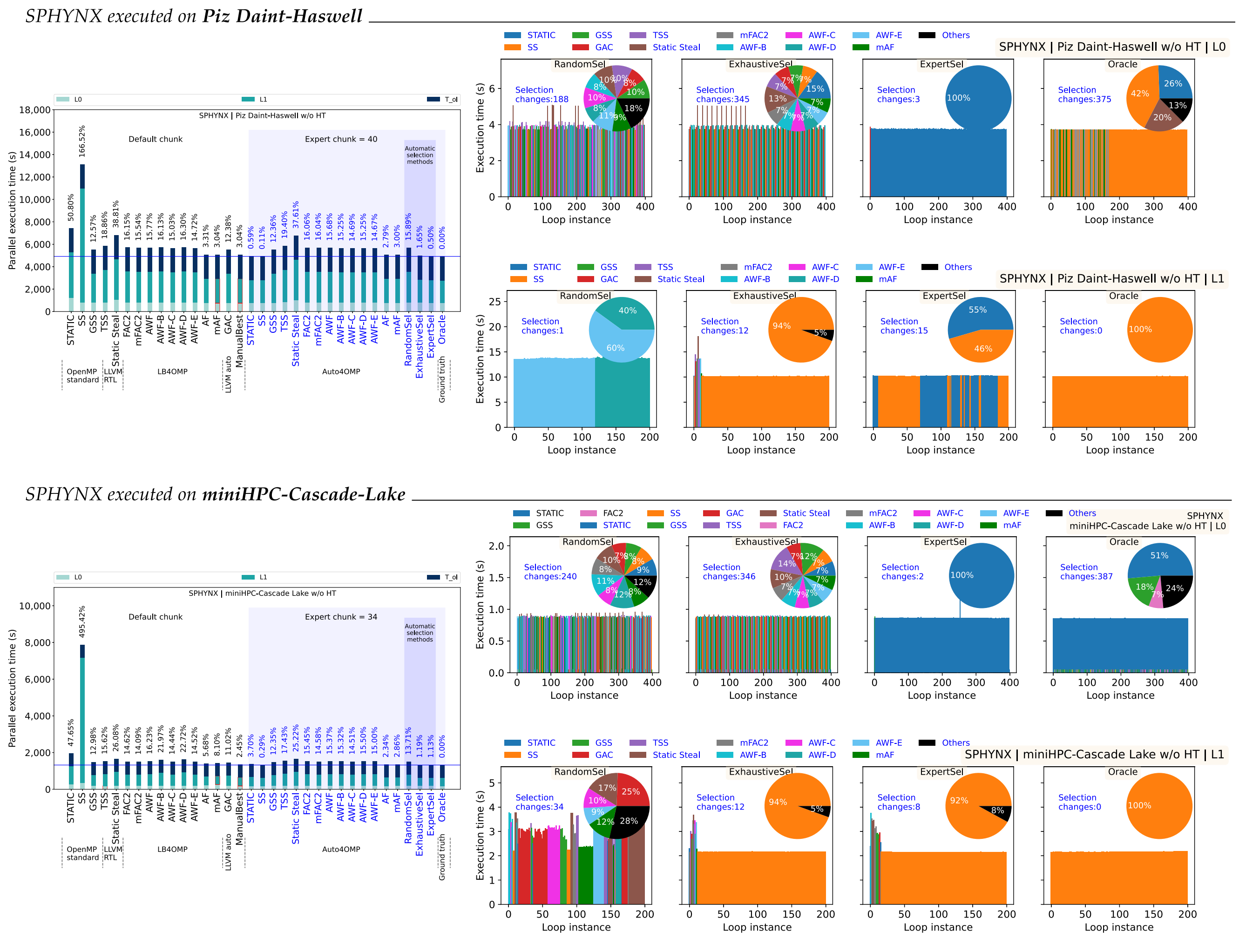

Fig. 7. *On the left*, median of parallel execution time (s) for each modified loop in SPHYNX executing on node Piz Daint-Haswell (top) and node miniHPC-Cascade-Lake (bottom). Percentages (%) denote performance degradation relative to `Oracle`. *On the right*, scheduling algorithms selected by the selection methods in Auto4OMP or by `Oracle`, per loop instance for SPHYNX's loops $L0$ and $L1$. The height of each bar represents the execution time (s) of the selected scheduling algorithm for the given loop instance shown on the x-axis. The pie charts show the percentage (%) among all selections that a scheduling algorithm was selected for a given loop. The average coefficient of variation (c.o.v.) among experiment repetitions for executions on Piz Daint-Haswell and miniHPC-Cascade-Lake nodes are 0.42% and 0.66%, respectively.

