A Comprehensive Performance Evaluation of the BinLPT Workload-Aware Loop Scheduler

Pedro Henrique Penna, Antônio Tadeu A. Gomes, Márcio Castro, Patrícia Plentz, Henrique Cota De Freitas, Francois Broquedis, Jean-François Mehaut

Received 26 April 2016; | Revised 6 June 2016; | Accepted 6 June 2016

DOI: xxx/xxxx

ARTICLE TYPE

A Comprehensive Performance Evaluation of the BinLPT Workload-Aware Loop Scheduler

Pedro Henrique Penna*[1,2,3] | Antônio Tadeu A. Gomes[4] | Márcio Castro[1] | Patricia D. M. Plentz[1] | Henrique C. Freitas[2] | François Broquedis[3] | Jean-François Méhaut[3,4]

[1]Distributed Systems Research Lab (LaPeSD), Universidade Federal de Santa Catarina, Florianópolis, Brazil

[2]Computer Architecture and Parallel Processing Team (CArT), Pontifícia Universidade Católica de Minas Gerais, Belo Horizonte, Brazil

[3]Compiler Optimization and Run-time Systems (CORSE), Université Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, Grenoble, France

[4]LNCC (National Laboratory for Scientific Computing), Petrópolis, Brazil

Correspondence

*Email: pedro.penna@sga.pucminas.br

Present Address

500, Avenida Dom José Gaspar

Minas Gerais - Belo Horizonte - Brazil

Summary

Workload-aware loop schedulers were introduced to deliver better performance than classical loop scheduling strategies. However, they presented limitations such as inflexible built-in workload estimators and suboptimal chunk scheduling. Targeting these challenges, we proposed previously a workload-aware scheduling strategy called BinLPT, which relies on three features: (i) user-supplied estimations of the workload of the loop; (ii) a greedy heuristic that adaptively partitions the iteration space in several chunks; and (iii) a scheduling scheme based on the Longest Processing Time (LPT) rule and on-demand technique. In this paper, we present two new contributions to the state-of-the-art. First, we introduce a multiloop support feature to BinLPT, which enables the reuse of estimations across loops. Based on this feature, we integrated BinLPT into a real-world elastodynamics application, and we evaluated it running on a supercomputer. Second, we present an evaluation of BinLPT using simulations as well as synthetic and application kernels. We carried out this analysis on a large-scale NUMA machine under a variety of workloads. Our results revealed that BinLPT is better at balancing the workloads of the loop iterations and this behavior improves as the algorithmic complexity of the loop increases. Overall, BinLPT delivers up to 37.15% and 9.11% better performance than well-known loop scheduling strategies, for the application kernels and the elastodynamics simulation, respectively.

KEYWORDS:

workload-aware, loop scheduling, performance evaluation, OpenMP

1 | INTRODUCTION

Evenly distributing the workload of an application among its threads is an NP-Hard minimization problem known as Multiprocessor Scheduling [1]. This problem is significant to the academic community and industry, and it is a hot research topic in High Performance Computing (HPC). For instance, in shared-memory applications, the Multiprocessor Scheduling Problem emerges when scheduling iterations of parallel loops [2,3,4]. In this context, it is traditionally referred as the Loop Scheduling Problem (LSP), and it is more precisely stated by the assignment of independent loop iterations to worker threads such that (a) the overall workload is evenly distributed; and (b) the overall scheduling overhead is minimized.

So far, several loop scheduling strategies have been introduced to address LSP [3,4,5,6,7,8,9], and they mainly rely on two techniques. In the first one, called on-demand scheduling, iterations are assigned on-the-fly at runtime, so that load imbalance and runtime variations may be dynamically handled. In the second technique, called chunk-size tuning, iterations are scheduled in optimally sized batches (*i.e.* chunks) so that: (i) scheduling overheads are mitigated; (ii) load imbalance is further amortized; and (iii) affinity across the iteration space is exploited. Together, these two

techniques may deliver reasonable performance gains in several scenarios. However, since they do not consider any information about the underlying workload of the target parallel loop, scheduling strategies built upon them end up being inherently suboptimal [10,11]. Therefore, to specifically address this limitation, workload-aware strategies were introduced [12,13,14,15]. In a nutshell, this latter class of strategies relies on some knowledge about the workload to adaptively partition the iteration space in chunks and schedule them, thereby further amortizing load imbalance and improving performance.

Despite the fact that workload-aware strategies may achieve better performance than their workload-unaware counterparts, they still face some drawbacks that should be addressed. First, existing workload-aware strategies rely on profiling and statistical regression techniques, and thus are inherently designed for well-behaved workloads. To tackle irregular loops, in which the workload varies drastically from one execution to another, some alternative for estimating the workload on-the-fly is necessary. Second, workload-aware loop schedulers fail to apply their estimations about the workload to effectively schedule loop iterations. Pragmatically, these strategies rely on the on-demand scheduling technique for carrying out this task, and thus they end up achieving suboptimal performance [1]. Furthermore, on-demand scheduling is known to lead to scalability problems due to locking contention in scheduling queues at runtime [12]. Finally, despite several existing workload-aware strategies, none of them is integrated into a widely-employed Application Programming Interface (API), such as OpenMP, TBB or Cilk. Indeed, the integration of workload-aware schedulers in these APIs is not trivial [13], restricting their adoption by the HPC community.

To address the aforementioned problems, in our previous work, we proposed a new workload-aware strategy called BinLPT and presented an integration of it into the OpenMP runtime system of GCC [16]. Our loop scheduler enables superior performance and flexibility than related ones based on three major features: (i) some user-supplied estimation about the workload of the target parallel loop; (ii) a greedy bin packing heuristic that adaptively partitions the iteration space into several chunks; and (iii) a hybrid chunk scheduling scheme based on the Longest Processing Time (LPT) and on-demand rules. This work extends the previous one and delivers the following new contributions to the state-of-the-art:

- **Multiloop Support for BinLPT.** In the original implementation of BinLPT [16], the thread-iteration assignment was recomputed every time for the target parallel loop. While in a trivial use-case this could be acceptable, in scenarios where the target loop is executed several times with minor workload variations, performance degradation would likely follow. To overcome this drawback, we introduce a multiloop support feature to BinLPT's original OpenMP implementation. This new functionality enables a HPC engineer to reuse workload estimations several times for each target parallel loop. Based on the multiloop feature, we discuss how we integrated BinLPT into a large-scale real-world-application: an Elastodynamics Simulator that may be employed by the oil and gas industry for identifying extraction hotspots on a seabed. Furthermore, we present a performance evaluation of this simulator running on the Santos Dumont supercomputer, located in the Brazilian National Laboratory for Scientific Computing (LNCC).

- **A Comprehensive Performance Evaluation of BinLPT.** To deliver wider performance assessment of our scheduler, we carried out a second analysis grounded on three different techniques. First, we used simulations to understand the performance of BinLPT in a wide range of scenarios. Next, we relied on a synthetic kernel to uncover the upper bound performance gains of our strategy. Finally, we employed three scientific kernels, each of which featuring distinct characteristics, to study the performance of BinLPT in more realistic use-case scenarios. Overall, we considered several performance factors, and we run experiments in a 192-core Non-Uniform Memory Access (NUMA) machine to provide a rich analysis.

The remainder of this work is organized as follows. In Section 2, we introduce the LSP and classical loop scheduling strategies. In Section 3, we detail the internals of BinLPT. In Section 4, we present related workload-aware loop scheduling strategies. In Section 5, we describe our evaluation methodology. In Section 6, we discuss the experimental results. Finally, we draw the conclusions of our work in Section 7.

2 | BACKGROUND ON LOOP SCHEDULING

In this section, we introduce the LSP. Then, we analyze the known bounds that are related to it. Such analysis is rarely found in related work, even though it is crucial when proposing a new loop scheduler. Finally, we present some classical loop scheduling strategies and briefly discuss their pros and cons. A discussion on related workload-aware loop scheduling strategies is presented in Section 4.

2.1 | The Loop Scheduling Problem (LSP)

The LSP is an NP-Hard minimization problem that boils down to partitioning an iteration space $\hat{x}$ into n non-overlapping sequences of iterations (chunks) and assigning these n chunks to p threads such that: (i) the number of n chunks that are used to partition the iteration space $\hat{x}$ is reduced; and (ii) the maximum load imbalance between any pair of threads in p is minimized. This formulation captures the relation between the four core variables of the LSP: (i) the loop iteration space $\hat{x}$; (ii) the load of iterations w_k; (iii) the number of chunks n in which $\hat{x}$ will be partitioned; and (iv) the

number of threads p that will process the n chunks in parallel. However, additional variables may also be considered when the problem is analyzed in a real-world context.

Related to the application characteristics, we can enumerate two variables: the scheduling overhead and the memory affinity. The former is an important concern for applications that feature a strong irregularity and rely on loop scheduling strategies to assign chunks of iterations to threads on-the-fly and on-demand. If contention in synchronization structures is costly and the scheduling strategy is frequently invoked, the irregular parallel loop may face severe scalability issues when the number of chunks grows asymptotically [2]. Memory affinity, on the other hand, is related to the temporal and spatial data localities that exist across the iteration space, which greatly impacts the performance of memory-intensive applications. When memory affinity is exploited, the memory hierarchy is efficiently used, thereby reducing contention in buses and other interconnection structures [6,17,18]. Memory affinity is likely to be exploited when using large chunks, which conversely may significantly increase load imbalance.

