LB4OMP: A Dynamic Load Balancing Library for Multithreaded Applications

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

Abstract—Exascale computing systems will exhibit high degrees of hierarchical parallelism, with thousands of computing nodes and hundreds of cores per node. Efficiently exploiting hierarchical parallelism is challenging due to load imbalance that arises at multiple levels. OpenMP is the most widely-used standard for expressing and exploiting the ever-increasing node-level parallelism. The scheduling options in OpenMP are insufficient to address the load imbalance that arises during the execution of multithreaded applications. The limited scheduling options in OpenMP hinder research on novel scheduling techniques which require comparison with others from the literature. This work introduces LB4OMP, an open-source dynamic load balancing library that implements successful scheduling algorithms from the literature. LB4OMP is a research infrastructure designed to spur and support present and future scheduling research, for the benefit of multithreaded applications performance. Through an extensive performance analysis campaign, we assess the effectiveness and demystify the performance of all loop scheduling techniques in the library. We show that, for numerous applications-systems pairs, the scheduling techniques in LB4OMP outperform the scheduling options in OpenMP. Node-level load balancing using LB4OMP leads to reduced cross-node load imbalance and to improved MPI+OpenMP applications performance, which is critical for Exascale computing.

Index Terms—hierarchical parallelism, dynamic load balancing, self-scheduling, runtime library, OpenMP, multithreaded programming, shared-memory systems

✦

1 INTRODUCTION

ON the road to Exascale, we observe that modern and future high performance computing (HPC) systems combine an increasing number of computing nodes and, in particular, cores per node. For example, the top 5 systems on the Top500 list [1] contain thousands of nodes and tens to hundreds of cores per node.

Recent reports [2] indicate that the next update of the Sunway TaihuLight system will include 520 cores per node, double that of its predecessor. Such hardware parallelism increase leads to the challenge of exposing and expressing corresponding degrees of hierarchical parallelism in software to efficiently exploit the hierarchical hardware parallelism.

Load imbalance is a significant performance degradation factor in *computationally-intensive* applications [3] [4], defined as processors idling while there exist units of computation ready to be executed that no processor has started. This results in uneven execution progress among the parallel processing units, which can emerge from numerous application-, algorithm-, and/or systemic characteristics. Computationally-intensive applications often represent *irregular* workloads (e.g., due to boundary conditions, convergence, conditions, and branches). Computing

systems may consist of *heterogeneous* processors and may be affected by nonuniform memory access (NUMA) times, operating system noise, and contention due to sharing of resources. Load imbalance can be mitigated by an efficient dynamic scheduling of computation units onto processing units. Finding optimal schedules is NP-hard [5]. Therefore, many scheduling heuristics have been proposed over the years [6] [7].

Scheduling and load balancing that exploit multiple levels of hardware parallelism across and within computing nodes are critical challenges for the upcoming Exascale systems [8] [9]. Dynamic self-scheduling explicitly addresses application- and system-induced performance variations while minimizing load imbalance and scheduling overhead [10] [11] [12].

It has been recently shown that thread-level load imbalance has a significant impact on the performance of hybrid MPI+OpenMP applications [13]. OpenMP is the most widely-used standard for expressing and exploiting node-level parallelism. The OpenMP standard specifies three loop schedule kinds: `static`, `dynamic`, and `guided`. These scheduling options limit the highest achievable performance as they do not cover the broad spectrum of applications and systems characteristics [14] [15] [16]. Furthermore, the absence of a comparative implementation of the multitude of scheduling techniques from the literature hinders research on novel scheduling techniques which typically requires comparison with the scheduling state of the art.

This work builds on recent work on multilevel load balancing [13] by concentrating on thread-level scheduling and deepening the analysis of its performance impact for multithreaded applications executing on hierarchical parallel systems. Specifically, this work **provides** a broad range

of dynamic loop self-scheduling (DLS) techniques, implemented in a unified OpenMP runtime library (RTL), called LB4OMP [17] that can readily be used for MPI+OpenMP applications. Aiming for a **wide reach** and **broad impact**, we implemented LB4OMP as an extension of LLVM's OpenMP RTL given its widespread use (e.g., in the US DOE's Exascale Computing Project [18]), open-source nature, and high compatibility with widely-used compilers (Intel, IBM, PGI, GNU).

The LB4OMP library supports 14 carefully selected *dynamic* (and *adaptive*) loop self-scheduling techniques, ready to use in addition to those existing in the standard-compliant OpenMP libraries. Applications using LB4OMP benefit from improved performance due to the portfolio of DLS techniques, of which certain techniques *adapt* during execution to unpredictable variations in application and systemic characteristics (see Section 3.1). These 14 techniques are selected to cover a broad spectrum of *dynamic* (and *adaptive*) scheduling techniques. Specifically, LB4OMP provides:

- Five *dynamic* but **non-adaptive** self-scheduling techniques: fixed size chunking (FSC) [19], factoring (FAC) [20], the practical variant of factoring (FAC2), tapering (TAP) [21], and the practical variant of weighted factoring (WF2) [22];
- Seven *dynamic* and **adaptive** self-scheduling techniques: Bold (BOLD) [23], adaptive weighted factoring (AWF), its variants (AWF-B,C,D,E) [24], and adaptive factoring (AF) [10];
- Features for performance measurement to measure loop-specific performance metrics for an in-depth analysis of loop scheduling and load balancing.

LB4OMP also provides mFAC and mAF, two improved implementations that reduce the overhead of FAC and AF.

The scheduling techniques in LB4OMP differ in the amount of work assigned to a thread at a time, referred to as a chunk of loop iterations. Specifically techniques with: (1) *simple* chunk calculation include FSC, FAC2, and WF2; (2) *profiling-based* chunk calculation include FAC, mFAC, and TAP; and (3) *adaptive* (non-linear) chunk calculation include BOLD, AWF, its variants AWF-B,C,D,E, and AF and mAF. This work makes the following **contributions**:

1) *A novel systematic and unified implementation* of 14 *dynamic* (and *adaptive*) scheduling techniques.
2) *Advanced features for performance measurement* of loop performance and loop-level load imbalance.
3) *An in-depth analysis of the performance potential and limitations* of the standard and newly implemented scheduling techniques, which were so far only partially known to the non-experts and/or scheduling practitioners.

The **novelty** of this work lies in providing a standalone and unified implementation of efficient scheduling techniques from literature, which is needed to **spur new research in scheduling and load balancing for Exascale systems**. Prior to this work, scheduling research was hindered by the absence of an environment that supports a fair comparison with the existing scheduling algorithms in the literature. Novel scheduling techniques or improved versions of the techniques implemented in this work can now be implemented and compared in this unified testbed. LB4OMP

enables and promotes research on automatic selection methods to identify, during execution, the highest performing scheduling technique for a given application-loop-*time-step* configuration.

This work is **significant** by *bridging the gap* between the state-of-the-art and the state-of-the-practice of load balancing in multithreaded applications. This will allow the large degrees of heterogeneous node-level parallelism in today's pre- and upcoming Exascale systems to be efficiently exploited for improving applications performance.

This work is organized as follows. Section 2 is a review of the related literature highlighting the differences between prior and the present work. Section 3 describes the LB4OMP design and highlights the required extensions to LLVM's OpenMP RTL. The use of the newly implemented scheduling techniques in OpenMP applications via the LB4OMP library is detailed in Section 3. The experimental design and performance analysis campaign are presented and discussed in Section 4. The work is concluded in Section 5, which also outlines directions for future work.

2 RELATED WORK

The performance potential of a small number of *dynamic* and **non-adaptive** loop scheduling techniques (TSS, FAC2, WF2, and RAND), implemented in the GNU OpenMP RTL, was recently explored [15]. The authors showed cases when applications achieve improved performance beyond the one offered by the scheduling techniques supported in the GNU OpenMP RTL. Another variant of FAC, called BO FSS, was proposed and compared against STATIC, GSS, TSS, FAC2, TAP [21], HSS [25], and BinLPT [26], in another extended implementation in the GNU OpenMP RTL [27]. The scheduling techniques considered in these research efforts does not consider *dynamic and adaptive* scheduling techniques. In general, other efforts only considered extending the GNU OpenMP RTL, which is not compatible with other compilers, unlike the LLVM OpenMP RTL.

LLVM has gained considerable traction in the software vendor community and improving the open source LLVM compiler and runtime ecosystem is a priority for the US DOE Exascale Computing Project [18]. The LLVM OpenMP RTL was extended only by an implementation of FAC2 [16]. Experiments therein showed improved performance of certain workloads with the newly added DLS technique.

The present work improves over the previous related work by (1) providing *a novel systematic and unified implementation* of a broader range of *dynamic* (and *adaptive*) scheduling techniques. (2) providing *advanced features for performance measurement* of loop performance and loop-level load imbalance. (3) *demystifying the performance potential and limitations* of the standard and the newly implemented scheduling techniques through *an in-depth performance analysis campaign*.

Another direction of related work includes efforts that propose generic interfaces to allow users to implement their own loop scheduling techniques in different runtime libraries [28] [29] [30]. These efforts reduce the development challenges associated with the direct modification to the RTL source codes, i.e., developers can implement their scheduling technique via simplified, and ideally, well-documented interfaces. However, these efforts do not

exclude the need for extensive scheduling libraries to validate novel scheduling techniques and exploit the increased hardware parallelism of modern HPC systems. Therefore, such efforts [28], [29], [30] can be seen as potential methods that facilitate the development of another version of the LB4OMP scheduling library in the future.