highest performance improvement of using Auto4OMP is achieved by `ExpertSel` with 0.5% performance degradation, compared to 12.38% incurred by `GAC` (LLVM's `schedule (auto)` implementation) for SPHYNX on Piz Daint-Haswell, which yields 11.88% performance gain by using `ExpertSel` . These observations confirm **H.1** for `ExpertSel` and `ExhaustiveSel` . However, `RandomSel` shows high performance variability across different application-system pairs due to suboptimal selections as discussed below (see Section 4.3).

Examining the performance of GROMACS in Table 3, one observes its high performance variability with `Static Steal` and `SS, expert chunk` across miniHPC-Broadwell and Piz Daint-Haswell nodes. Even though GROMACS performance with `SS, expert chunk` is among the best on miniHPC-Broadwell node, it is among the worst on Piz Daint-Haswell node. Automated algorithm selection methods do not exhibit such high performance variability across different nodes for the same application. Performance variability of GROMACS on miniHPC-Broadwell and Piz Daint-Haswell shows a case where the improvement of performance of the same application requires different scheduling algorithm across different systems as hypothesized in **H.2**.

We take a closer look at the results for SPHYNX executed on Piz Daint-Haswell and miniHPC-Cascade-Lake, to investigate hypotheses **H.3**, **H.4**, and **H.5**. SPHYNX shows high sensitivity to different scheduling algorithms and selection methods (see Table 3). Fig. 7 shows the results for SPHYNX executing on Piz Daint-Haswell (top) and miniHPC-Cascade-Lake (bottom).

The left side of Fig. 7 compares the performance of all scheduling algorithms without and with the proposed `expert chunk` and automatic algorithm selection methods proposed in Auto4OMP relative to `Oracle`. The variation of loop execution time across various scheduling techniques (from 8,000 to $\approx 1,750$ seconds) shows the significance and impact of selecting a suitable scheduling algorithm and a chunk parameter.

The right side of Fig. 7 shows loop execution time per execution instance (time-step).[6] The different colors identify

6. SPHYNX $L0$ executes twice per time-step, hence it appears in double the instances than $L1$.

the scheduling algorithm selected for the respective time-step by each selection method. ManualBest and GAC are not included since they do not change the selected algorithm across time-steps (e.g. left side of Fig. 7, node miniHPC-Cascade-Lake, shows that ManualBest considers TSS for $L0$ and mAF for $L1$ in all time-steps on miniHPC-Cascade-Lake). Selection changes in blue denotes the number of times a selection methods changes the scheduling algorithm all time steps. The pie charts present the frequency of which a given scheduling algorithm was selected. Legend on the top right with blue font color indicates that the scheduling algorithm uses the proposed expert chunk parameter.

From the results on the left side of Fig. 7, we observe that most scheduling algorithms achieved higher performance with the proposed expert chunk parameter than with the default chunk parameter, which *validates H.3*.

SS with the proposed expert chunk parameter achieved the highest performance on most time-steps of all loops, outperforming the highest performing option available in LB4OMP (mAF and AF), by 2.93% and 5.39%, on Piz Daint-Haswell and miniHPC-Cascade-Lake nodes, respectively. ExpertSel achieved the highest performance among the proposed automated selection methods, on both Piz Daint-Haswell and miniHPC-Cascade-Lake nodes, being only 0.50% and 1.13% worse than Oracle, respectively. ExhaustiveSel and ExpertSel show the highest adaptability/portability achieving high performance on both systems.

Inspecting Oracle for $L0$, one observes that the highest performing scheduling algorithm varies from one loop execution instance to another, as hypothesised with **H.4** and **H.5**. However, such variation of the highest performing scheduling algorithm is insignificant to the application execution time due to the short execution time of $L0$.

A change in the application, system, or any of their characteristics may render the Oracle selection invalid. The only realistic, adaptable, and portable approach to achieving performance close or equal to Oracle is through the automated selection of scheduling algorithms during execution.