Orthogonal to variables related to application features, there are those that concern the underlying platform itself. For instance, in Asymmetric Multiprocessing (AMP) machines, the processing capacity and features of each processing unit may greatly differ from one another, and thus impact the overall chunk scheduling performance [19]. Fortunately, such architectural characteristics may be queried at runtime and thus may be handled optimally in a per-platform fashion. Additionally, the availability of the platform may be taken into account if the application runs concurrently with other jobs or processes. However, unlike platform heterogeneity, availability is unpredictable (*i.e.* happens at runtime), which in turn makes it harder to deal with [13].

2.2 | Known Bounds on Scheduling

Recall that the LSP is an instance of the Multiprocessor Scheduling Problem. The latter is a classical problem and has been extensively studied so far, thus in this section, we carry out a discussion on its known bounds aiming at uncovering the bounds on loop scheduling as well. The following analysis considers the core variables that were introduced in the previous section, and it assumes that cores have equal processing capacities.

The lower bound solution for Multiprocessor Scheduling is to assign to each thread a workload that equals the overall workload divided by the number of threads. Indeed, this solution may not be achievable in several instances of the problem because scheduling units may not be arbitrarily divided up (*i.e.* schedule half of a task) and thus there may not even exist a schedule that would lead to such an optimum workload distribution. Nevertheless, this lower bound may be used as a baseline for evaluating how good a given scheduling strategy is. In the LSP, for instance, an analogous optimum strategy would be able to evenly distribute the workload of a target parallel loop among the threads that execute that loop.

On the other hand, an upper bound solution for Multiprocess Scheduling can be obtained by a greedy algorithm that considers tasks in arbitrary order, and assign them on-demand to idle threads [1]. This strategy is known as List Scheduling (LS), which leads to a 2-approximation solution. Indeed, an even tighter bound solution was introduced and it is known as the LPT strategy [1]. This strategy leads to a $4/3$-approximation of the optimum solution and works similarly to LS, but it considers tasks in decreasing order of load instead. Based on the aforementioned algorithms, we can conclude that on-demand fine-grain loop scheduling strategies are bounded to a 2-approximation solution. In this paper, we interchangeably refer to the "LS strategy" as "LS rule" and to the "LPT strategy" as "LPT rule" respectively, unless otherwise stated.

The previous conclusion holds when the scheduling overhead incurred by fine-grain scheduling is negligible. Unfortunately, this is not true in practice, and thus fine-grain on-demand loop schedulers may present worse performance than in theory. However, if the number of iterations in the parallel loop is asymptotically large, medium-grain on-demand scheduling yields to 2-approximation solutions [20,4]. The rationale behind this relies on the observation that if for each batch of chunks that is scheduled there is more than a half of iterations left to be scheduled in subsequent batches, there exists a high probability that all threads will end up receiving an even portion of the overall workload, regardless of its characteristics. In this work, we refer to this outcome as the *Load Imbalance Amortization Principle*.

2.3 | Classical Loop Scheduling Strategies

Loop scheduling strategies boil down to one of the following two approaches: *static*, in which loop iterations are assigned to the threads of the parallel application at compile-time; and *runtime*, in which scheduling decisions are made at runtime, either before or during the execution of the target loop [14]. Static scheduling strategies introduce no runtime overhead, but they are suitable only for parallel loops that feature a compile-time predictable workload. In contrast, runtime strategies are employed to address parallel loops that perform computation on a workload that is not known in advance. The classical loop schedulers that make decisions at runtime are the following:

- **Pure Dynamic Scheduling (PDS)** It assigns individual iterations to threads on-demand. Whenever a thread becomes idle, the next unprocessed iteration of the parallel loop is assigned to it. This strategy relies on the LS rule, and thus achieves good load balancing but at the price of a possibly high runtime overhead.

- **Chunk Self-Scheduling (CSS)** It works like PDS, but instead of assigning iterations one by one, it assigns iterations in equally-sized chunks [2]. Each chunk is a non-empty subset of contiguous iterations of the parallel loop. Small chunk sizes deliver good load balancing, but they likely introduce prohibitive runtime overheads. When the chunk size equals to one, this scheduling strategy degenerates to PDS. In contrast, large chunk sizes avoid this problem but may increase load imbalance. When the chunk size is fine-tuned, however, near-optimal load balancing may be achieved [1,2,10].

- **Guided Self-Scheduling (GSS)** It also assigns chunks of loop iterations to threads on demand, but it dynamically changes their size at runtime according to the Load Imbalance Amortization Principle [3]. More precisely, the size of the next chunk is proportional to the number of remaining iterations divided by the number of threads. The idea of having a decreasing chunk size is to offer a compromise between achieving good load balancing while reducing runtime overhead. Nevertheless, GSS may not deliver good performance when first iterations of a parallel loop are much more time-consuming than the iterations that follow.

- **Factoring Self-Scheduling (FSS)** It works similarly to GSS, but it differs in the way that chunk sizes are determined [4]. To address the scenarios in which GSS does not perform so well, FSS computes the next chunk size by dividing a subsequence of the remaining loop iterations (usually half) evenly among the threads. FSS introduces no significant runtime overhead compared to GSS, and it may deliver better performance.

The aforementioned strategies may deliver reasonable performance to a wide range of irregular parallel loops. However it is important to note that these schedulers do not consider any information about the underlying workload of the target loop to neither partition the iteration space into chunks nor schedule these chunks to threads. As a consequence, these strategies are inherently suboptimal when the workload estimation is available. In this paper, we focus on the problem of scheduling irregular parallel loops in which the workload maybe somehow estimated at runtime. In the next section we present BinLPT, a loop scheduler that uses user-supplied information about the workload to better balance the load of irregular parallel loops [16].

3 | THE BINLPT LOOP SCHEDULER

In this section, we present our workload-aware loop scheduling strategy named BinLPT. First, we discuss how our scheduler works. Next, we detail how the input parameters of BinLPT impact on the output scheduling solution, as well as we present some guidelines for determining them. Finally, we present an overview on the integration of our scheduler with libGOMP, the OpenMP runtime system of the GNU Compiler Collection (GCC).

3.1 | Overview of BinLPT

BinLPT operates in two phases to deliver load balance to irregular parallel loops, namely *chunk partitioning* and *chunk scheduling*. The heuristics used in each phase are depicted in Figure 1, and they are indeed the key features that enable the superior performance of BinLPT. In the *chunk partitioning* phase, the primary goal is to split the iteration space into chunks so as to amortize load imbalance while minimizing the number of chunks that are produced. In this way, runtime scheduling overheads can be reduced. To partition the iteration space, BinLPT relies on a workload-aware adaptive technique that takes as input a user-supplied threshold k, which corresponds to the maximum number of chunks to be generated, and works as follows. First, it computes the average load ω_{avg} for a chunk based on the workload information and k. In Figure 1a, two threads compute a parallel loop with an overall load of 125; k is set to 4 and the average workload is $\omega_{avg} = 31.25$. Next, BinLPT uses a greedy bin packing heuristic that bundles into a single chunk the maximum number of contiguous iterations whose overall load does not exceed ω_{avg} (Figure 1b).

In the *chunk scheduling* phase, the goal is to come up with a chunk/thread assignment that minimizes load imbalance. Therefore, BinLPT relies on a hybrid scheduling scheme that is based on the LPT and on-demand scheduling rules. First, chunks are statically scheduled using the former approach. To do so, chunks are sorted in descending order according to their loads (Figure 1c) and they are statically scheduled to threads using the LPT rule (Figure 1d). Then, during the execution of the parallel loop, whenever a thread finishes processing all chunks that were assigned to it, a chunk that has not yet been processed by another thread is assigned to this thread, on-demand. Figure 1e shows the final thread/chunk assignment after applying on-demand scheduling.

Algorithm 1 outlines the BinLPT scheduler. It takes as input three parameters: an array that gives a load estimation of each iteration in the target parallel loop (A), the maximum number of chunks to generate (k) and the number of working threads (n). Then it returns a multiset (P) that states the thread/chunk assignment. In the depicted notation, T_j denotes the load of thread j; P_{T_j} the set of chunks assigned to thread j; and $C \leftarrow C \cup \{\widehat{c_j}\}$ the inclusion of iteration A_i in the chunk $\widehat{c_j}$. The algorithm starts by computing chunks according to the greedy bin packing heuristic (line 2). This heuristic is implemented by the Compute-Chunks(A, k) function (lines 12 to 23). Then, our scheduler sorts the produced chunks according to their loads (line 3). Next, chunks are statically scheduled following the LPT rule (lines 7 to 10). Later, during the execution of the loop, whenever a thread