Considering the vast amount of DLS techniques proposed in the literature, the following non-trivial question arises: *What are the criteria to include a particular scheduling technique into a unified scheduling library?* A number of research efforts attempted to answer this question [31] [20] [21] [23] [10] [24].

FSC [32], TSS [31], FAC [20], and TAP [21] were introduced by separate research groups. However, the performance of each of these techniques was compared against at least one of three main scheduling techniques STATIC, SS [33], and GSS [34] that nowadays correspond to schedule(static), schedule(dynamic,1), and schedule(guided,1) specified in the OpenMP standard. The chunk calculation in TSS, for instance, is simpler than the non-linear chunk calculation in GSS. Therefore, we state that the *simplicity of chunk calculation* is an important selection criterion.

FSC, TAP, and FAC are based on probabilistic analyses and use profiling information to calculate the chunk sizes that achieve the most balanced load execution for a given application with a high probability. The profiling information is obtained prior to the applications' execution. We state that the *profiling-based chunk calculation* is another important selection criterion.

Another set of related research efforts includes BOLD [23], AWF-B,C,D,E [24], and AF [10]. These techniques use profiling information obtained during applications' execution to adapt during execution the calculated chunk sizes to achieve the most balanced load execution for a given application. We state that the *adaptivity of chunk calculation* is another valuable selection criterion.

Based on the above criteria, FSC [32], FAC [20], TAP [21], WF [22], BOLD [23], AWF-B,C,D,E [24], and AF [10] were selected for implementation into the LB4OMP scheduling library. Other scheduling techniques that meet these criteria can also be considered for inclusion in LB4OMP. The DLS techniques selected in this work can also be applied to schedule OpenMP tasks and taskloops [35] [36]. The use of LB4OMP for scheduling tasks and taskloops is part of separate ongoing work by the authors.

3 THE LB4OMP LIBRARY

LB4OMP extends the LLVM OpenMP RTL version 8.0, which is widely used and compatible with various compilers, including Intel, IBM, GNU, and PGI. The choice of extending the LLVM OpenMP RTL meets important research goals and priorities of the HPC community [18].

Fig. 1 shows the LB4OMP loop scheduling mechanism which extends the scheduling mechanism in the LLVM OpenMP RTL. The three main functions responsible for the chunk calculation are implemented in the file kmp_dispatch.cpp. Upon initialization, each thread calls the __kmp_dispatch_init_algorithm function inside the kmp_dispatch.cpp file (init in

Fig. 1). This function then initializes the needed structures for the selected scheduling technique and calls __kmp_dispatch_next_algorithm (next in Fig. 1). The logic of the chunk calculation of all DLS techniques is implemented in the __kmp_dispatch_next_algorithm function. The __kmp_dispatch_next_algorithm is called each time a thread needs to obtain work. Since the threads obtain work from a shared queue, __kmp_dispatch_next_algorithm relies on different synchronization operations (sync in Fig. 1) depending on the scheduling technique in execution. Finally, the threads call __kmp_dispatch_finish (finish in Fig. 1) to reset variables or free allocated memory.
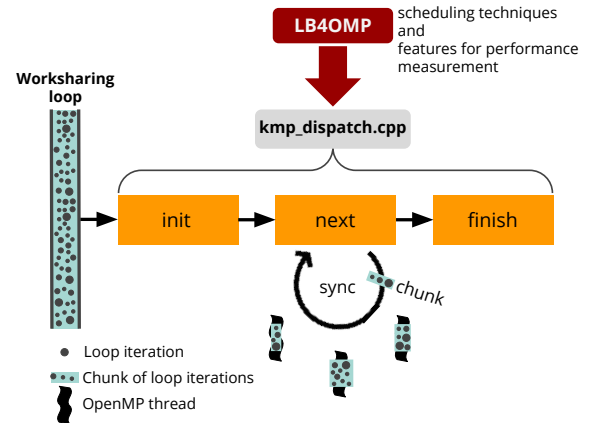

Fig. 1. Extension of the OpenMP LLVM RTL scheduling process for worksharing loops with LB4OMP.

Significance of the chunk parameter. The OpenMP standard scheduling techniques and the newly implemented scheduling techniques in LB4OMP support the declaration of a *chunk parameter* which bears different meanings among the scheduling techniques. For schedule(static, chunk) and schedule(dynamic,chunk), the chunk parameter denotes the amount of iterations that the threads should receive for every work request. For the other techniques, the chunk parameter works as a **threshold**, in the sense that when chunks sizes, calculated by a scheduling technique, fall below this threshold they will be replaced by a chunk sizes equal to the size of the chunk parameter. The chunk parameter was introduced by the OpenMP standard to minimize the scheduling overhead and to improve data locality. Declaring a proper *chunk parameter* improves performance since threads perform fewer scheduling rounds than without this threshold. This is confirmed by the experiments described in Section 4.

3.1 Dynamic Loop Scheduling Techniques

LB4OMP *bridges the gap* between the literature and the practice in dynamic load balancing of multithreaded applications. It represents an environment for a fair comparison of scheduling techniques and lays the ground for future research in loop scheduling of multithreaded applications.

The loop scheduling techniques implemented in LB4OMP are *dynamic* and a number of them are also *adaptive* self-scheduling techniques. With self-scheduling techniques,

free threads request, calculate, and obtain their own next chunk of units of work (loop iterations) by accessing a central shared work queue containing all iterations of a given loop. The chunk size is calculated according to the loop scheduling techniques in the OpenMP RTL.

Following is a brief description of each scheduling technique in LB4OMP, starting with *dynamic* and **non-adaptive** self-scheduling techniques followed by the *dynamic* and **adaptive** self-scheduling techniques. More details about the various chunk calculations for these techniques can be found in the literature [37].

Dynamic and non-adaptive self-scheduling.
SS (or dynamic,1 in OpenMP) [33] is a dynamic self-scheduling technique wherein the chunk size is always one loop iteration. SS incurs the highest scheduling overhead due to the largest number of chunks (equal to the number of loop iterations). SS can achieve a highly load-balanced execution in highly irregular execution environments.

FSC [19] determines an optimal chunk size that achieves a balanced execution of loop iterations with the smallest overhead. To calculate the optimal chunk size, FSC requires that the variability in iteration execution times and the scheduling overhead of assigning loop iterations are known before applications' execution.

GSS [34] is a trades-off between the load balancing achievable with SS and the low scheduling overhead incurred by STATIC. Unlike FSC, GSS assigns decreasing chunk sizes to balance the loop execution progress among all threads. For every work request, GSS assigns a chunk size equal to the number of remaining loop iterations divided by the total number of threads.

TAP [21] is based on a probabilistic analysis that represents a general case of GSS. It considers the average of loop iteration execution times μ and the standard deviation σ to achieve a higher load balance than GSS.

TSS [31] assigns decreasing chunk sizes similar to GSS. However, TSS uses a linear function to decrement chunk sizes. This linearity results in lower scheduling overhead in each scheduling step compared to GSS.

FAC [20] schedules the loop iterations in batches of equally-sized chunks. FAC evolved from comprehensive probabilistic analyses, and assumes prior knowledge about the average iteration execution times (μ) and their standard deviation (σ). A practical implementation of FAC, denoted FAC2, assigns half of the remaining loop iterations for every batch. The initial chunk size of FAC2 is half of the initial chunk size of GSS. If more time-consuming loop iterations are at the beginning of the loop, FAC2 is expected to better balance their execution than GSS.

WF [22] is similar to FAC, with the difference that each processing unit executes variably-sized chunks of a given batch according to its relative processing weights. The processing weights, W_{pj}, are determined prior to applications' execution and remain constant during execution. WF2 is the practical implementation of WF that is based on FAC2.

mFAC is our improved implementation of FAC. In the original FAC algorithm, the first thread that starts a new batch of iterations locks a mutex and computes the chunk size for the current batch. The subsequent threads simply read and reuse the already computed chunk size until the

iterations in the batch have been scheduled. This requires mutex-based synchronization. mFAC avoids such costly synchronization by involving more computation. Specifically, in mFAC, a shared counter is *atomically* incremented so that threads identify the current batch number. Hence, each thread calculates its *own* next chunk size depending on the batch counter. Depending on the synchronization and computation overheads in a given computing systems, one may use either FAC or mFAC.