4.2 Auto4OMP in MPI+OpenMP Applications

We investigate **H.6** by showing the impact of Auto4OMP on the performance of SPHYNX simulating an Evrard collapse over 200 time-steps using hybrid MPI+OpenMP parallelization. We use 4 nodes of miniHPC-Broadwell with 1 MPI rank and 20 OpenMP threads per node. The process-level scheduling is STATIC, i.e., each MPI rank executes 1/4-th of the loops iterations, 250,000. The thread-level scheduling is performed with Auto4OMP's three selection methods and the expert chunk. Fig. 8 shows SPHYNX's total execution time per time-step (including $L0$ and $L1$). Algorithm names in blue font color indicate the use of the expert chunk parameter (48 iterations in this case).

One can observe in Fig. 8 that SS,48, ExhaustiveSel and ExpertSel outperform STATIC by approximately $1.4s$, in every time-step. Cumulated over the entire application execution (200 time-steps), GAC, SS,expert chunk parameter, RandomSel , ExhaustiveSel , and ExpertSel , outperform STATIC by 17.39%, 22.08%, 21.17%, 21.65%, and 21.22% respectively.

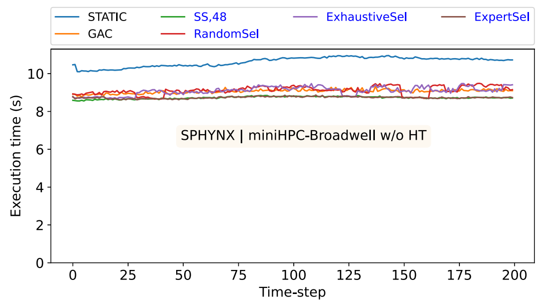

Fig. 8. SPHYNX execution time per time-step with 4 MPI ranks and 20 OpenMP threads/MPI rank on 4 miniHPC-Broadwell nodes.

This experiment shows that thread-level load balancing plays a significant role in improving the overall performance of hybrid MPI+OpenMP applications, which validates **H.6**. This observation is in agreement with existing work which showed that improvements in load imbalance at one level of parallelism (OpenMP thread in our case) propagate to reduce load imbalance at other levels (MPI processes in our case) [4], [5], [44].

4.3 Discussion

The increasing number of scheduling algorithms, options for chunk parameter, and the sensitivity of applications performance to such choices motivate the need for automated scheduling in OpenMP. The left side of Fig. 7 shows such sensitivity of the SPHYNX application to the choices of scheduling algorithm and chunk parameter.

Auto4OMP presents three scheduling algorithm selection methods, ranging from simple (RandomSel) to complex (ExpertSel), which requires expert knowledge for its design choices. Through the three automated scheduling selection methods, we explored the effects of various selection criteria on the choice of the scheduling algorithm and the achieved performance. For instance, RandomSel does not consider any characteristics about the application or scheduling algorithm during selection. This is reflected by its relatively poor choices and performance, in most cases. Another example, ExhaustiveSel, aims to minimize loop execution times and the results confirm the efficiency of such strategy in most cases.

ExpertSel employs fuzzy logic to encode existing scheduling expertise to select suitable scheduling algorithm. Distinct from other proposed automated selection methods, ExpertSel considers both the loop execution time and load imbalance for algorithm selection, together with their variation across execution instances. Many design choices for the fuzzy systems of ExpertSel depend on our expert knowledge and were also tuned during early experiments to reach a suitable performance. ExpertSel also made the fewest selection changes (in most cases) among the DLS algorithms during execution as it directly selects the most appropriate scheduling algorithm (compared to ExhaustiveSel) and its balanced expert rules. ExpertSel consistently selects high performance scheduling methods across all applications and systems considered herein, without further tuning to various applications or systems characteristics.

Overall, the automated selection methods adapt to various applications and systems at no cost from the user, whilst ManualBest and Oracle require exhaustive offline experimentation. Even if the `Oracle` combination of scheduling algorithm choices would be possible to know ahead of time, it would be impractical to pass the information to the OpenMP runtime library during applications' execution.