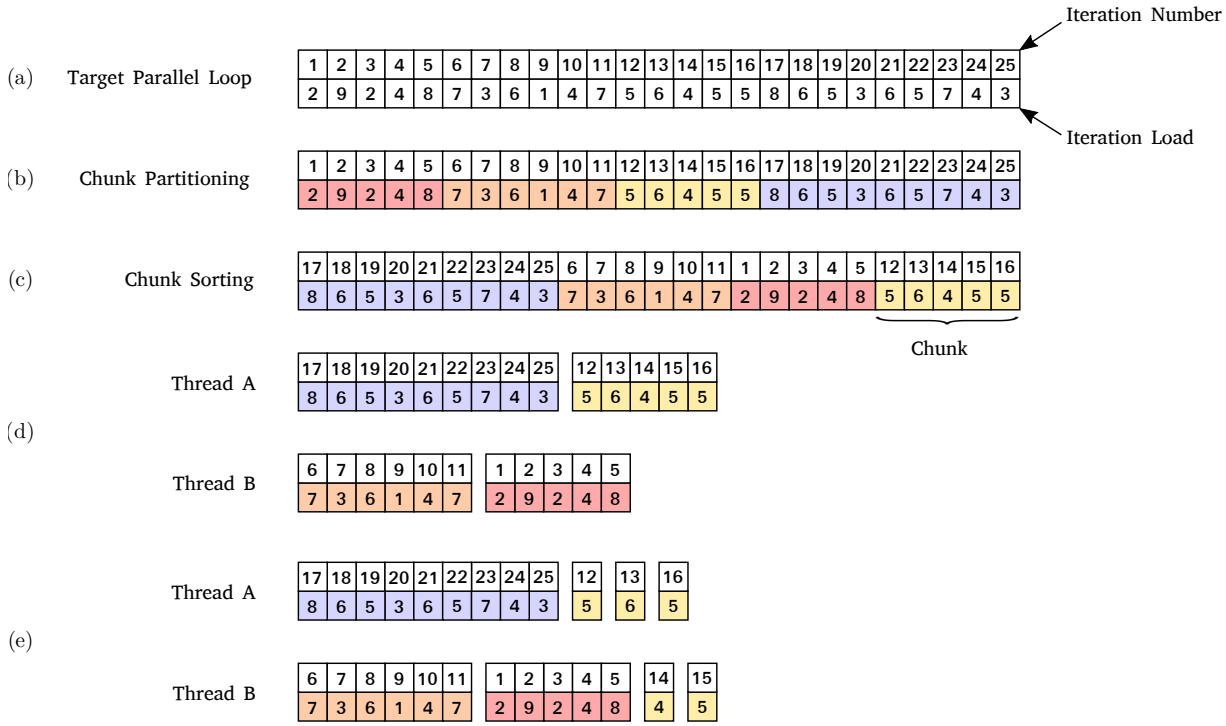


FIGURE 1 Example of loop scheduling using BinLPT: (a) target parallel loop; (b) chunk partitioning; (c) chunk sorting; (d) static scheduling based on LPT rule; and (e) on-demand scheduling.

becomes idle, a new chunk that has not yet been processed is assigned on-demand to this thread, thereby accounting for unpredictable runtime phenomena that may arise.

3.2 | Input Parameters and Scheduling Optimality

The hybrid scheduling scheme, which is based on the LPT and on-demand scheduling rules, enables BinLPT to handle load imbalance for both predictable and unpredictable phenomena in the best known possible way. In contrast, existing workload-aware loop scheduling strategies only rely on the latter technique (*i.e.* on-demand scheduling), even though they are aware of some information regarding the underlying workload for applying the former (*i.e.* LPT rule). Therefore, existing workload-aware loop scheduling strategies are inherently suboptimal because they do not comprehensively exploit the available information about the workload of the target parallel loop. Notwithstanding, it is important to observe that the actual quality of the scheduling solution output by BinLPT is greatly impacted by both the accuracy of the user-supplied estimation of the

Algorithm 1 BinLPT loop scheduling strategy.

1: **function** BinLPT(A, k, n)
2: C ← Compute-Chunks(A, k)
3: Sort(C, descending order)
4: **for** i **from** 0 to n **do**
5: $T_i \leftarrow 0$
6: $P_{T_i} \leftarrow \emptyset$
7: **for** i **from** 0 **to** | C | **do**
8: $T_j \leftarrow \min\ T$
9: $P_{T_j} \leftarrow P_{T_j} \cup \{\widehat{C}_i\}$
10: $T_j \leftarrow T_j + \omega(\widehat{C}_i)$
11: **return** P
12: **function** Compute-Chunks(A, k)
13: $j \leftarrow 0$
14: C ← empty multiset
15: $\widehat{c}_0 \leftarrow$ empty sequence
16: $\omega_{avg} \leftarrow \frac{\sum_{i_j \in A} w_j}{k}$
17: **for** i **from** 0 to | A | − 1 **do**
18: **if** $\omega((\widehat{c}_j, A_i)) < \omega_{avg}$ **then**
19: $\widehat{c}_j \leftarrow (\widehat{c}_j, A_i)$
20: **else**
21: $C \leftarrow C \cup \{\widehat{c}_j\}$
22: $j \leftarrow j + 1$
23: **return** C

workload and the value of the k parameter. The more accurate is the former parameter, the closer the solution of BinLPT will resemble the 4/3-approximated solution of LPT [1]. In contrast, the k parameter controls the maximum number of chunks that will be generated, thereby enabling the engineer to control the overhead of the on-demand scheduler.

The optimal value for k may be determined by iterative refinement. As a general rule to guide this process, the HPC engineer should account for the locking capabilities of the underlying platform. If fast hardware locking mechanisms are available, one should set a high value for k and gradually decrease it, until a desirable trade-off between performance and overhead is reached. Conversely, if locking overhead is too high in the underlying platform, then one should start with a small value for k and gradually increase it. The rationale for this strategy is based on the fact that a high-value for k is likely to force a fine-grain static LPT scheduling, but it may introduce a great overhead at execution time if the application greatly suffers from runtime load imbalance and hardware locking is not supported (or offers poor performance).

The accuracy of the user-supplied workload estimation depends on the estimation technique employed by the HPC engineer. Unlike other workload-aware loop scheduling strategies, BinLPT does not rely on a particular workload estimator, thereby enabling the engineer to pick up one that best meets the requirements of the target parallel loop. The most straightforward approach is to instrument the target parallel loop and profile it, either offline or online. This solution is likely to lead to an accurate estimation when the workload does not change considerably from one execution to another [12]. However, it is important to note that profiling may be time-consuming and non trivial [13]. The second approach is to use compile-time analysis to estimate the load of each iteration. Indeed, this alternative may give a high-quality approximation for numeric parallel loops [21], but it fails when the irregularity arises from the input data itself. Finally, for some parallel loops, the load of each iteration may be estimated from the problem structure itself, as a function of the input workload. Indeed, this approach may yield the most accurate estimation but its usability is application-dependent. As a general rule, the HPC engineer may rely on the following steps to determine which estimation technique should be employed:

- If the target loop presents a well-defined behavior from one execution to another, an offline profiling should yield to an accurate estimation;
- If the target loop does not present a well-defined behavior from one execution to another, but it presents an iterative behavior (*i.e.* it is executed many times), an online profiling technique should yield to an accurate estimation;
- If the target loop does not present a well-defined behavior from one execution to another, nor it presents an iterative behavior, the HPC engineer should design a domain-specific analytic model for estimating the workload of the target parallel loop.

It is worth noting that regardless of the estimation technique that is employed, if the estimated workload is inaccurate, in the worst case the scheduling solution output by BinLPT is asymptotically as bad as the Chunk Self-Scheduling (CSS) scheduler. The rationale for this lies with the observation that the on-demand scheduler of BinLPT will become heavily active and it will schedule iterations according to the LS rule, which provides a 2-approximation solution for the LSP.

3.3 | Integration with libGOMP

As we mentioned before, we implemented BinLPT in libGOMP, which is the OpenMP runtime system that comes with GCC. We made the enhanced version of this runtime system publicly available[1] under the GPL v3 License. Thus, any parallel application implemented with OpenMP can easily use our scheduling strategy. Overall, we introduced four major changes to the mainstream libGOMP project in order to integrate our strategy.

First, we modified the `gomp_loop_runtime_start()` and `gomp_loop_init()` functions. The former invokes the runtime scheduler selected by the user, and we thus added the BinLPT strategy as a new possible option there. The latter function initializes the loop scheduler, and we inserted into it the BinLPT code corresponding to the chunk partitioning phase and the static chunk scheduling based on the LPT rule (Algorithm 1). Second, we added the `gomp_iter_binlpt_next()` function to the library. This function looks up the iteration/thread map provided by the BinLPT strategy and effectively assigns iterations to the corresponding threads. In addition, it performs on-demand chunk scheduling for those threads that have finished processing the chunks initially assigned to them. Finally, to introduce the multiloop support feature, we added two new functions to libGOMP. The first one, named `omp_loop_register()`, registers a parallel loop and returns its unique identifier. The second function, called `omp_set_workload()`, enables the HPC engineer to either set the workload information of a target parallel loop or to re-use a previously-computed thread-iteration assignment.

Snippet 1 illustrates how to use BinLPT in an adjoint convolution operation (an example of how the new multiloop feature of our scheduler may be employed is presented in Section 3.4). To invoke BinLPT, the engineer should select the `runtime` scheduler in the OpenMP `schedule` clause and set the OpenMP environment variable `OMP_SCHEDULE` to `binlpt,k`, where the k parameter controls the maximum number of chunks to be generated by BinLPT. First, the target parallel loop in which BinLPT will be used is registered with the `omp_register()` function (line 3). Next,

[1] Available at: https://www.github.com/lapesd/libgomp

Snippet 1: Adjoint convolution in OpenMP with BinLPT.

```
1 void adjconv(double *a, const double *b, const double *c, unsigned N)
2 {
3   unsigned id = omp_loop_register("main-loop");
4   unsigned *w = calloc(N*N, sizeof(unsigned));
5
6   for (int i = 0; i < N*N; i++)
7     w[i] = N*N - i;
8
9   omp_set_workload(id, w, N*N, true);
10
11  #pragma omp parallel for schedule(runtime)
12  for (int i = 0; i < N*N; i++)
13    for (int j = i; j < N*N; j++)
14      a[i] += F*b[j]*c[i - j];
15 }
```

the workload is estimated based on the number of operations each outer loop iteration will execute (lines 6 and 7). Then, the load estimations are informed to the OpenMP runtime by calling the `omp_set_workload()` function (line 9). The target loop is specified by using the loop identifier previously returned by `omp_loop_register()` and the last parameter is set to true, to point out that the runtime should override any previously-computed thread-iteration assignment for that target loop. Finally, we invoke BinLPT by either selecting the runtime scheduler (line 11) or by calling `omp_set_schedule(omp_sched_binlpt, k)` within the application code.