Fig. 2 depicts an example of the chunk sizes calculated by the *dynamic* and *non-adaptive* scheduling techniques and their progression over the work requests for scheduling the iterations of the main loop (**L1**) of SPHYNX [38] (more details in Section 4.1). STATIC and SS are not shown in Fig. 2 as their chunk size progression is constant (straight line), at the size of the chosen chunk parameter. The chunk size progression for the *dynamic non-adaptive* scheduling techniques follows a decreasing chunk size pattern, wherein the next chunk of iterations is equal to or smaller than the previous. One can note that not only the chunk sizes but also the total the number of chunks allocated varies among the scheduling techniques. A small number of large chunk sizes may not mitigate load imbalance but incur a smaller scheduling overhead due to fewer scheduling operations while a greater number of smaller chunks may improve load balancing at the cost of increased scheduling overhead.
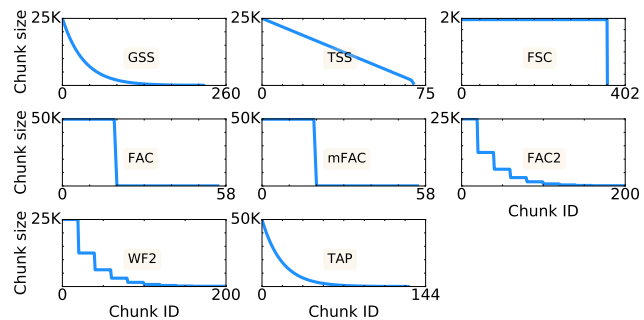

Fig. 2. Progression of chunk sizes for *dynamic* and **non-adaptive** techniques for scheduling the main loop (**L1**) of SPHYNX with $1,000,000$ loop iterations on a 20-thread miniHPC-Broadwell node and a chunk parameter of 97 loop iterations (Section 4.1). The chunk IDs are shown on the x axis and the number of iterations per chunk on the y axis.

Dynamic and adaptive self-scheduling.
Adaptive scheduling techniques regularly measure execution performance **during the application execution** and the scheduling decisions are taken based on this information. The adaptive scheduling techniques incur a higher scheduling overhead compared to non-adaptive techniques but are designed to outperform the non-adaptive ones in highly irregular execution environments.

BOLD [23] is a 'bolder' version of FAC and a further development of TAP. As with TAP, it uses the mean μ and the standard deviation σ of the iteration execution times as well as an estimate of the scheduling overhead. The driving idea behind the BOLD strategy was to increase early chunk sizes such that scheduling overhead is reduced while considering the risk of potentially too large chunks of iterations.

AWF [24] is similar to WF in that each thread executes variably-sized chunks of a given batch according to its rel-

ative processing weight. The processing weight is updated during execution based on the performance of each thread. `AWF` is devised for time-stepping applications and threads processing weights are updated at the end of each time-step. Variants of `AWF`, namely `AWF-B` and `AWF-C`, relaxed this constraint by updating processing weights at the end of every batch and chunk execution, respectively. Additional variants of `AWF`, namely `AWF-E` and `AWF-D`, are similar to `AWF-B` and `AWF-C`, respectively. In addition, `AWF-E` and `AWF-D` take into account the overhead of scheduling in calculating the relative processing weights.

`AF` [10] is an adaptive DLS technique derived from `FAC`. In contrast to `FAC`, `AF` *learns* both μ and σ for each computing resource during application execution to ensure *full adaptability* to all factors that cause load imbalance. `AF` adapts the chunk size during application execution based on the continuous updates of the mean loop iteration execution times μ and their standard deviation σ.

`mAF` is our improved implementation of `AF`. In the original `AF` algorithm, the execution time of earlier loop iterations (from the same execution) are collected to calculate the next chunk size. The collected times only consider the execution time of the loop iterations themselves. In LB4OMP, `mAF` also considers the scheduling overhead. Hence, `mAF` employs a more precise *performance estimation* for calculating the chunk size, which is expected to lead to higher load balance and performance.

Fig. 3 shows the chunk sizes calculated by the *dynamic* and *adaptive* scheduling techniques and their progression over the work requests for scheduling the iterations of the main loop (**L1**) of SPHYNX (more details in Section 4.1). The chunk sizes calculated by the adaptive scheduling techniques do not strictly decrease with each work request, but increase or decrease depending on the requesting thread's performance during execution. **This is the quintessence of adaptive self-scheduling and load balancing: threads which require more time to compute receive less work while threads which compute faster receive more work.**
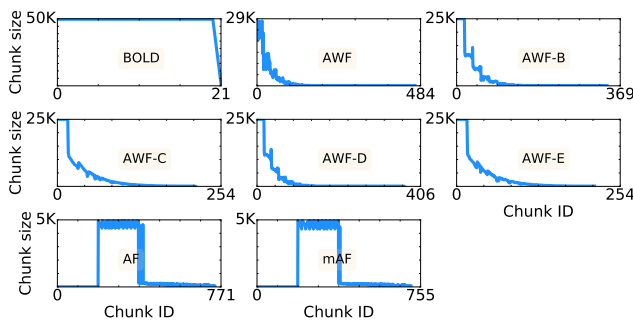

Fig. 3. Progression of chunk sizes for *dynamic* and ***adaptive*** techniques for self-scheduling the main loop (**L1**) of SPHYNX with $1,000,000$ loop iterations on a 20-thread miniHPC-Broadwell node and a `chunk` parameter of 97 loop iterations (Section 4.1). The chunk IDs are shown on the x axis and number of loop iterations per chunk on the y axis.

Dynamic and ***adaptive*** self-scheduling and load balancing techniques will be critical for achieving performance on upcoming Exascale systems with heterogeneous architectures in which scheduling needs to dynamically adapt to threads executing on slower or faster processing units. The observation from Fig. 2 also holds true for Fig. 3 regarding

the trade-off between fewer and larger chunks *and* more and smaller chunks.

3.2 Features for Performance Measurement

LB4OMP provides a number of features for performance measurements for target loops associated with the OpenMP `schedule` clause. These features are crucial for the analysis of loop scheduling and load balancing.

Thread execution time. This feature reports the threads execution times per loop execution instance. This information is important for the estimation of load imbalance in a parallel loop. This feature is enabled by defining the environment variable `KMP_TIME_LOOPS` and declaring the path where the measured performance data will be stored.

The *thread execution time* feature enables the calculation of well-known load imbalance metrics such as *coefficient of variation* (*c.o.v.*) [20] and *percent imbalance* (*p.i.*) [39]. The *c.o.v.* and *p.i.* equations are defined in Table 1, where T_{par}^{loop} denotes the parallel execution time of the loop. In this work, these metrics are calculated based on the data measured with LB4OMP for *individual* OpenMP loops and presented later in Section 4.

Chunk information. LB4OMP collects and stores the calculated chunk sizes for each thread in each scheduling round. This functionality can be enabled by setting the environment variable `KMP_PRINT_CHUNKS` to 1. The collected information is stored at the location defined in `KMP_TIME_LOOPS` (see above).

A detailed analysis of the chunk sizes calculated by each scheduling technique for given OpenMP loops is fundamental for understanding their performance. We used this feature for the in-depth performance analysis of the impact of the chunk parameter on all scheduling techniques described in Sections 4.3 and 4.4.

Statistical information about loop iterations execution times. LB4OMP provides a *profiling* feature that collects the mean of loop iterations execution times (μ) and their standard deviation σ. These measurements are required by the `FSC`, `FAC`, `TAP`, and `BOLD` scheduling techniques. The *profiling* feature relieves the (non-expert) user from the burden of collecting such profiling information.

This feature can be enabled by defining `schedule(runtime)` in the target loop, exporting `OMP_SCHEDULE=profiling`, and setting the environment variable `KMP_PROFILE_DATA` to the path where the profiling data will be stored.

3.3 Load Balancing Applications with LB4OMP

The use of LB4OMP with an OpenMP application is straightforward and illustrated in Fig. 4. First, one must ensure that the target OpenMP loops in the application contain the `schedule(runtime)` clause. If that is the case, no other changes are required and there is no need to recompile the code. Otherwise, one needs to change (or add, if the loop structure permits) the existing scheduling clause to `runtime` in all target loops and recompile the application. Next, one needs to add the path to the compiled LB4OMP to the environment variable that the linker uses to load dynamic and shared libraries (e.g., `LD_LIBRARY_PATH` on Linux/Unix systems). The workflow in Fig. 4 is almost

independent of the target system. The only system-related parameter required by LB4OMP is the host CPU clock frequency. This is passed to LB4OMP via the environment variable `KMP_CPU_SPEED` as an integer variable in MHz. The adaptive scheduling techniques in LB4OMP use low overhead cycle counters as `RDTSCP` to measure the execution time of previous chunks of iterations. We use the clock frequency to convert the cycles into time (which are the values expected by the formulas of those techniques). One may argue that the static value defined by the user in `KMP_CPU_SPEED` may be inaccurate for a number of modern processors that allow dynamic clock frequency change during execution. The measured performance of the threads is relative to each other, so variations in clock frequency during execution do not affect the chunk calculation. Work is ongoing to automate the process of collecting the clock speed for use in LB4OMP.
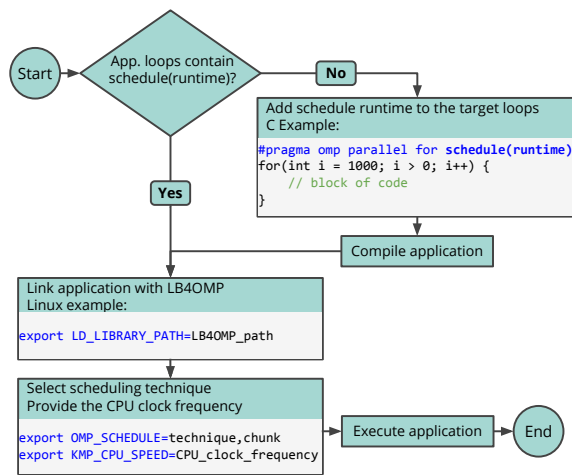

Fig. 4. Workflow for dynamic load balancing the execution of OpenMP applications using the scheduling techniques in LB4OMP.