The execution of $L0$ of SPHYNX in Fig. 7 is of particular interest due to loop $L0$ executing twice per time-step, one very short instance of $\approx 20ms$ and another instance of $\approx 1s$. Automatically selecting the most suitable scheduling algorithm for such short yet similar loops is challenging, and `ExhaustiveSel` and `ExpertSel` outperformed other selection methods and single scheduling algorithms. Due to the particular selection criteria considered by the different automated selection methods, the scheduling algorithm selection and corresponding performance vary, and in certain cases greatly so.

The automatic choice of the `expert chunk` parameter improved applications performance with no additional cost. Certain scheduling algorithms dynamically (and some adaptively) calculate a variable chunk size for each scheduling round. Searching for the optimal chunk size per scheduling round as shown in other studies [45]) could be quite costly. Auto4OMP does not need such fine grain search for the optimal chunk size, since the selected scheduling algorithm calculates a suitable chunk size per scheduling round.

This work makes a first step towards automated load balancing in OpenMP, opening many directions and motivating the need for further improvement of the automated selection methods proposed herein.

5 Conclusion and Future Work

This work introduced Auto4OMP designed to address the scheduling algorithm selection problem in OpenMP applications. Auto4OMP employs a new `expert chunk` parameter and three new scheduling algorithm selection methods: `RandomSel` , `ExhaustiveSel` , and `ExpertSel` for OpenMP. Auto4OMP's performance was evaluated on five applications, executed on three multi-core architectures to test six research hypotheses. Each hypothesis as well as Auto4OMP's impact on applications' performance was analyzed and discussed.

The experiments show that Auto4OMP improved performance by up to 11% compared to LLVM's `schedule(auto) implementation` and outperformed manual selection. The benefits of an automated selection of scheduling algorithms go beyond OpenMP applications, as Auto4OMP also improved the performance of a hybrid MPI+OpenMP application, (SPHYNX running Evrard collapse) by up to 4% and 22% compared to using only `GAC` and `STATIC` as fixed OpenMP schedules, respectively. By leveraging OpenMP's `auto` schedule kind Auto4OMP is a step towards automated load balancing. Auto4OMP's improvements can be in terms of algorithm selection criteria and their relative priorities to one another.

One of Auto4OMP's limitation can be its applicability only to time-stepping applications (and not single-sweep problems). Another limitation is that the selection overhead might slightly degrade performance for purely memory-bound loops, where `STATIC` achieved the highest performance as seen with GROMACS.

The extension of the automated selection approach to process-level load balancing of MPI-only and MPI +OpenMP hybrid applications is part of author's ongoing work. Expanding the automated load balancing features of Auto4OMP to other OpenMP constructs and further experimentation with Auto4OMP on other applications and systems is seen as an important part of future work.

Acknowledgments

The authors acknowledge access to Piz Daint at the Swiss National Supercomputing Centre, Switzerland under the PASC SPH-EXA's share with the project ID c16.

References

[1] J. H. M. Korndörfer, A. Eleliemy, A. Mohammed, and F. M. Ciorba, "LB4OMP: A dynamic load balancing library for multithreaded applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 4, pp. 830–841, Apr. 2022.

[2] "Top500 list," Accessed: Jun. 14, 2022. [Online]. Available: https://top500.org/lists/top500/2022/06/

[3] K. Bergman *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Adv. Res. Projects Agency Informat. Process. Techn. Office*, vol. 15, 2008.

[4] D. Böhme, M. Geimer, L. Arnold, F. Voigtlaender, and F. Wolf, "Identifying the root causes of wait states in large-scale parallel applications," *ACM Trans. Parallel Comput.*, vol. 3, no. 2, 2016, Art. no. 11.

[5] A. Mohammed, A. Cavelan, F. M. Ciorba, R. M. Cabezón, and I. Banicescu, "Two-level dynamic load balancing for high performance scientific applications," in *Proc. SIAM Conf. Parallel Process. Sci. Comput.*, 2020, pp. 69–80.

[6] L. Dagum and R. Menon, "Openmp: An industry standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998.

[7] F. M. Ciorba, C. Iwainsky, and P. Buder, "OpenMP loop scheduling revisited: Making a case for more schedules," in *Proc. Int. Workshop OpenMP*, 2018, pp. 21–36.