3.4 | Multiloop Scheduling

Recall that during its static scheduling phase, BinLPT runs the adaptive chunk partitioning heuristic and the LPT rule. Although both steps feature linear space and time requirements, if the target parallel loop is nested within another loop, it turns out that the static scheduling phase of BinLPT will execute many times. This is unavoidable in applications that feature rapidly changing workloads, because a great irregularity is present and thus the workload must be constantly balanced. However, in applications that have their workload slowly changing across the execution of the target loop, an unnecessary overhead is introduced. If substantial workload changes happen only every `SCHED_FREQ` iterations, where `SCHED_FREQ` may be either a constant value or a function of the input workload, BinLPT will actually perform redundant computations. Alternatively, to improve the overall performance, our scheduler could re-use previously computed static scheduling assignments every `SCHED_FREQ` iterations. Therefore, to tackle this problematic scenario, we have introduced the multiloop feature support, which stands out for enabling BinLPT to re-use its previously computed static scheduling assignments. The HPC engineer may state whether or not the static scheduling should run, and thus fine-tune the behaviour of our strategy.

In Snippet 2 we depict an overview on how the multiloop feature of BinLPT may be used in an iterative simulation. We employed a similar strategy to effectively use BinLPT in the Elastodynamics Simulator (see Section 5.4). First, we register the target (inner) loop with the `omp_loop_register()` function. This routine takes as a parameter a symbolic name for the target loop and it returns an unique identifier for that loop. Next, in the outer loop, we inserted the code to exploit the multiloop feature. To avoid introducing additional overheads due to many scheduling calls, we instruct BinLPT to re-use the previously computed static scheduling assignment for the further `SCHED_FREQ`-1 iterations (line 13). We do so by calling the workload estimator with the current simulation state (line 14), and setting the flag for refreshing the current static scheduling (line 15). Finally, the new workload estimation is considered to re-compute the static scheduling for the next target parallel loop, when we call the `omp_set_workload()` (line 18). At the end of the outer loop, the `refresh` variable is reset, so that in the next iteration of the outer loop, the static scheduling will be re-used.

4 | RELATED LOOP SCHEDULING STRATEGIES

Targeting time-step applications with irregular parallel loops, Banicescu[13] proposed Adaptive Weighted Factoring (AWF). In this strategy, the chunk size is dynamically adapted after each step in the application. The newly computed chunk size decreases across the iteration space, likewise in FSS. However, the performance of the threads during the last time-step and their accumulative performance during all the previous ones is additionally considered in this adjustment. To evaluate the performance of AWF, two in-house applications were studied: (i) Laplace's Equation Solver on an unstructured grid using Jacobi's method; and (ii) N-Body Simulations. Pure Static Scheduling (PSS) and FSS strategies were considered as baselines. Experiments were carried out on a synthetically-loaded homogeneous cluster, and the results unveiled that AWF may achieve up to

Snippet 2: Using multiloop feature of BinLPT in an iterative OpenMP application.

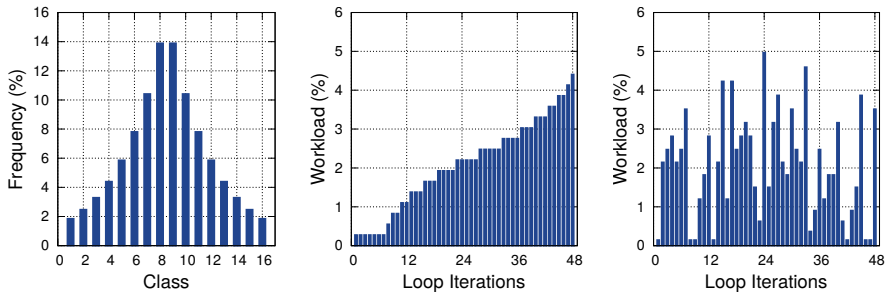
```
1 #define NITERATIONS 1000 /* Number of iterations. */
2 #define SCHED_FREQ    50 /* Scheduling frequency. */
3
4 void simulation(struct state *output, const struct state *input, unsigned N)
5 {
6   unsigned workload[N];
7   bool refresh = false;
8   unsigned id = omp_loop_register("loop0");
9
10  memcpy(output, input, N*sizeof(struct state));
11
12  for (int k = 0; k < NITERATIONS; k++) {
13    if (k%SCHED_FREQ == 0) {
14      predict(workload, output, N);
15      refresh = true;
16    }
17
18    omp_set_workload(id, workload, N, refresh);
19    #pragma omp parallel for schedule(runtime)
20    for (int i = 0; i < n; i++)
21      kernel(output, k, i);
22
23    refresh = false;
24  }
25 }
```

46% better performance than the baseline strategies. Due to the notable performance of AWF, extensions have been proposed to enable its use on non-iterative applications as well[22]. Nevertheless, the enhanced version of this strategy presented a performance that is comparable to the one achieved by FSS.

To address a broader class of applications, Kejariwal *et al.*[14] proposed History-Aware Self-Scheduling (HSS). Unlike AWF, HSS relies on statistical information collected offline via profiling to carry out smarter scheduling. Based on this extra knowledge, at every scheduling round, HSS computes chunk sizes in a decreasing fashion like FSS and considers the load of previously executed iterations and their corresponding actual loads. To assess the performance of HSS, irregular parallel loops extracted from the Standard Performance Evaluation Corporation (SPEC) Benchmarks were studied, and the FSS and AWF strategies were considered as baselines. Experiments were carried out on an in-house simulator, and the results unveiled that HSS may outperform baseline strategies up to 18%.

Based on a similar offline profiling-guided approach to HSS, Wang *et al.*[15] introduced Knowledge-Based Adaptive Self-Scheduling (KASS). This strategy primary targets distributed heterogeneous platforms and works in two phases: a static partitioning phase, and a dynamic scheduling phase. In the first phase, a knowledge-based approach is used to partition iterations of the parallel loop into local work queues of threads, which distributes the total workload approximately equally to the threads. In the second phase, chunks of iterations in local work queues are scheduled by decreasing sizes as in FSS. Each thread gets a chunk from its local queue to execute, and when it finishes the execution of all the chunks in its local queue, it steals chunks from other threads. To evaluate the performance of KASS, two scenarios were studied: (i) parallel loops extracted from the SPEC Benchmarks; and (ii) and three in-house application kernels, namely Over-Relaxation, Jacobi Iteration and Transitive Closure. The GSS, FSS, Trapezoid Self-Scheduling (TSS) and Affinity Self-Scheduling (AFS) strategies were considered as baselines. Experiments were carried out on a Symmetric Multiprocessing (SMP) machine, where artificial workloads were intensionally introduced to promote heterogeneity in the processing capacity of the platform. The results unveiled that for the parallel loops, KASS is up to 16.9% faster than the baseline strategies. On the other hand, for application kernels, KASS achieved up to 21% better performance.

BinLPT differs from the aforementioned strategies in several points. First, it does not rely on a particular workload-estimation technique. Indeed, the HPC engineer is free to couple BinLPT with the one that yields to the best workload estimation for the application. Consequently, the applicability of our strategy is not restricted to time-step applications such as AWF nor to applications that present well-behaved workloads like HSS, which relies on offline profiling techniques. Second, even though existing workload-aware loop schedulers do use their estimations of the workload to partition the iteration space (*i.e.* to create chunks of iterations), they do not use the same knowledge to actually schedule chunks of iterations to threads. Alternatively, BinLPT uses a hybrid scheduling scheme based on the LPT rule and on the on-demand scheduling technique. The former handles workload imbalance in a $4/3$-approximative fashion, while the latter deals with unpredictable phenomena with 2-approximation optimality[1]. Finally, existing strategies are not integrated into any widely-employed parallel programming environment, thus making it harder for one to actually use them in a real-word application. Furthermore, the source code of these strategies is not available, and their reported algorithmic



(a) PDF. (b) Load of iterations. (c) Loop shuffling.

FIGURE 2 Breakdown of synthetic workload generation.

description is not detailed enough to enable a highly-precise in-house implementation of them. In contrast, BinLPT is open-source and is built into GCC's OpenMP runtime.

5 | EVALUATION METHODOLOGY

To deliver a comprehensive performance evaluation of BinLPT, we followed two different approaches. In the first one, we employed simulations, synthetic kernel benchmarking, and application kernel benchmarking, to respectively (i) understand the performance of our strategy in a large number of scenarios; (ii) uncover the upper bound performance gains of our strategy in a real platform; and (iii) to assess the performance of BinLPT on realistic use-cases. In contrast, in the second approach, we used BinLPT in a large-scale real-world software (Elastodynamics Simulator) and evaluated its performance in a production environment. In this section, we discuss the methodology that we adopted for the aforementioned approaches.

5.1 | SimSched: A Loop Scheduler Simulator

Simulation is a useful technique for (i) isolating the core variables of a given problem, (ii) precisely controlling and monitoring the experimental environment, and (iii) evaluating and prototyping solutions with minimum effort. In this work, we relied on SimSched [11] to carry out our simulation experiments. SimSched is an open-source event-driven simulator[2] that enables the performance evaluation of loop schedulers under a variety of workloads with high precision. In this work, we considered the following parameters of SimSched:

- **Scheduling Strategies** SimSched exports an interface that enables both, static and runtime scheduling strategies to be studied. The simulator already offered built-in support for several classical loop schedulers, including PDS, CSS, GSS and FSS (see Section 2). We extended SimSched to additionally support the HSS, KASS and BinLPT strategies.
- **Workload** SimSched may carry out a simulation either based on a user-supplied trace file or a synthetic workload. For the latter case, SimSched relies on four parameters for generating the workload; (i) the number of loop iterations, (ii) the Probability Density Function (PDF) associated to them, (iii) the load of each iteration and (iv) the loop iteration shuffling seed. Figure 2 illustrates how synthetic workload generation happens. First, the PDF is used to generate the frequency of iterations in each class of loop iterations. Then each class of loop iterations is assigned to a different load (Figure 2b). Finally, the iteration shuffling seed models how loop iterations are actually shuffled in a for loop (Figure 2c).