Applications with multiple loops may need to employ different scheduling techniques in different parts of the code. LB4OMP uses the `schedule(runtime)` option available in OpenMP and, therefore, the scheduling technique selected by the user is read from the environment variable `OMP_SCHEDULE`. To select different scheduling techniques for an application in different parts of the code, one can use the function specified in the OpenMP standard `omp_set_schedule(omp_sched_t kind, int chunk_size)` [40]. This is used to update the scheduling technique specified with `OMP_SCHEDULE` during execution. One can also export environment variables directly from the application code to update the configuration of LB4OMP and, if preferred, to update the scheduling technique itself.

4 PERFORMANCE RESULTS AND DISCUSSION

We use three applications, two microbenchmarks, and three computing node types to evaluate the performance of the existing in LLVM OpenMP RTL and newly implemented DLS techniques in LB4OMP (see Table 1).

The applications SPEC OMP 2012 352.nab [41], SPHYNX, and the microbenchmark DIST [42] were selected

since they contain imbalanced and computationally-intensive loops, which are the most promising optimization targets for improved performance with dynamic (and adaptive) self-scheduling techniques [15]. The application GROMACS [43] and the microbenchmark STREAM [44] were selected to provide an overview of the scheduling overhead, ccNUMA effects, and locality loss incurred by the scheduling techniques in LB4OMP. These loops are balanced, of low arithmetic intensity, and mainly perform memory operations which stresses the possibly negative effects of dynamic (and adaptive) self-scheduling.

We will use the following notation to specify details regarding the applications and systems. T denotes number of time-steps, L the IDs of loop with modified `schedule` clauses, and T_ol the time spent by the application outside of loops. The loops for which we modify the `schedule` clause were parallel and not nested[1].

4.1 Design of Factorial Experiments

Table 1 presents the design of the factorial experiments needed for the extensive performance analysis campaign.

For SPEC OMP 2012 352.nab, part of the SPEC OMP 2012 benchmark suite, we used the *reference* input size. For SPHYNX, the Evrard Collapse test case was performed with $1,000,000$ particles. For GROMACS, the input size used in the experiments was the *Test Case B* taken from the Unified European Application Benchmark Suite (UEABS) [45].

The STREAM microbenchmark was executed with its default array size of $80,000,000$ elements (doubles), memory per array 610.4 MB, which required a total of 1831.1 MB in memory. DIST is synthetic microbenchmark used to show how the scheduling techniques react to different statistical loop workload distributions across iterations. Each loop of DIST follows a different workload distribution as indicated in Table 1.

Each experiment was repeated 5 times (STREAM was repeated 20 times) and the average execution time or memory bandwidth (in MB/s) for STREAM is reported. The applications SPEC OMP 2012 352.nab, SPHYNX, and GROMACS are time-stepping simulations. The *computationally-intensive loops* with modified `schedule` clause from each application and microbenchmark are indicated in Table 1. The applications and the LB4OMP library were compiled with the Intel compiler version 19.0.1.144. The characteristics of the computing systems are also indicated in Table 1.

Throughout the performance analysis campaign, all scheduling techniques used the *default chunk parameter* (1 loop iteration). We used the *thread execution time* LB4OMP feature (Section 3.2) and measured the loop execution and threads finishing times per loop to derive load imbalance.

4.2 Performance Analysis

The goal of this performance analysis campaign is to examine which DLS technique provides the *highest* performance and *lowest* load imbalance for each application's loop scheduled with LB4OMP.

Fig. 5 shows the average parallel execution time for each modified loop in SPEC OMP 2012 352.nab, SPHYNX, and

---

1. LB4OMP can schedule nested and non-nested parallel loops with independent iterations.

TABLE 1
Criteria used for the design of factorial experiments, resulting in a total of $4,826$ experiments.

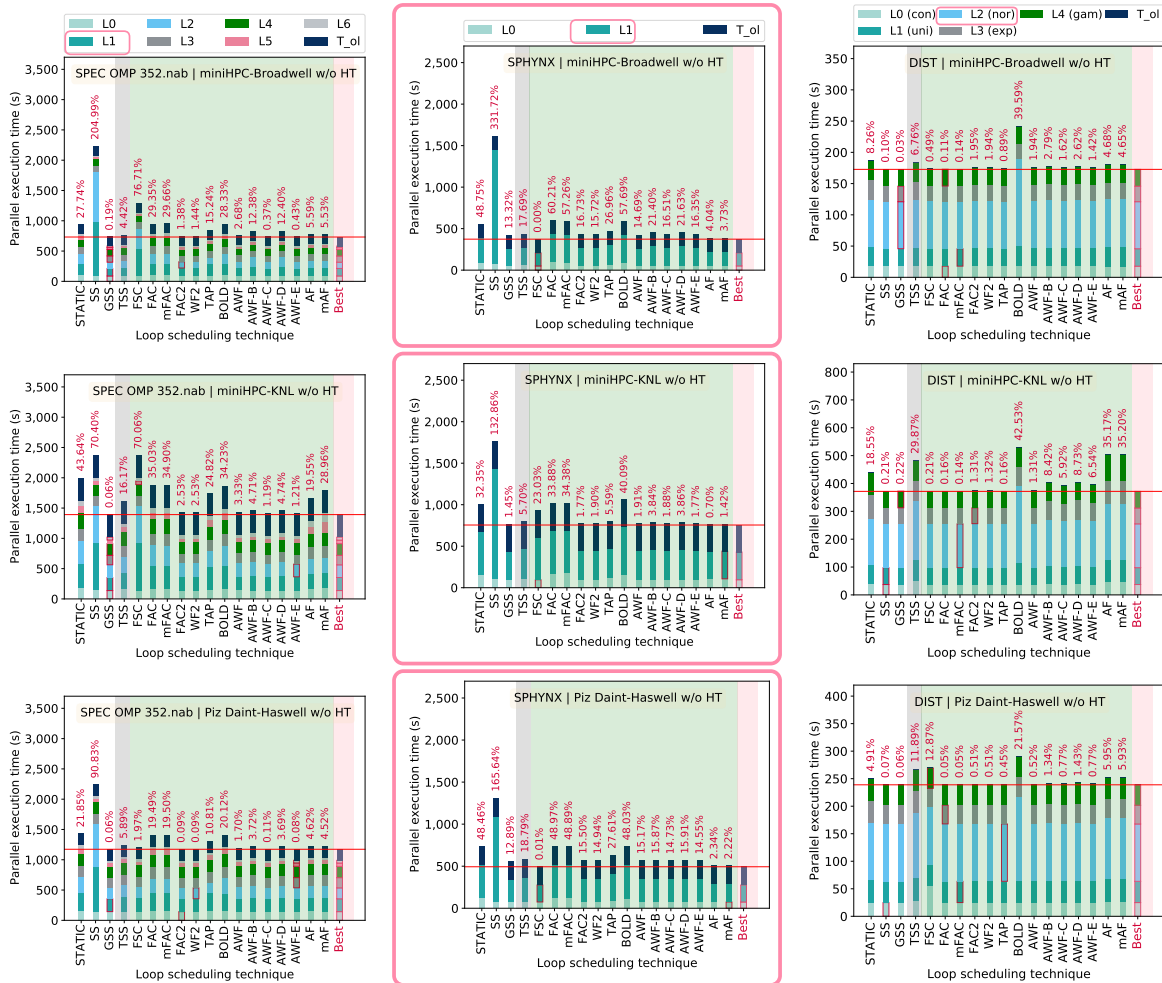| Factors		Values	Properties
Applications		SPEC OMP 2012 352.nab	$N = 44,794$ | $T = 1,002$ | Total loops = 13 | Modified loops = 7
		SPHYNX Evrard Collapse	$N = 1,000,000$ | $T = 20$ | Total loops = 37 | Modified loops = 2
		GROMACS	$N = 3,316,463$ | $T = 10,000$ | Total loops = 90 | Modified loops = 1
Microbenchmarks		STREAM	$N = 80,000,000$ | $T = 1$ | Total loops = 4 | Modified loops = 4 copy: $a(i) = b(i)$, bytes/iteration = 16, FLOP/iteration = 0 scale: $a(i) = q * b(i)$, bytes/iteration = 16, FLOP/iteration = 1 add: $a(i) = b(i) + c(i)$, bytes/iteration = 24, FLOP/iteration = 1 triad: $a(i) = b(i) + q * c(i)$, bytes/iteration = 24, FLOP/iteration = 2
		DIST	$N = 1,000$ | $T = 1$ | Total loops = 5 | Modified loops = 5 L0 (constant): 2.3×10^8 FLOP per iteration, L1 (uniform): $[10^3, 7 \times 10^8]$ FLOP per iteration, L2 (normal): $\mu = 9.5 \times 10^8$ FLOP, $\sigma = 7 \times 10^7$ FLOP, $[6 \times 10^8, 1.3 \times 10^9]$ FLOP per iteration, L3 (exponential): $\lambda = 1/3 \times 10^8$ FLOP, $[948, 4.5 \times 10^9]$ FLOP per iteration, L4 (gamma): $k = 2$, $\theta = 10^8$ FLOP, $[4.1 \times 10^6, 2.7 \times 10^9]$ FLOP per iteration,
Scheduling techniques	OpenMP standard	`static` (STATIC)	Straightforward parallelization
		`guided` (GSS), `dynamic,1` (SS)	
	OpenMP non-standard	TSS	Dynamic and non-adaptive self-scheduling techniques
	LB4OMP	FSC, FAC, FAC2, TAP, WF2, mFAC	
		BOLD, AWF, AWF-B, AWF-C, AWF-D, AWF-E, AF, mAF	Dynamic and adaptive self-scheduling techniques
Chunk parameter		$N/(2P)$, $N/(4P)$, $N/(8P)$, $N/(16P)$, ..., 1 Fewest number of chunk parameter tested: 7 for SPEC OMP 2012 352.nab on node miniHPC-KNL with hyperthreading Largest number of chunk parameter tested: 16 for SPHYNX on node Piz Daint-Haswell without hyperthreading	For `dynamic` and `static` the chunk parameter denotes the fixed amount of iterations in every chunk For all others DLS techniques, the chunk parameter represents the smallest chunk size a thread can obtain with a given self-scheduling technique
Computing nodes		miniHPC-Broadwell	Intel Broadwell E5-2640 v4 (2 sockets, 10 cores each) $P = 20$ without hyperthreading, $P = 40$ with hyperthreading Pinning: OMP_PLACES=cores, OMP_PROC_BIND=close
		miniHPC-KNL	Intel Xeon Phi KNL 7210 (1 socket, 64 cores) $P = 64$ without hyperthreading, $P = 256$ with 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, $P = 24$ with hyperthreading Pinning: OMP_PLACES=cores, OMP_PROC_BIND=close
Metrics		Performance per loop	Parallel loop execution time T_{par}^{loop}
		Load imbalance per loop	$c.o.v. = \sigma/\mu$ $p.i. = \frac{T_{max}^{loop} - \mu}{T_{par}^{loop}} \times \frac{P}{P-1} \times 100\%$