[8] P. H. Penna *et al.*, "A comprehensive performance evaluation of the binLPT workload-aware loop scheduler," *Concurrency Computation: Pract. Experience*, vol. 31, no. 18, 2019, Art. no. e5170.

[9] F. Kasielke, R. Tscüter, M. Velten, F. M. Ciorba, C. Iwainsky, and I. Banicescu, "Exploring loop scheduling enhancements in OpenMP: An LLVM case study," in *Proc. 18th Int. Symp. Parallel Distrib. Comput.*, 2019, pp. 131–138.

[10] J. R. Rice, "The algorithm selection problem," in *Advances in Computers*, Amsterdam, Netherlands: Elsevier, 1976, pp. 65–118.

[11] T. Harrison, K. Waite, and P. White, "Analysis by paralysis: The pension purchase decision process," *Int. J. Bank Marketing*, vol. 24, pp. 5–23, 2006.

[12] O. A. R. Board, "OpenMP application programming interface standard v.3.0," Accessed: Apr. 7, 2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/spec30.pdf

[13] O. A. R. Board, "OpenMP application programming interface standard v.5.0," Accessed: Apr. 7, 2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[14] G. O. runtime library, Accessed: Apr. 7, 2021. [Online]. Available: https://gcc.gnu.org/onlinedocs/libgomp/

[15] M. Vazquez *et al.*, "Alya: Towards exascale for engineering simulation codes," 2014, *arXiv:1404.4881*.

[16] M. J. Abraham *et al.*, "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *Softw.X*, vol. 1-2, pp. 19–25, 2015.

[17] B. B. Mandelbrot, "Fractal aspects of the iteration of z $\rightarrow \Lambda$ z (1-z) for complex Λ and z," *J. Ann. New York Acad. Sci.*, vol. 357, no. 1, pp. 249–259, 1980.

[18] M. S. Müller *et al.*, "SPEC OMP2012â€"an application benchmark suite for parallel systems using OpenMP," in *Proc. Int. Workshop OpenMP*, 2012, pp. 223–236.

[19] R. M. Cabezón, D. Garcia-Senz, and J. Figueira, "SPHYNX: An accurate density-based SPH method for astrophysical applications," *Astron. Astrophys.*, vol. 606, 2017, Art. no. A78.

[20] C. P. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 10, pp. 1001–1016, Oct. 1985.

[21] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A method for scheduling parallel loops," *ACM J. Commun.*, vol. 35, pp. 90–101, 1992.

[22] S. Lucco, "A dynamic scheduling method for irregular parallel programs," in *Proc. ACM Conf. Program. Lang. Des. Implementation*, 1992, pp. 200–211.

[23] S. Flynn Hummel, J. Schmidt, R. N. Uma, and J. Wein, "Load-sharing in heterogeneous systems via weighted factoring," in *Proc. Annu. ACM Symp. Parallel Algorithms Architectures*, 1996, pp. 318–328.

[24] T. Hagerup, "Allocating independent tasks to parallel processors: An experimental study," *J. Parallel Distrib. Comput.*, vol. 47, pp. 185–197, 1997.

[25] I. Banicescu, V. Velusamy, and J. Devaprasad, "On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring," *J. Cluster Comput.*, vol. 6, pp. 215–226, 2003.

[26] I. Banicescu and Z. Liu, "Adaptive factoring: A dynamic scheduling method tuned to the rate of weight changes," in *Proc. High Perform. Comput. Symp.*, 2000, pp. 122–129.

[27] Y. Zhang, M. Voss, and E. Rogers, "Runtime empirical selection of loop schedulers on hyperthreaded SMPs," in *Proc. 19th Int. Parallel Distrib. Process. Symp.*, 2005, Art. no. 10.

[28] V. Sreenivasan, R. Javali, M. Hall, P. Balaprakash, T. R. Scogland, and B. R. de Supinski, "A framework for enabling openmp autotuning," in *International Workshop on OpenMP*, New York, NY, USA: Springer, 2019, pp. 50–60.

[29] A. Mohammed and F. M. Ciorba, "SiL: An approach for adjusting applications to heterogeneous systems under perturbations," in *Proc. Int. Workshop Algorithms Models Tools Parallel Comput. Heterogeneous Platforms, 24th Int. Eur. Conf. Parallel Distrib. Comput.*, 2018, pp. 456–468.