[2]Available at: https://www.github.com/cart-pucminas/scheduler

Algorithm 2 Synthetic benchmark.

```
1:  function Synthetic-Kernel(w, l, f)
2:      a ← 0
3:      switch f do
4:          case linear
5:              m ← w
6:          case logarithmic
7:              m ← w · log2(w)
8:          case quadratic
9:              m ← w · w
10:     for i from 0 to m do
11:         for j from 0 to l do
12:             a ← a + 1
13:     return a
14: function Synthetic-Benchmark(n, w, s, k, l, f)
15:     set scheduler to s
16:     set number of threads to k
17:     parallel for i from 0 to n do
18:         Synthetic-Kernel(wi, l, f)
```

5.2 | SchedBench: A Synthetic Kernel

While simulations in SimSched enable an upper bound assessment of loop scheduling strategies, benchmarking with synthetic kernels would allow such analysis to be carried out in a real experimenting environment. Unfortunately, no such a tool was publicly available for use, thus we implemented our own loop scheduler synthetic kernel benchmark. We made this synthetic kernel publicly available[3].

Our synthetic kernel benchmark, entitled SchedBench, was implemented in OpenMP and it consists in an extension of the kernel proposed by Mark Bull *et al.*[12]. SchedBench (Algorithm 2) performs embarrassingly parallel computations on a private variable and thereby benchmarks the load balancing performance of a loop scheduler. This synthetic kernel takes six input parameters: (i) *n*, the number of loop iterations; (ii) *w*, the load associated to each iteration; (iii) *s*, the scheduling strategy to use; (iv) *k*, the number of threads; (v) *l*, the load of one operation in the synthetic kernel; and (vi) *f*, the computing complexity of the synthetic kernel. First, the benchmark sets the loop scheduling strategy *s* and the number of working threads *n* to use. Next, it performs dummy computations in a parallel loop whose width equals n. Note that the number of operations that the synthetic kernel actually executes is proportional to *w*, *l* and *f*. The first parameter models the computing load associated to an iteration. The second parameter adjusts this load, so as it is costly enough to be benchmarked. On the other hand, the third parameter models the computing complexity of the kernel. That is, linear ($O(n)$), logarithmic ($O(\log n)$) or quadratic ($O(n^2)$). The more complex the kernel is, the stronger is load imbalance.

5.3 | Application Kernels

Although synthetic kernel benchmarking enables an upper bound performance analysis of loop scheduling strategies, this technique inherently fails to deliver a more realistic assessment. For instance, data access patterns and instruction execution flows are too artificial. Furthermore, branch prediction, floating point and vector units, as well as instruction pipelining and thread synchronization mechanisms are not exercised. Therefore, as an attempt to fill this gap, we selected three application kernels to study. These kernels are of great importance to the scientific community, span over different Dwarfs[23] and feature an irregular computation.

SMV is an application kernel from Sparse Linear Algebra that performs a multiplication between a sparse matrix and a dense vector. Besides finding applications in several scientific and engineering domains[24], sparse matrix-vector multiplication is a frequently studied application kernel within the context of loop scheduling[25,26]. We extracted the SMV kernel from the Conjugate Gradient application from the NAS Parallel Benchmarks (NPB)[27]. In the SMV kernel, the sparse matrix is stored in compressed row format so that memory can be saved and data affinity exploited. The matrix is tiled in several blocks in a row-fashion, and these blocks are processed in parallel. The actual number of floating point operations required to process each block varies accordingly to the number of non-zero elements in each block. Dense blocks are more costly to process than sparse blocks. We relied on this observation to feed workload-aware loop scheduling strategies in our experiments.

MST is a Graph Traversal application kernel that clusters a set of data points using the minimum spanning tree algorithm. Clustering strategies find applications in different fields, such as Cluster Analysis, Circuit Designing and Networking[28]. The MST kernel works as follows. First, the Euclidean space is partitioned into several regions (Figure 3a). Then, Prim's Algorithm is executed on each of them (Figure 3b). The resulting minimum spanning tree on each region clusters local data according to the minimum Euclidean distance. Finally, minimum spanning trees are recursively merged, according to the minimum Euclidean distance, and the final data cluster is produced (Figure 3c). In this kernel, note that the cost for computing each local minimum spanning tree varies accordingly to the number of points that lie within each region. We relied on this observation to feed the workload-aware loop schedulers that we considered in our experiments.

[3]Available at: https://www.github.com/lapesd/libgomp-benchmarks

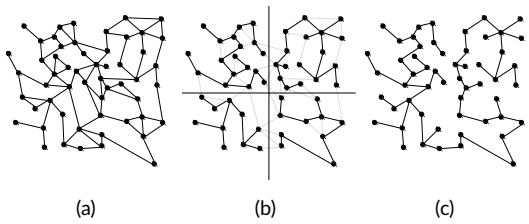


FIGURE 3 Computation performed by the MST kernel. (a) Initial data. (b) Partitioned data. (c) Clustered Data.

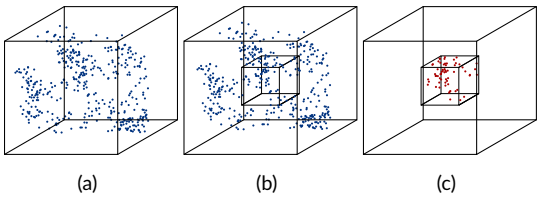


FIGURE 4 Computation performed by the LavaMD kernel. (a) Physical domain. (b) Domain decomposition. (c) Particle interactions.

LavaMD is an application kernel from Computational Fluid Dynamics (CFD) that performs N-Body Simulations. N-Body Simulations find applications in several scientific and engineering domains [29], and are frequently studied within the context of loop scheduling [13,15]. LavaMD was extracted from the Rodinia Benchmarks Suite [30], and it carries out a high-resolution simulation of the pressure-induced solidification of molten tantalum and quenched uranium atoms in a finitely-sized three-dimensional domain. The LavaMD kernel works as follows. The 3D domain (Figure 4a) is decomposed into several equally-dimensioned n^3 boxes (Figure 4b). In this decomposition, any box has 26 adjacent boxes, except for boxes that lie within the boundaries of the domain, which in turn have fewer neighbors. At each computing step, particles interact only with other particles that lie within a user-specified cutoff radius, since ones at larger distances exert negligible forces (Figure 4c). Therefore, the box dimensions are chosen such that the cutoff radius of any particular box does not span beyond the boundaries of an adjacent box. Note that the actual number of interactions to compute for a given particle α is proportional to the number of particles in the same box of α plus the total number of particles among all the boxes that surround α's box. We used this knowledge to estimate the computing load of each box when using workload-aware strategies in our experiments.

5.4 | Elastodynamics Simulator

The Elastodynamics Simulator relies on a family of finite element methods called Multiscale Hybrid-Mixed (MHM) [31,32] to carry out high-order precision simulations of problems with highly heterogeneous coefficients. The simulator computes the numerical solution of each variable of interest (e.g. displacement) as a direct sum of a solution to a global problem (defined over the skeleton of a "coarse" mesh) and the solutions to local problems (applied to element-wise sub-meshes). The local problems are embedded in an upscaling procedure that provides multiscale basis functions to the global problem.

A software library for the MHM methods was developed in C++ and it offers hybrid parallelism by means of MPI and OpenMP. The MHM software library is built on top of key computing primitives that bear correspondence with the major mathematical building blocks of the MHM method. These primitives are combined to implement the Elastodynamics Simulator as depicted in Algorithm 3. The splitProblem primitive divides the original problem posed over a coarse mesh $\mathcal{T}_H$ into a global problem (the 1st MHM-level) and a set of local problems (the 2nd MHM-level), one for each element K_t in the coarse mesh. This primitive returns the set of such local problems as $P_{\mathcal{T}_H}$. When MPI is used, the R MPI processes that make part of a MHM simulator are omnipotent; each MPI process $r \in R$ is responsible for running its own *share* of the splitProblem primitive.

Algorithm 3 Pseudocode of *main* procedure of a MHM simulator for the elastodynamics equations.

Require: $\mathcal{T}_H = \{K_t\}_{t \in T}$ ▷ Mesh with indexed elements
Require: $u_H^0 = \{u_t^0\}_{t \in T}$ ▷ Initial solution for each element of the mesh
Ensure: $u_H = \{u_t\}_{t \in T}$ ▷ Current solution at each element of the mesh
Ensure: $\lambda_H = \{\lambda_t\}_{t \in T}$ ▷ Local contributions from global problem to each element of the mesh

```
P_{T_H} ← splitProblem(T_H)
u_H ← u_H^0
for i = 1, NumTimesteps do
    S_{T_H} ← ∅
    for all t ∈ P_{T_H} do
        s_t ← solveLocalProblem(t, u_t)
        S_{T_H} ← S_{T_H} ∪ s_t
    λ_H ← solveGlobalProblem(S_{T_H})
    for all t ∈ P_{T_H} do
        u_t ← sumLocalAndGlobalSolutions(λ_t, s_t)
```

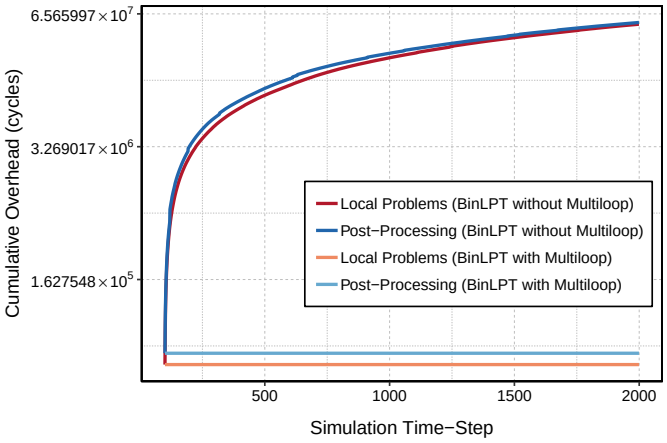


FIGURE 5 Scheduling overhead analysis in MHM when using BinLPT.