Fig. 5. Average parallel execution time for each modified loop in SPEC OMP 2012 352.nab, SPHYNX, and DIST with the default chunk parameter executing on all node types without hyperthreading. The x axis shows the DLS techniques, while the y axis presents the parallel execution time of each modified loop (L). The most time consuming loop of each application is highlighted in the legend. The red rectangles encompassing the bars represent the *best* performing scheduling technique for a given loop. On the x axis, the Best presents the shortest achievable execution time by selecting the combination of all individually highest performing techniques per loop together. The background color highlights: in white, the OpenMP standard DLS techniques, in gray, the non-standard technique that was already implemented in the LLVM OpenMP RTL, in green LB4OMP, and in dark pink the Best combination of techniques. The percentages denote performance degradation due to executing the applications with a single DLS technique vs. using the Best combination. The plots highlighted in light pink will be further explored in the next sections.

DIST with the default chunk parameter executing on all node types without hyperthreading. In Fig. 5, the `Best` combination of scheduling techniques varies greatly between applications and systems, and outperforms every single technique in most cases. This *reinforces* the need for additional scheduling options in OpenMP [15] since the `Best` combination commonly includes the techniques implemented in LB4OMP.

From the results of the *dynamic* and *non-adaptive* scheduling techniques in Fig. 5, we observe that `FAC2` and `GSS` presented fairly high performance in almost all experiments, despite the performance for SPHYNX on node miniHPC-Broadwell and Piz Daint-Haswell. Despite the high performance achieved by `TSS`, `FAC`, `mFAC`, and `TAP` for DIST, these techniques achieved low performance for the majority of other applications and systems. With profiling information, `FSC` calculated a proper chunk size achieving high performance in almost all experiments, despite the performance for SPHYNX on node miniHPC-KNL, SPEC OMP 2012 352.nab on nodes miniHPC-Broadwell and miniHPC-KNL, and DIST on Piz Daint-Haswell. The highest achieved performance improvement with a *dynamic* and *non-adaptive* scheduling technique was on SPHYNX with `FSC` on miniHPC-Broadwell outperforming `GSS`, the best standard technique in this case, by 13.32%.

The *dynamic* and *adaptive* loop scheduling techniques naturally add overhead. However, they adapt to application and system variations and heterogeneity without requiring profiling information. In Fig. 5, we observe that, except for `BOLD` (and for `AF`, `mAF` for SPEC OMP 2012 352.nab and DIST on miniHPC-KNL), all adaptive scheduling techniques consistently achieved high performance. In numerous cases, the adaptive scheduling techniques are included in the `Best` combination. For example, `AF` and `mAF`, consistently presented high performance for all results with SPHYNX, in which `mAF` is included in the `Best` combination for both miniHPC-KNL and Piz Daint-Haswell nodes. The highest achieved performance improvement with a *dynamic* and *adaptive* scheduling technique was on SPHYNX with `mAF` on Piz Daint-Haswell outperforming `GSS`, the best standard technique in this case, by 10.67%.

The results for SPHYNX on node miniHPC-Broadwell show that `AF` and `mAF` reasonably outperformed `GSS` by approximately 9.28% and 9.59% respectively. SPHYNX executing on miniHPC-Broadwell node also shows that `FSC` calculated a proper chunk size for both loops, obtaining the highest overall performance, outperforming `GSS` by 13.32% and `mAF` by 3.73%. This behavior is consistent among the results on nodes of miniHPC-KNL and Piz Daint-Haswell (Fig. 5). In the following Section 4.3, we further investigate the performance of the scheduling techniques for the most time-consuming loop of SPHYNX, $L1$, while varying the chunk parameter.

Fig. 6 presents the *load imbalance metrics*, *c.o.v.* and *p.i.*, calculated for the most time-consuming loop of SPHYNX ($L1$) execution on miniHPC-Broadwell node. These results show that most scheduling techniques achieve nearly perfect load balancing. Although these applications are computationally-intensive, with few memory operations, we can observe that almost perfect load balancing does not directly translate to high performance due to the additional

scheduling overhead and loss of data locality. For instance, in Fig. 6, `AWF-B` achieved perfect load balancing while in Fig. 5 we can observe that the execution time of SPHYNX executing on miniHPC-Broadwell node with `AWF-B` was 21.40% slower than `Best`.
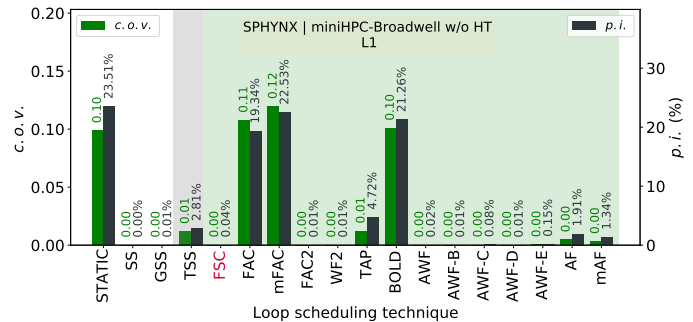

Fig. 6. Load imbalance metrics *c.o.v.* and *p.i.* for the most time consuming loop of SPHYNX ($L1$) highlighted in Fig. 5 and calculated based on the results obtained from miniHPC-Broadwell without hyperthreading.

Aspects such as scheduling overhead, ccNUMA effects, and data locality cannot directly be observed in Fig. 5, as the scheduling overhead is absorbed by improvement in the loop execution time due to dynamic and adaptive scheduling. Instead, we use a computationally-inexpensive loop of a widely used molecular dynamics application GROMACS [43], to reveal the overhead of all scheduling techniques. This particular loop in GROMACS has very low arithmetic intensity, regular loop iterations, and initializes three vector data structures which stresses ccNUMA effects and locality issues.

We also use the STREAM microbenchmark, a simple synthetic program that measures sustainable memory bandwidth, to show the memory bandwidth drop caused by ccNUMA effects and the locality issues that arise during dynamic and adaptive self-scheduling.

Fig. 7 shows the parallel loop execution time for the loop from GROMACS executing on node miniHPC-Broadwell while Fig. 8 shows the memory bandwidth (in MB/s) maintained by each kernel from STREAM also executing on node miniHPC-Broadwell. We use miniHPC-Broadwell since it is a two-socket node, making ccNUMA effects and data locality issues more prominent.

Three factors contribute to the scheduling overhead shown in Fig. 7: (1) Number of scheduling rounds o_{sr}; (2) Cost of calculating a chunk size o_{cs}; and (3) Synchronization cost between threads to obtain or to calculate a new chunk of loop iterations o_{sync}. Note that o_{cs} and o_{sync} are incurred with every scheduling round, therefore, growing with o_{sr}. **It is important to note that due to the non-deterministic and stochastic nature of dynamic and adaptive self-scheduling, high o_{sr} potentiates the loss of data locality and the importance of ccNUMA effects, which in this case are compounded with the overhead (compared to `STATIC`) seen in Fig. 7.**

`STATIC` in Fig. 7 shows the *smallest* scheduling overhead, where $o_{sr} = 1$, o_{cs} is a simple and deterministic division, and $o_{sync} = 0$, since threads need no synchronization.