[30] A. Mohammed and F. M. Ciorba, "SimAS: A simulation-assisted approach for the algorithm selection problem of scheduling under perturbations," *Concurrency Computation: Pract. Experience*, vol. 32, 2020, Art. no. 5648.

[31] I. Banicescu, F. M. Ciorba, and S. Srivastava, *Scalable Computing: Theory and Practice*. Hoboken, NJ, USA: Wiley, 2013, pp. 437–466.

[32] A. Boulmier, I. Banicescu, F. M. Ciorba, and N. Abdennadher, "An autonomic approach for the selection of robust dynamic loop scheduling techniques," in *Proc. 16th Int. Symp. Parallel Distrib. Comput.*, 2017, pp. 9–17.

[33] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *J. Parallel Distrib. Comput.*, vol. 74, no. 10, pp. 2899–2917, 2014.

[34] N. Sukhija, B. Malone, S. Srivastava, I. Banicescu, and F. M. Ciorba, "Portfolio-based selection of robust dynamic loop scheduling algorithms using machine learning," in *Proc. 28th IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2014, pp. 1638–1647.

[35] P. Thoman, H. Jordan, S. Pellegrini, and T. Fahringer, "Automatic openmp loop scheduling: A combined compiler and runtime approach," in *International Workshop on OpenMP*, New York, NY, USA: Springer, 2012, pp. 88–101.

[36] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Proc. Int. Conf. Compiler Construction*, 2010, pp. 283–303.

[37] R. M. Cabezón, "SPHYNX website," Accessed: Oct. 20, 2019. [Online]. Available: https://astro.physik.unibas.ch/en/people/ruben-cabezon/sphynx/

[38] M. Livio, *The Golden Ratio: The Story of Phi, The World's Most Astonishing Number*. New York, NY, USA: Broadway Books, 2008.

[39] L. DeRose, B. Homer, and D. Johnson, "Detecting application load imbalance on high end massively parallel systems," in *Proc. Eur. Conf. Parallel Process.*, 2007, pp. 150–159.

[40] L. A. Zadeh, "Fuzzy Sets," in *Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems: Selected Papers by Lotfi A Zadeh*. Singapore: World Scientific, 1996, pp. 394–432.

[41] L. A. Zadeh, "Outline of a new approach to the analysis of complex systems and decision processes," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-3, no. 1, pp. 28–44, Jan. 1973.

[42] N. N. Karnik, J. M. Mendel, and Q. Liang, "Type-2 fuzzy logic systems," *IEEE Trans. Fuzzy Syst.*, vol. 7, no. 6, pp. 643–658, Dec. 1999.

[43] G. Klir and B. Yuan, *Fuzzy Sets and Fuzzy Logic*. Hoboken, NJ, USA: Prentice Hall, 1995.

[44] A. Eleliemy, A. Mohammed, and F. M. Ciorba, "Exploring the relation between two levels of scheduling using a novel simulation approach," in *Proc. 16th Int. Symp. Parallel Distrib. Comput.*, 2017, Art. no. 8.

[45] J. D. Booth and P. Lane, "An adaptive self-scheduling loop scheduler," 2020, *arXiv:2007.07977*.

Ali Mohammed received the doctoral degree from the University of Basel, in 2020. He is a research engineer with HPE's HPC/AI EMEA Research Lab (ERL), Switzerland. From March 2020 to April 2021, he was a postdoctoral researcher with the High-Performance Computing Group, University of Basel, Switzerland. His interests include data orchestration, scheduling, and performance simulation.

Jonas H. Müller Korndörfer is currently working toward the PhD degree with the Department of Mathematics and Computer Science, University of Basel, Switzerland. His main research interests include load balancing, scheduling, and mapping of computation, and communication intensive applications.

Ahmed Eleliemy received the doctoral degree in multilevel scheduling of computations on large-scale parallel systems from the University of Basel, in 2021. He is a postdoctoral researcher with the High Performance Computing Group, Department of Mathematics and Computer Science, University of Basel, Switzerland.

Florina M. Ciorba is currently an associate professor of High Performance Computing with the University of Basel, Switzerland. Her research interests include exploiting multilevel/hierarchical parallelism, dynamic and adaptive load balancing and scheduling, robustness, resilience, scalability, reproducibility, and benchmarking.