This share corresponds to the execution of splitProblem over a partition $\mathcal{T}_H^r$ of $\mathcal{T}_H$. Such a partitioning configures a static process scheduling at the 1st MHM-level that is kept until the end of the simulation. The solveLocalProblem computes contributions s_t from local problems associated with each specific element K_t in the coarse mesh. These contributions are obtained through the solution of a set of independent linear systems, whose computations are embarrassingly parallel. Therefore, within each MPI process, its set of local problems is scheduled by an OpenMP loop so that each local problem is solved in a single core, with the intention to make as many local problems be solved in parallel in a single computing resource as the number of cores the resource provides. Importantly, within each element K_t in the coarse mesh, a sub-mesh with a different level of refinement can be defined – this is the first source of computational imbalance in OpenMP loops within each MPI process. The solveGlobalProblem primitive gathers the contributions s_t from the local problems to assemble a global linear system. This step is not embarrassingly parallel; the distributed version of the MUMPS[4] sparse direct solver is used for solving the global linear system. Finally, owing to the sumLocalAndGlobalSolutions primitive, the numerical solution u_t at each element K_t of the mesh can be computed (this is referred to as '*post-processing*' the final numerical solution in the MHM methods). Within each MPI process the computation of u_t at each element K_t in the coarse mesh is also scheduled by an OpenMP loop so that each u_t is computed in a single core. The Eigen library[5] is used for computing the various matrix and vector operations needed in this phase. Again, as different levels of refinement can be defined within each element of the coarse element, a second source of computational imbalance arises within each MPI process.

In order to integrate BinLPT into the Elastodynamics Simulator, we coupled the profiling technique with the multiloop support feature of our scheduler as follows. For the first 5% of the simulation time-steps, we applied the dynamic scheduling strategy to each OpenMP loop in all MPI tasks and we measured the computational cost of iterations. We carried out this measurement by means of clock cycles using hardware registers and special machine instructions of the underlying platform, in order to mitigate profiling overhead. Then, in the last time-step of the first 5%, we computed the average cost of each iteration, for each OpenMP loop, and fed BinLPT with that information. Finally, in the remaining 95% simulation time-steps, we relied on the previously computed BinLPT scheduling and the multiloop feature of BinLPT to re-apply our scheduling strategy to each OpenMP loop in all MPI tasks.

With multiloop scheduling, we were able to reduce scheduling overheads in the MHM application. Figure 5 quantifies this improvement (details about the configuration of this experiment are shown in Section 5.5). In this plot, we present the cumulative overhead for solving the local problems and carrying out the post-processing in MHM when using BinLPT without the multiloop featuring (*i.e.*, invoking it at each application time-step) and when employing the multiloop feature. Overall, we noted that without the multiloop feature, the cumulative scheduling overhead grows linearly with the number of iterations, whereas when employing the multiloop feature, this performance drawback is steady. The reason for this relies on the observation that, when using the multiloop feature, the actual static scheduling in BinLPT happens just once, whereas a naive use of our strategy incurs in multiple calls to the static scheduler. As a final remark, it is important to note that in real-world simulations, such as MHM, thousands to millions of time-steps are often carried out. Therefore, even though the scheduling overhead in each iteration is relatively small, the cumulative overhead may account for a significant increase in the execution time and energy consumption of the application.

[4]Available at: http://mumps.enseeiht.fr
[5]Available at: http://eigen.tuxfamily.org

5.5 | Experimental Design

We performed three sets of experiments, namely *Simulation*, *Synthetic Kernel Benchmarking* and *Application Kernel Benchmarking*. The parameters of each experiment set is presented in Table 1 and are discussed next. The *Workload PDFs* are frequently studied in the context of the Loop Scheduling Problem [20,13,33,34]. They were generated using the synthetic workload generator of SimSched with a 16-point sampling precision, and with the following parameters: (i) Exponential $\gamma = 0.2$; (ii) Gaussian $\mu = 2.5$ and $\sigma = 1.0$; and (iii) Uniform $\alpha = 0.0$ and $\beta = 1.0$. The values of these parameters were chosen so that they would be consistent with related work [33,10]. The range of *Workload Shuffling Seed* in the *Simulation* was intentionally calculated to be big enough to yield to confidence intervals of 95%. The *Loop Scheduling Seed* for the *Synthetic Kernel Benchmarking* and *Application Kernel Benchmarking* were randomly chosen within this range. *Loop Sizes* and *Problem Sizes* were chosen so as to reflect the full processing capacity of the experimental platform (detailed later in this section). Baseline strategies were selected to be consistent with related works [13,14,15]. The GSS (`guided`) and CSS (`dynamic`) loop schedulers are shipped with libGOMP by default[6], thus we used them in our *Synthetic Kernel Benchmarking* and *Application Kernel Benchmarking*. However, the HSS scheduler is not shipped with libGOMP neither is publicly available. Therefore, we implemented and integrated this strategy in libGOMP using the algorithmic description presented by Kejariwal *et al.* [14]. It is worth noting that we chose to implement HSS in libGOMP and take it as our baseline in benchmarking experiments because it presents better performance than AWF [14,22], and KASS was conceived specifically to target distributed heterogeneous platforms (*i.e.* computer clusters and grids), which are not addressed in OpenMP. Still, we present results for GSS, PDS, KASS and HSS in SimSched. Chunk sizes for the `guided` and `dynamic` schedulers as well as the k_1 parameter for HSS and the k_2 parameter for BinLPT were carefully selected based on earlier experiments that revealed them to be the optimal values for each scenario. In each experiment set, we adopted the following experimental design:

- **Simulation:** We used SimSched to carry out a state-space exploration of the Loop Scheduling Problem, and thus assess the performance of BinLPT in a great number of scenarios. We set the number of threads to 192 to reflect the full computational power of the experimental platform; and we adopted a full factorial experimental design, thereby resulting in 9216 scenarios for each strategy. In all experiments of this set, we fed HSS, KASS and BinLPT with perfect estimations of the workload.
- **Synthetic Kernel Benchmarking:** We used SchedBench to benchmark the overall performance of BinLPT in a real experimental environment, when varying the *Workload PDFs* and *Loop Sizes*. We additionally studied the impact of different *Kernel Complexities* in the performance of BinLPT. The levels for the latter parameter were chosen to match the complexity of widely studied algorithms. In this experiment, we also set the number of threads to 192 and adopted a full factorial experimental design, thereby resulting in 72 different scenarios for each strategy. In all experiments of this set, we fed HSS and BinLPT with perfect estimations of the workload.
- **Application Kernel Benchmarking:** We considered the three application kernels, for say LavaMD, MST and SMV, to evaluate the performance of BinLPT within real-world contexts. For this experiment, we adopted a fractioned experimental design, where we varied the number of threads from 24 to 192, with a constant step of 24. Overall, we studied 21 different scenarios for each scheduling strategy. In all experiments of this set, we fed BinLPT with accurate, but not perfect, estimations of the workload computed at runtime (see description of kernels in Section 5.3) and HSS with perfect estimations obtained with offline profiling.

TABLE 1 Parameters for simulation, synthetic kernel and application kernel experiments.

Parameters	Levels for Simulation	Levels for Synthetic Kernel	Levels for Application Kernels
Workload PDF	Exponential, Gaussian, and Uniform	Exponential, Gaussian and Uniform	Exponential
Workload Shuffling Seed	{1...384}	308	308
Loop Size	{384, 768, 1152, .., 3072 }	{384, 768, 1152, .., 3072 }	–
Chunk Sizes	`guided` {1}, `dynamic` {1}, KASS, HSS {1}, BinLPT {384, 768, 1536}	`guided` {1, 2, 4}, `dynamic` {1, 2, 4}, HSS {1, 2, 4}, BinLPT {384, 768, 1536}	`guided` {2, 3, 4}, `dynamic` {2, 3, 4}, HSS {2, 3, 4}, BinLPT {384, 576, 768 }
Number of Threads	192	192	{24, 48, 96, 120, 144, 168, 192}
Kernel Load	–	2^{25} integer additions	–
Kernel Complexities	–	Linear, Logarithmic and Quadratic	–
LavaMD (Grid Size)	–	–	$11 \times 11 \times 11$
SMV (Matrix Size)	–	–	$(192 \cdot 2^{11}) \times (192 \cdot 2^{11})$
MST (Number of Points)	–	–	2^{22} $\mathbb{R}^2$

[6]The OpenMP community pragmatically refers to GSS, PDS, and CSS as `guided`, `dynamic` (1) and `dynamic` (n), respectively.