`SS` shows the largest overhead due to $o_{sr} \leq N$, o_{cs} is negligible, and o_{sync} can either be negligible if $o_{sr} \ll N$
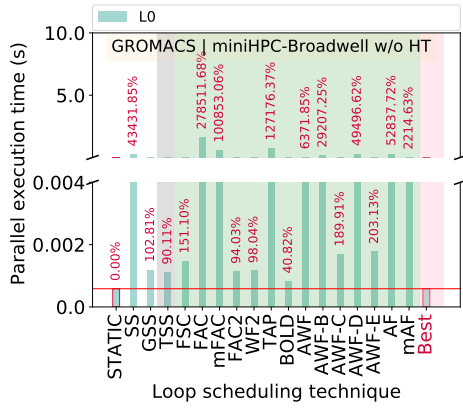
Fig. 7. Parallel loop execution time for GROMACS' loop $L0$. The performance of the dynamic and adaptive self-scheduling techniques is compared to the performance of $STATIC$, which offers highest performance and data locality at lowest scheduling overhead.

or significant if $o_{sr} = N$. In specific situations, some techniques may show higher overhead than SS (e.g. in this case, FAC, mFAC, TAP, AWF-D, and AF). If a scheduling technique with higher o_{cs} than SS calculates the same chunk size as SS it will cause more overhead than SS.

GSS, TSS, and FSC incur less overhead than SS due to assigning larger chunks of loop iterations, which reduces $o_{sr} < N$, increases data locality, while keeping o_{cs} and o_{sync} comparable by using simple chunk calculation functions and atomic operations to synchronize the threads.

The extremely large overhead with FAC ($> 43,000\%$) is due to the combination of high o_{sr}, o_{cs}, o_{sync} overheads and loss of data locality. FAC uses a complex function to calculate the chunk size, requiring profiling information and a mutex to synchronize the threads. mFAC has lower o_{sync} than FAC by using atomic operations, leading to lower overall overhead than FAC. FAC2 and WF2 outperform FAC and mFAC in terms of o_{cs} by using a simple chunk calculation function, not requiring profiling information, and using atomic operations for synchronization.

Similar to FAC and mFAC, TAP failed to calculate an appropriate chunk size based on the profiling information due to the very small loop iteration granularity, which resulted in loss of data locality, high o_{sr}, and o_{cs}.

BOLD generates chunk sizes very similar to STATIC but at a very high chunk calculation cost, therefore, incurring high o_{cs}, and low o_{sr} and o_{sync}.

AWF-B and AWF-D do not manage to adapt to the very fine iteration granularity of this GROMACS' loop and assign very small chunks, reducing data locality and increasing o_{sr}, and the cost of adaptation o_{cs}. In contrast, AWF-C and AWF-E only incur high o_{cs}.

AF and mAF also have very high o_{cs}. However, mAF also considers o_{cs} for the chunk size calculation. Therefore, it increases its chunk size to reduce o_{sr}, offering improvement over AF.

In Fig. 7 and Fig. 8, one can note that SS causes high scheduling overhead due to high o_{sr} and loss of data locality, which justifies the low memory bandwidth shown in Fig. 8 for all STREAM kernels. The low memory bandwidth achieved by FAC and mFAC is justifiable since those tech-

niques not only cause high o_{cs} and o_{sync} overheads but also need to read profiling information collected on a separate execution of the application.
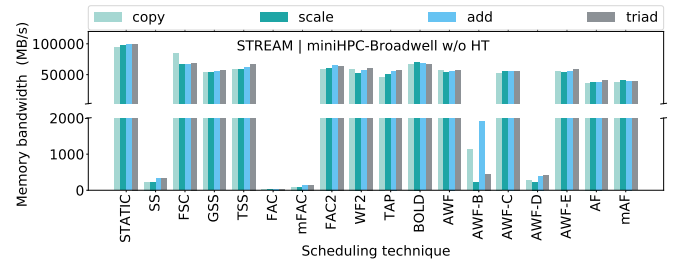

Fig. 8. Memory bandwidth (MB/s) for executing STREAM with all scheduling techniques executing on miniHPC-Broadwell.

For simple kernels, such as those from STREAM and $L0$ from GROMACS, profiling the execution of each loop iteration may adversely influence execution performance, and may lead FAC and mFAC to calculate very small chunk sizes, increasing o_{sr} and consequently also increasing scheduling overhead, ccNUMA effects, and loss of data locality. This observation is also valid for dynamic and adaptive techniques which measure (during execution) the execution time of previous chunks of iterations to determine the next chunk size. If the loop kernel's arithmetic intensity is low, dynamic and adaptive techniques may measure inaccurate values which may result in small chunk sizes, higher o_{sr}, non-negligible ccNUMA effects and loss of data locality.

4.3 Impact of Chunk Parameter Choice

We explore the performance impact of the chunk parameter for all DLS techniques. We experimented with many different values for the chunk parameter per DLS technique, loop, application, and node type/configuration as indicated in Table 1. Fig. 9 presents an overview of the results for SPHYNX comparing the Best combination of DLS techniques for loops $L0$ and $L1$ with the *default* value of the chunk parameter vs. the most performing combination of DLS techniques with the *best* value of the chunk parameter. The *best* value of the chunk parameter is identified by testing the application performance with values of the chunk parameter from $N/2P$ down to 1 (see Table 1).
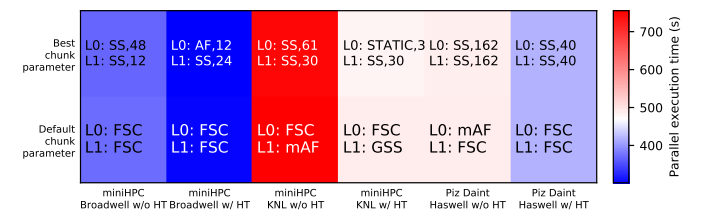

Fig. 9. Comparison of the parallel execution time of loops $L0$ and $L1$ for the highest performing combination of DLS techniques with the default chunk parameter (Best in Fig. 5) vs. with the *best* chunk parameter. Shown is the parallel execution time (heat-bar) of SPHYNX on various node types/configurations (x-axis) scheduled with the Best technique and two chunk parameters: default (chunk) vs. *best* (chunk) (y-axis).

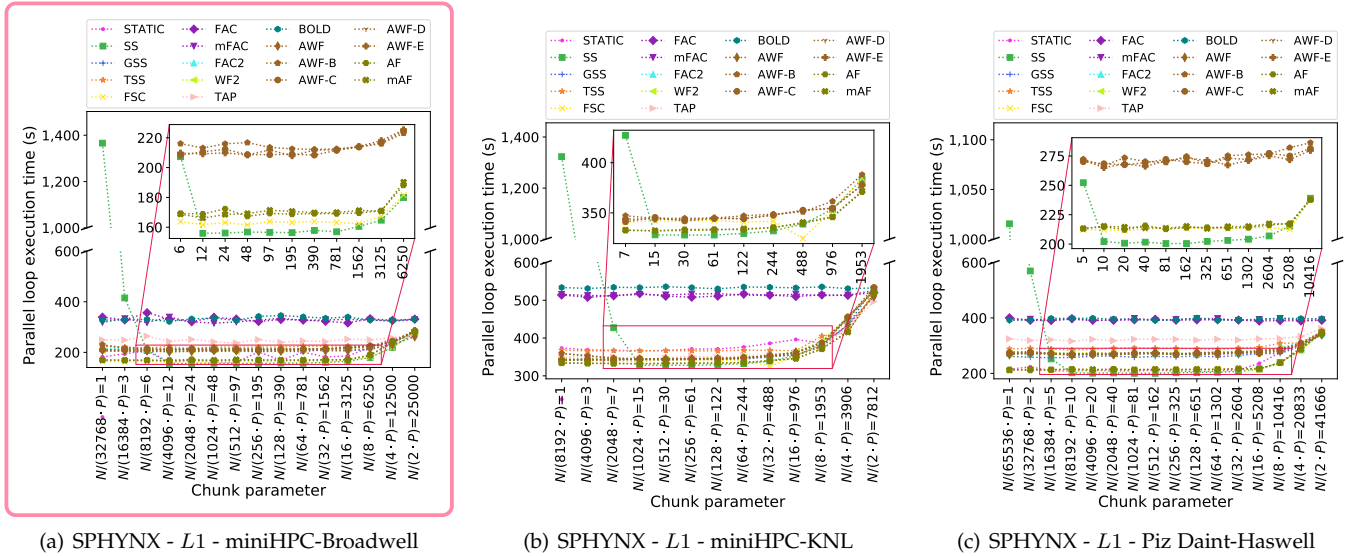In Fig. 9, the best chunk parameter always improved the performance of the applications. Similar colors indicate

Fig. 10. Parallel cumulative loop execution time for **SPHYNX**'s $L1$ loop executing on miniHPC-Broadwell, miniHPC-KNL, and Piz Daint-Haswell without hyperthreading. The red rectangles are zoomed in for the range of chunk parameter values that achieved high performance to show the performance of *dynamic* and *adaptive* loop scheduling techniques `AWF-B,C,D,E`, `AF`, and `mAF` and the *dynamic* and *non-adaptive* loop scheduling techniques `SS` and `FSC`. A proper chunk parameter value for `SS` reduces overall overhead and improves data locality, allowing `SS` to reach high performance or even to outperform all other techniques.

that the performance improvement was very low. A carefully selected chunk parameter for `SS` frequently achieves the highest performance. However, the process of finding such an optimal value requires extensive experimentation (e.g., such as the experiments presented here), and must be performed for each loop and system that the application will execute on. Furthermore, in the case of system variation, the optimal chunk parameter, once found, may no longer provide the highest performance since it would not be adapted during execution. It is impractical to rely exclusively on a manual and extensive experimentation process to find an optimal chunk parameter. This makes ***dynamically adaptive* loop scheduling techniques** a highly promising solution, especially on upcoming Exascale systems, which will increasingly be heterogeneous. The existing dynamic and adaptive scheduling techniques offer a first step for performance *auto-tuning* on a per loop basis against system and application variability.

Based on the results in Fig. 5 and Fig. 9, it is interesting to examine the impact of the chunk parameter on the performance of most time-consuming loop from SPHYNX, $L1$. These results are shown in Fig. 10, wherein the parallel execution time of $L1$ (y-axis) is shown for different chunk parameter values (x-axis). The chunk parameter values differ for each system since they are calculated using the available number of threads (see Table 1).

In Fig. 10 we expect to see `SS` reaching or outperforming `FSC`. The *dynamic* and *adaptive* loop scheduling techniques are expected to improve the performance by reducing overhead since with a chunk parameter they preserve improved data locality, and are executed fewer times.

The performance of `SS` indeed reaches and outperforms that of `FSC` with larger chunk parameters between 12 and $1,562$ for miniHPC-Broadwell, 15 and 122 for miniHPC-KNL, and 10 and $1,302$ for Piz Daint-Haswell. This is due to the improved data locality and scheduling overhead of `SS`

with a larger chunk parameter value. `FSC` is unaffected since it calculates a chunk size slightly larger than the range of chunk parameter values that achieve highest performance. The performance of `FSC` is only affected when the chosen chunk parameter value is larger than the chunk size calculated by the technique itself.

All results in Fig. 10 show that performance degrades with large chunk parameter values, approximately $2,000$ loop iterations. This happens since the $L1$ loop from SPHYNX is irregular (see Fig. 6, $L1$ of SPHYNX executed with `STATIC`) and, therefore, certain threads receive more work than others resulting in poor performance due to a load imbalanced execution. We expected that the *dynamic* and *adaptive* loop scheduling techniques show improved performance when the chunk parameter is chosen since their overhead would be reduced while preserving data locality. This was not the case. These results are discussed in Section 4.4, where the progression of the chunk sizes of each DLS technique is explored, highlighting why no improvement can be observed for the *dynamic* and *adaptive* loop scheduling techniques in this particular case.

4.4 Influence of Chunk Size Progression

The chunk size progression for the DLS techniques during the scheduling of the $L1$ loop from SPHYNX is shown in Fig. 11, with chunk ID on the x axis (denoting the number of chunks produced) and their sizes on the y axis. We used LB4OMP with `KMP_PRINT_CHUNKS=1` (*chunk information feature*, Section 3.2) to collect and report the chunk sizes assigned by each DLS technique in every scheduling round.

In general, fewer chunks imply improved data locality and a smaller scheduling overhead due to fewer scheduling operations. Fewer chunks may also result in potentially higher load imbalance since the chunk sizes are larger than with fewer chunks, as observed in Fig. 10, in Section 4.3. The results for the *dynamic* and *adaptive* loop scheduling

techniques `AWF-B,C,D,E`, `AF`, and `mAF` in Fig. 11, clarify why, in this case, no performance improvements can be observed when a chunk parameter is given. Even with a relatively large value of the chunk parameter, such as $3,125$, none of the *adaptive* loop scheduling techniques reaches the given value early enough to benefit from improved data locality and the lower overhead of executing fewer scheduling rounds. A much larger chunk parameter value would not necessarily improve loop performance due to the potential of load imbalance associated with large chunks.
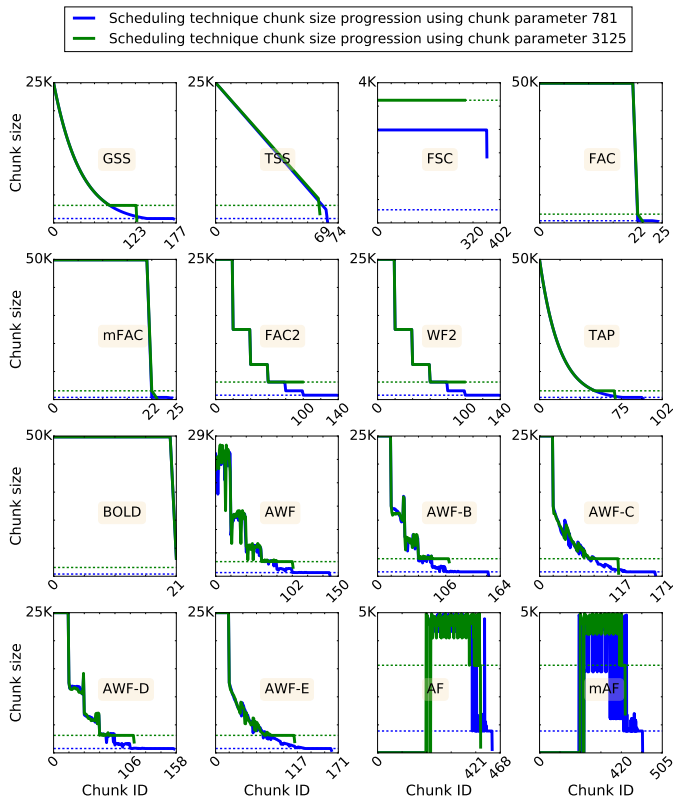
Fig. 11. Progression of chunk sizes with the DLS techniques in LB4OMP for scheduling **SPHYNX**'s $L1$ loop of $1,000,000$ loop iterations on a 20-thread miniHPC-Broadwell and chunk parameters $N/(64P) = 781$ iterations (blue dotted line) and $N/(16P) = 3,125$ (green dotted line) iterations, respectively. Chunk IDs are shown on the x axis and their sizes on the y axis. The blue and green curves represent the progression of the chunk sizes produced by a DLS technique instantiated either with 781 or $3,125$ as chunk parameter (lower threshold for calculating a chunk size).

Apart from `AF`, `mAF`, and `FSC`, all scheduling techniques follow a decreasing chunk size pattern. The *dynamic* and *adaptive* loop scheduling techniques `AWF-B,C,D,E` follow a decreasing chunk size pattern (similar to `GSS` and `FAC2`), with the major difference of adapting to system variation by increasing or decreasing their chunk sizes *during execution*.

Each DLS technique produces a different number of chunks (x-axis) depending on its chunk calculation method (non-adaptive methods) and loop performance during execution (adaptive methods). As described in Section 4.2, the number of chunks is proportional to the number of scheduling rounds, o_{sr}, and contributes to the scheduling overhead associated with a technique.

The first chunks calculated by `AF` and `mAF` are small as these DLS techniques perform a warm-up scheduling round

where they gather initial information about the loop iterations performance. These first chunks sizes are hard-coded to 10 loop iterations and are unaffected by the declaration of the chunk parameter.

5 CONCLUSIONS AND FUTURE WORK

We introduced LB4OMP, a novel open-source library for dynamic load balancing of multithreaded applications that use OpenMP, implemented as an extension of LLVM's OpenMP RTL. This work contributes: *a systematic and unified implementation* of 14 *dynamic* (and *adaptive*) loop scheduling techniques; *features for advanced performance measurement* of loop performance and load imbalance; and *an in-depth analysis of the performance potential and limitations* of the OpenMP standard and the newly implemented scheduling techniques. Through an extensive performance analysis campaign we showed that for numerous application-systems pairs, the scheduling techniques in LB4OMP outperform those from the OpenMP standard.

With this work, we *bridge the gap* between the state-of-the-art and the state-of-the-practice of load balancing in multithreaded applications. This will allow the efficient exploitation of large degrees of heterogeneous node-level parallelism for improving the performance of applications on upcoming Exascale systems.

LB4OMP represents the *first and necessary* step for devising automated methods to dynamically select the highest performing loop scheduling techniques during applications execution. Devising such methods is part of ongoing work by the authors.

A possible extension is to expand the selection criteria to include additional DLS techniques in LB4OMP. The study of locality-aware self-scheduling techniques is a promising research direction. We plan to patch and up-stream the DLS techniques implemented in LB4OMP to the main LLVM OpenMP RTL, facilitating a broad use and impact for OpenMP applications. Applying LB4OMP to *explicit OpenMP task scheduling* is also planned as future work.

REFERENCES

[1] "Top500 List," https://www.top500.org/lists/top500/2020/11/, Accessed: February 22, 2021.

[2] "A first Peek at China's Sunway Exascale Supercomputer," www.nextplatform.com/2021/02/10/a-sneak-peek-at-chinas-sunway-exascale-supercomputer/, Accessed: February 22, 2021.

[3] S. Flynn Hummel, I. Banicescu, C.-T. Wang, and J. Wein, "Load Balancing and Data Locality via Fractiling: An Experimental Study," in *Lang., Compi. RT. Sys. Scal. Compu.*, 1996, pp. 85–98.

[4] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig *et al.*, "The International Exascale Software Project Roadmap," *Intern. J. H. P. Comp. App.*, pp. 3–60, 2011.

[5] D. S. Johnson, "The NP-completeness Column: An Ongoing Guide," *J. of Alg.*, pp. 434–451, 1985.

[6] A. Khan, C. L. McCreary, and M. S. Jones, "A Comparison of Multiprocessor Scheduling Heuristics," in *P. Intern. C. on Par. Proc.*, vol. 2, 1994, pp. 243–250.

[7] H. Izakian, A. Abraham, and V. Snasel, "Comparison of Heuristics for Scheduling Independent tasks on Heterogeneous Distributed Environments," in *Intern. C. Compu. Sci. Opt.*, vol. 1, 2009, pp. 8–12.