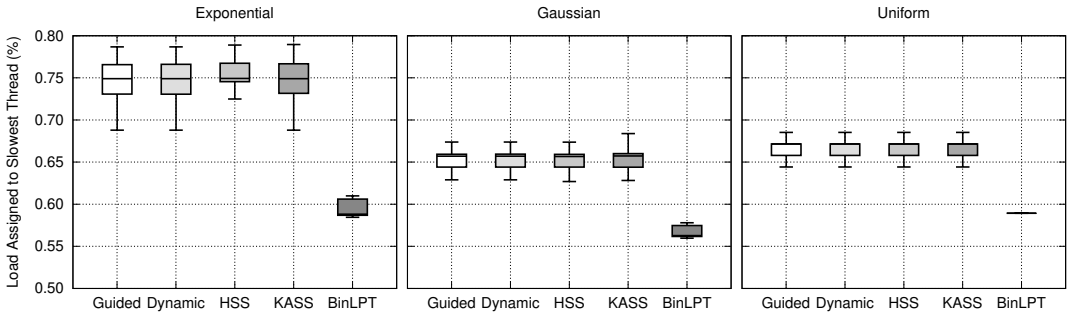


FIGURE 6 Assigned load per PDF in the simulation.

- **Elastodynamics Simulator** We considered the solution of a problem for a linear model posed over a 3D domain with dimensions $[10\text{ km}, 10\text{ km}, 4.5\text{ km}]$. The model represents a wave propagation phenomenon in a domain with 16 layers of different materials, triggered by an explosive source buried at a depth of 10 m below the center of the upper surface, with a wavelet of 0.2 s of duration. This model has been based on the HPC4e Seismic Test Suite (https://hpc4e.eu). A total amount of 16 tasks were dispatched over 8 nodes of the experimental platform. This renders a total amount of 48 local problems per MPI task, to be scheduled over 12 OpenMP threads. To better capture the explosive source, more refined sub-meshes have been applied to the elements surrounding it in the coarse mesh, thus resulting in an important imbalance. In this experiment, we fed BinLPT with accurate, but not perfect, estimations of the workload computed at runtime (see description of the application in Section 5.4).

In the *Synthetic Kernel Benchmarking* and *Application kernel Benchmarking*, we carried out five replications of each possible configuration to account for the inherent variance of the measures in the experimental environment. For each replicate, the actual order in which individual runs were executed was randomly determined. This approach ensures that experimental results and errors are independent and identically distributed (i.i.d) random variables. In our experiments, the maximum relative standard deviation error (σ/μ) observed was below to 1.0%. Furthermore, we considered the following performance metrics in our analysis [22]:

- *Performance* (ρ): the ratio of the total amount of work to the parallel time. Ideally, performance should scale up linearly with the number of threads.
- *Coefficient of Variance (C.o.V)* (λ): the ratio between the standard deviation and the mean execution time of the threads. Near-zero C.o.V suggests that there is no load imbalance, whereas a C.o.V near one indicates the opposite.

Experiments with the synthetic and application kernels were carried out on an SGI Altix UV 2000 machine, which features 24 cache coherent NUMA nodes interconnected through SGI's proprietary NUMAlink6 (bidirectional). Each node has an Intel Xeon E5 Sandy-Bridge processor and 32 GB of DDR3 memory. Overall, this platform features 192 physical cores and 768 GB of memory. In the experiments running on this machine, we disabled Simultaneous Multithreading (SMT) and we used a first-touch memory allocation strategy coupled with a compact thread affinity policy to mitigate runtime NUMA effects. On the other hand, experiments with the Elastodynamics Simulator were conducted in the Santos Dumont Supercomputer, which is installed at LNCC (http://sdumont.lncc.br). This platform has 756 computing nodes, each of which featuring two 12-core Intel Xeon E5 processors running at 2.4GHz. Each node has 64 GB DDR3 RAM and the nodes are altogether interconnected by a 112.752 GB/s Infiniband FDR network. In this paper, we used the full capacity of SGI Altix UV 2000 (192 cores) and 8 nodes of Santos Dumont Supercomputer (192 cores).

6 | EXPERIMENTAL RESULTS

In this section, we discuss our experimental results. We carry out this analysis in a top-down fashion, first unveiling simulation results; then uncovering synthetic kernel and application benchmarking results; and finally presenting the results observed for the Elastodynamics Simulator. As a side remark, unless otherwise stated, we depict experimental results for the best chunk size configuration for each strategy in all plots.

6.1 | Simulation

Workload PDF Breakdown Figure 6 presents the load assigned to the slowest thread per PDF in the simulations when fixing the *Loop Size* in 768 iterations, and varying the *Workload Shuffling Seed*. In these plots, the median is evidenced, and the whiskers extend from each end of the box for a range equal to $1.5\times$ the interquartile range. Furthermore, it is worth noting that y-axis starts at 0.5%, because the load assigned to the most overloaded thread should be at least $100/192 \approx 0.52\%$ (*i.e.* optimal solution for 192 threads).

Overall, the simulation results show that BinLPT achieves better results regardless the PDF type. When compared to the baseline strategies, our strategy may deliver from $1.14\times$ (Gaussian PDF) to $1.27\times$ (Exponential PDF) superior workload balancing. Furthermore, results point out that there is no statistically significant difference between the baseline strategies themselves, by means of the load that is assigned to the most overloaded thread. Indeed, this result further evidences the fact that baseline strategies do not effectively account for the workload when scheduling chunks – HSS and KASS only do so when partitioning the iteration space into chunks. In an SMP platform, the clever partition scheme of KASS for heterogeneous processors is not active, and the scheduling itself degenerates to an affinity version of GSS. Finally, when observing the interquartile ranges, we noted that BinLPT presented the shortest one, thus suggesting that it is less sensitive to how the workload is distributed across the loop.

6.2 | Synthetic Kernel Benchmarking

Overhead Scaling The scheduling overhead of on-demand loop scheduling strategies depends on the number of chunks they produce[22], due to their required synchronizations in the scheduling queue. In large-scale computational environments, high core-counts leads to contention on scheduling data structures. Therefore, the higher is the number of chunks produced by the loop scheduler, the higher will be the time wasted on internal locks in the runtime system.

In this context, recall that the number of chunks that are produced by BinLPT and the considered baseline strategies may be somehow controlled. In the `guided` and `dynamic` strategies, the number of chunks that are generated depends on the size $|\widehat{\mathrm{x}}|$ of the iteration space (*i.e. Loop Size*). In the former strategy, the number of chunks grows proportionally to $\mathrm{O}(\log|\widehat{\mathrm{x}}|)$, whereas in the latter strategy it grows with $\mathrm{O}(|\widehat{\mathrm{x}}|)$. In both strategies, the granularity of the chunk sizes may be further fine-tuned according to a parameter b. For instance, `(dynamic,1)` will cause `dynamic` to split the iteration space into unit-sized chunks ($\mathrm{b}=1$). On the other hand, `(guided,2)` instructs `guided` to generate chunks which are not smaller than 2 ($\mathrm{b}=2$). In the HSS and BinLPT strategies, the chunk size dynamically adapts to the characteristics of the underlying workload, thus creating a variable number of chunks. The granularity of the smallest chunk in HSS may be adjusted by a parameter k_1; and the actual number of chunks produced by BinLPT may be fine-tuned by its k_2 parameter. For instance, `(HSS,2)` will cause HSS to generate chunks that are not smaller than $\mathrm{k}_1=2$ iterations; and `(BinLPT,288)` instructs BinLPT to produce at most $\mathrm{k}_2=288$ chunks, pragmatically.

Figure 7 presents the number of chunks generated by each strategy for an Exponential-generated workload when varying the size of the iteration space at a constant increase (384 iterations). We observed similar behaviors for the other workloads (*i.e.* Gamma- and Gaussian-based), and thus we omitted them due to space limitations. Overall, the results show that the number of chunks produced by BinLPT is far smaller than the one produced by `guided`, `dynamic` and HSS loop schedulers. The number of chunks generated by `guided` and HSS grows logarithmically, and for `dynamic` it grows linearly; whereas for BinLPT, it grows approximately with $(\mathrm{k}\cdot\log|\widehat{\mathrm{x}}|)/|\widehat{\mathrm{x}}|$.

Kernel Complexity Breakdown Figure 8 depicts the Performance (ρ) results per *Kernel Complexity*, when varying the loop scheduler strategy and their parameters, and fixing the *Loop Size* in 1536 iterations, for an Exponential-generated underlying workload. In these plots, the median is

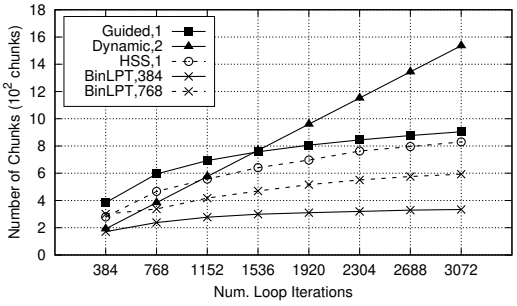


FIGURE 7 Number of chunks output by strategies.

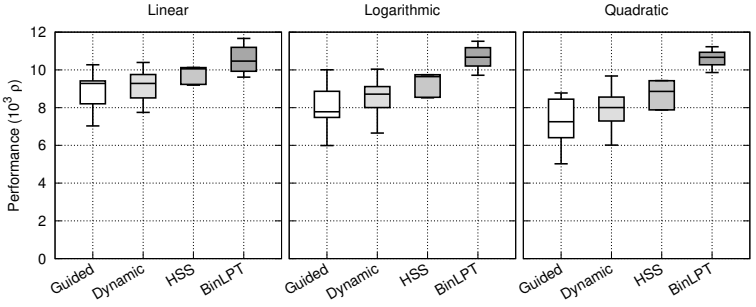


FIGURE 8 Breakdown of kernels for synthetic benchmarking.