[8] K. Bergman, S. Borkar, D. Campbell, W. Carlson, and et al., "Exascale Computing Study: Technology Challenges in Achieving Exascale Systems," *Def. Adv. Research Proj. Ag. Info. Proc. Tech. Office, Tech. Rep*, 2008.

[9] M. Asch, T. Moore, R. Badia, M. Beck, P. Beckman, T. Bidot, F. Bodin, F. Cappello, A. Choudhary, B. de Supinski *et al.*, "Big data and Extreme-scale Computing: Pathways to Convergence-toward a Shaping Strategy for a Future Software and Data Ecosystem for Scientific Inquiry," *Intern. J. H. P. Comp. App.*, vol. 32, no. 4, pp. 435–479, 2018.

[10] I. Banicescu and Z. Liu, "Adaptive Factoring: A Dynamic Scheduling Method Tuned to the Rate of Weight Changes," in *P. of th H. P. C. Symp.*, 2000, pp. 122–129.

[11] P. Thoman, H. Jordan, S. Pellegrini, and T. Fahringer, "Automatic OpenMP Loop Scheduling: A Combined Compiler and Runtime Approach," in *P. Intern. W. on OpenMP*, 2012, pp. 88–101.

[12] Y. Wang, W. Ji, F. Shi, Q. Zuo, and N. Deng, "Knowledge-Based Adaptive Self-Scheduling," in *P. Intern. C. on Net. and Par. Comp.*, 2012, pp. 22–32.

[13] A. Mohammed, A. Cavelan, F. M. Ciorba, R. M. Cabezón, and I. Banicescu, "Two-level Dynamic Load Balancing for High Performance Scientific Applications," in *P. SIAM C. on Par. Proc. Sci. Comp.*, 2020, pp. 69–80.

[14] E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, X. Martorell, and R. Silvera, "Is the Schedule Clause Really Necessary in OpenMP?" in *P. Intern. W. on OpenMP App. and Tools*, 2003, pp. 147–159.

[15] F. M. Ciorba, C. Iwainsky, and P. Buder, "OpenMP Loop Scheduling Revisited: Making a Case for More Schedules," in *P. Intern. W. on OpenMP*, 2018.

[16] F. Kasielke, R. Tschüter, C. Iwainsky, M. Velten, F. M. Ciorba, and I. Banicescu, "Exploring Loop Scheduling Enhancements in OpenMP: An LLVM Case Study," in *P. Intern. Symp. on Par. Dist. Comp.*, Amsterdam, 2019.

[17] J. H. Müller Korndörfer, A. Eleliemy, A. Mohammed, and F. M. Ciorba, "unibas-dmi-hpc/LB4OMP: LB4OMP v1.0," 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3872907

[18] M. A. Heroux, J. Carter, R. Thakur, L. McInnes, J. Ahrens, T. Munson, J. Robert Neely, and J. S. Vetter, "ECP Software Technology Capability Assessment Report," Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), Tech. Rep., 2020.

[19] C. P. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors," *J. Trans. on Soft. Eng.*, pp. 1001–1016, 1985.

[20] S. Flynn Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A Method for Scheduling Parallel Loops," *J. of Comm.*, pp. 90–101, 1992.

[21] S. Lucco, "A Dynamic Scheduling Method for Irregular Parallel Programs," in *P. C. on Progra. Lang. D. and Impl.*, 1992, pp. 200–211.

[22] S. Flynn Hummel, J. Schmidt, R. N. Uma, and J. Wein, "Load-sharing in Heterogeneous Systems via Weighted Factoring," in *P. Symp. on Par. Alg. Arch.*, 1996, pp. 318–328.

[23] T. Hagerup, "Allocating Independent Tasks to Parallel Processors: An Experimental Study," *J. Par. and Dist. Comp.*, pp. 185–197, 1997.

[24] I. Banicescu, V. Velusamy, and J. Devaprasad, "On the Scalability of Dynamic Scheduling Scientific Applications with Adaptive Weighted Factoring," *J. of Clus. Comp.*, pp. 215–226, 2003.

[25] A. Kejariwal, A. Nicolau, and C. D. Polychronopoulos, "History-aware self-scheduling," in *P. Intern. C. on Par. Proc.*, 2006, pp. 185–192.

[26] P. H. Penna, M. Castro, P. Plentz, H. C. Freitas, F. Broquedis, and J.-F. Méhaut, "BinLPT: A Novel Workload-aware Loop Scheduler for Irregular Parallel Loops," *Simp. em Sis. Comp. de Alto Desemp.*, 2017.

[27] K.-R. Kim, K. Youngjae, and S. Park, "A probabilistic machine learning approach to scheduling parallel loops with bayesian optimization," *J. Trans. on Par. and Dist. Sys.*, 2020.

[28] S. Bak, Y. Guo, P. Balaji, and V. Sarkar, "Optimized Execution of Parallel Loops via User-Defined Scheduling Policies," in *P. Intern. C. on Par. Proc.*, 2019.

[29] V. Kale, C. Iwainsky, M. Klemm, J. H. Korndörfer Müller, and F. M. Ciorba, "Towards A Standard Interface for User-Defined Scheduling in OpenMP," in *P. Intern. W. on OpenMP*, 2019.

[30] A. Santana, V. Freitas, M. Castro, L. Lima Pilla, and J.-F. Méhaut, "ARTful: A Specification for User-defined Schedulers Targeting Multiple HPC Runtime Systems," 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02454426

[31] T. H. Tzen and L. M. Ni, "Trapezoid Self-scheduling: A Practical Scheduling Scheme for Parallel Compilers," *J. Trans. on Par. Dist. Sys.*, pp. 87–98, 1993.

[32] M. D. Durand, T. Montaut, L. Kervella, and W. Jalby, "Impact of Memory Contention on Dynamic Scheduling on NUMA Multiprocessors," in *P. Intern. C. on Par. Proc.*, 1993, pp. 258–262.

[33] T. Peiyi and Y. Pen-Chung, "Processor Self-Scheduling for Multiple-Nested Parallel Loops," in *P. Intern. C. on Par. Proc.*, 1986, pp. 528–535.

[34] C. D. Polychronopoulos and D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *J. Trans. on Compu.*, pp. 1425–1439, 1987.

[35] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of OpenMP Task Scheduling Strategies," in *Intern. W. on OpenMP*, 2008, pp. 100–110.

[36] J. Clet-Ortega, P. Carribault, and M. Pérache, "Evaluation of OpenMP Task Scheduling Algorithms for Large NUMA Architectures," in *Eu. C. on Par. Proc.*, 2014, pp. 596–607.

[37] A. Eleliemy and F. M. Ciorba, "A Distributed Chunk Calculation Approach for Self-scheduling of Parallel Applications on Distributed-memory Systems," *J. of Computa. Sci.*, p. 101284, 2021.

[38] "SPHYNX Website," https://astro.physik.unibas.ch/people/ruben-cabezon/sphynx.html, accessed: October 20, 2020.

[39] L. DeRose, B. Homer, and D. Johnson, "Detecting Application Load Imbalance on High End Massively Parallel Systems," in *Eu. C. on Par. Proc.* Springer, 2007, pp. 150–159.

[40] "omp_set_schedule function: ," www.openmp.org/spec-html/5.0/openmpsu121.html, accessed: May 20, 2020.

[41] "SPEC OpenMP 2012 352.nab benchmark," https://www.spec.org/omp2012/Docs/352.nab.html, Accessed: October 16, 2020.

[42] "DIST Microbenchmark," https://drive.switch.ch/index.php/s/XIhieSdmoRuLcRR, Accessed: March 28, 2021.

[43] D. Van Der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark, and H. J. Berendsen, "GROMACS: fast, flexible, and free," *J. of compu. chemistry*, pp. 1701–1718, 2005.

[44] "STREAM Microbenchmark," http://www.cs.virginia.edu/stream/ref.html, Accessed: March 29, 2021.

[45] "Unified European Application Benchmark Suite (UEABS)," https://repository.prace-ri.eu/git/UEABS/ueabs, accessed: July, 10, 2020.

Jonas H. Müller Korndörfer is a PhD candidate at the Department of Mathematics and Computer Science at the University of Basel, Switzerland. His main research interests include load balancing, scheduling, and mapping of computation and communication intensive applications. Website: hpc.dmi.unibas.ch/en/people/jonas-h-mueller-korndoerfer.

Ahmed Eleliemy is a postdoctoral researcher at the High Performance Computing Group at the Department of Mathematics and Computer Science at the University of Basel, Switzerland. He received his doctoral degree in multilevel scheduling of computations on large-scale parallel systems from the University of Basel in 2021. Website: hpc.dmi.unibas.ch/en/people/ahmed-eleliemy.

Ali Mohammed is a research engineer at HPE's HPC/AI EMEA Research Lab (ERL), Switzerland. From March 2020 to April 2021, he was a postdoctoral researcher at the High-Performance Computing group at the University of Basel, Switzerland. He received his doctoral degree in robust scheduling for high performance computing from University of Basel in 2020. His interests include scheduling, fault tolerance, and performance simulation. Website: hpc.dmi.unibas.ch/en/people/ali-mohammed.

Florina M. Ciorba is an Associate Professor of High Performance Computing at 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. Website: hpc.dmi.unibas.ch/en/people/florina-ciorba.