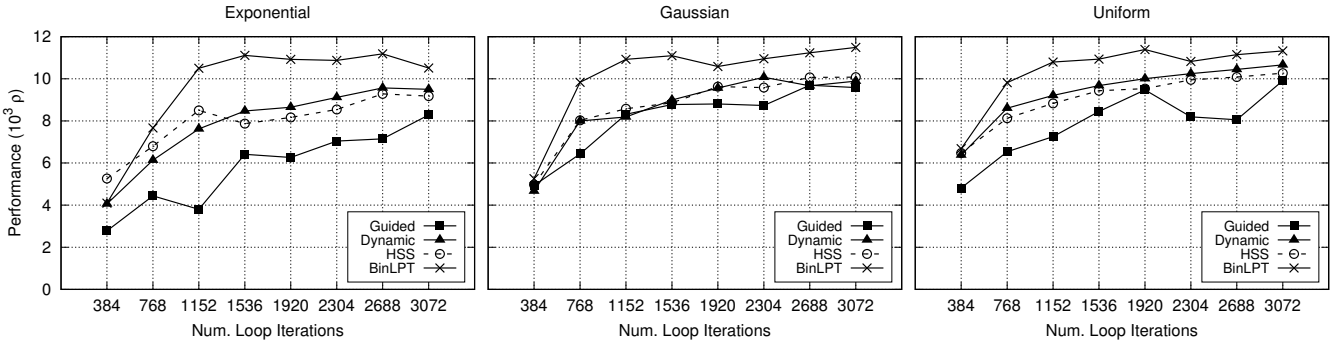


FIGURE 9 Synthetic benchmarking results for workload scaling.

evidenced, and the whiskers extend from each end of the box for a range equal to $1.5\times$ the interquartile range. An overview analysis uncovers two remarkable conclusions about BinLPT. First, our strategy leads to superior Performance (ρ) than the baseline strategies, regardless the kernel complexity. Second, BinLPT delivers constant and more accurate Performance (ρ) when the kernel complexity increases. In contrast, baseline strategies are significantly impacted by the kernel complexity. The greatest Performance (ρ) gap was observed for a Quadratic kernel. In this scenario, BinLPT performed $1.77\times$, $1.27\times$ and $1.22\times$ better than `guided`, `dynamic` and HSS in mean values, respectively. We observed similar results for Gaussian- and Uniform-generated workloads and thus we omitted their respective plots due to space limitations.

Loop Size Scaling Figure 9 presents Performance (ρ) results per *Workload PDF*, when fixing the *Kernel Complexity* in Quadratic and varying the iteration space (*i.e. Loop Size*). In these plots, run-time synchronization overheads were filtered out; and each point pictures the best chunk size configuration for each strategy. Overall, the results unveiled that BinLPT turns out to be the best loop scheduling strategy. The highest Performance (ρ) observed for BinLPT was for the scenario with 1536 iterations Exponential-generated workload, where it delivered at least 21.25% superior Performance (ρ) than the best baseline, `(dynamic,1)`. The smallest Performance (ρ) gain observed for BinLPT was for the scenario with 2304 loop iterations and Uniform-generated workload, where it presented 4.76% of performance increase over `(dynamic,1)`. The worst Performance (ρ) drop was for the scenario with 384 loop iterations and exponential-generated workload.

In the worst-case scenario for BinLPT, the scheduling granularity was not fine-grained enough for amortizing load imbalance. More precisely, BinLPT generated 1024 chunks, while `dynamic` generated $2.25\times$ more chunks. Putting it differently, a fair comparison between BinLPT and the baselines should account for the equivalence between the number of chunks generated by them and the number of chunks produced by our strategy. Recall that `(BinLPT,k)` splits the iteration space in at most k variable-size chunks, whereas `(dynamic,1)` produces unit-sized chunks (b $=$ 1). Therefore, for instance, a fair comparison would be in the scenario with 2304 iterations, between `(BinLPT,2304)` (at most 2304 chunks) against `(dynamic,1)` (2304 chunks). When accounting for such observation, BinLPT may deliver up to $1.44\times$ better Performance (ρ) than the best baseline strategy.

As a final remark, a note on the results observed for HSS is essential. Even though this scheduler presented similar performance to `(dynamic,1)`, we recall that we filtered out synchronization overheads from these plots. Overheads observed for `guided`, `dynamic` and BinLPT were far inferior to the execution time of the benchmark itself (*i.e.* less than 1%). However, for the HSS scheduler synchronization overheads dominated execution time of experiments, thus yielding to effective Performance (ρ) results that were at least $2\times$ worse than the other strategies. The rationale for this behavior relies on the fact that HSS relies on costly locking structures (*i.e.* spin-locks). In contrast, default OpenMP schedulers make use of platform-dependent features (*i.e.* atomic machine instructions) to mitigate synchronization overheads; and BinLPT is a lock-free loop scheduler. For this reason, in the remainder of this work, we omit comparison results with HSS.

6.3 | Application Kernel Benchmarking

SMV Kernel Figure 10 presents Performance (ρ) results for an Exponential-generated workload when varying the number of working threads. These results picture the following scheduling configurations: `(guided,4)`, `(dynamic,4)` and `(BinLPT,768)`. Recall that the SMV performs a sparse matrix-vector multiplication. This kernel presents a linear complexity ($O(mn)$) and presents a moderate memory affinity across the iteration space. Overall, BinLPT delivers better scalability to this kernel than baseline strategies, but performance differences were not highly expressive due to the low complexity of the kernel itself and the fine-grain scheduling employed in BinLPT. A more detailed analysis uncovered that the three loop scheduling strategies present similar performance when up to 120 working threads are used. However, beyond

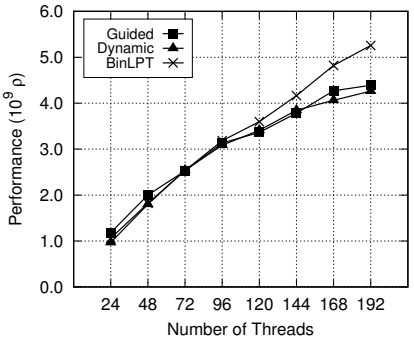


FIGURE 10 Performance (ρ) for SMV.

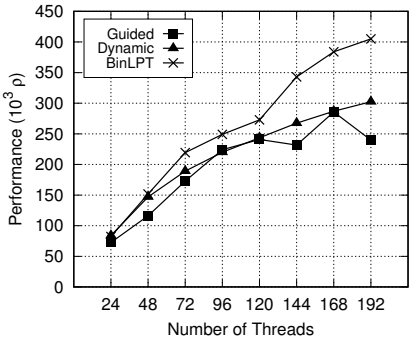


FIGURE 11 Performance (ρ) for MST.

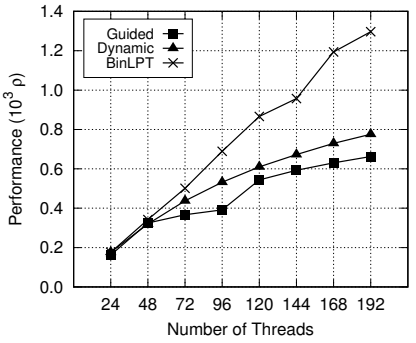


FIGURE 12 Performance (ρ) for LavaMD.

this point, BinLPT stood out as the best strategy. The greatest performance difference was observed when the full processing capacity of the platform was used, with BinLPT delivering 15.78% better Performance (ρ) than baseline strategies (Figure 10). When analyzing C.o.V (λ) results, we observed that BinLPT yields to slower variance values, thereby strengthening our hypothesis that BinLPT delivers better load balancing.

MST Kernel Figure 11 presents Performance (ρ) values for a Exponential-generated workload, when varying the number of working threads. These results concern the following scheduling configurations: `(guided,2)`, `(dynamic,2)` and `(BinLPT,576)`. This kernel features an $O(v^2)$ complexity and an unpredictable memory access pattern, thereby introducing execution irregularities at run-time and favoring on-demand scheduling strategies (*i.e.* `guided` and `dynamic`). Overall, results unveiled that BinLPT delivers better scalability than baseline strategies. Performance (ρ) values showed that when BinLPT faces a strong scaling scenario, it delivers quasi-linear scalability, and a maximum performance gain of 37.5% and 25.0% in contrast to `guided` and `dynamic` loop scheduling strategies, respectively. On the other hand, for C.o.V (λ) results, we have noticed that BinLPT presented a linear-growth behavior, likewise `guided` and `dynamic`. This outcome suggests that there is still room for improvement concerning parallel loops featuring run-time related load imbalance, such as irregular memory accesses.

LavaMD Kernel Figure 12 presents Performance (ρ) values for an Exponential-generated workload, when varying the number of working threads. These results concern the following scheduling configurations: `(guided,2)`, `(dynamic,2)` and `(BinLPT,768)`. Recall that LavaMD performs n-body simulations on a 3D domain. This kernel presents $O(n^3)$ complexity and features unbalanced CPU intensive computations. When analyzing the results, we observed that BinLPT delivers quasi-linear strong scaling Performance (ρ) scalability (Figure 12), in contrast to `guided` and `dynamic` that presented logarithmic Performance (ρ). When analyzing C.o.V (λ) values , we found out that BinLPT led to lower execution time variance than `guided` and `dynamic`, thus showing that our strategy delivers better load balancing. Moreover, we noted that BinLPT presented a stepped behavior, with steps occurring whenever the number of working threads perfectly divides the maximum processing capacity of the platform. This finding thus evidences the scenario where BinLPT performs better.

6.4 | Elastodynamics Simulator

Figures 13 and 14 depict the minimum, mean and maximum times to compute the iterations of the two loops in the Elastodynamics Simulator, when using the `dynamic` and BinLPT loop schedulers, respectively. Note that minimums and maximums are plotted as vertical bars in lighter colors, and the means as points in darker colors. The red bars and points correspond to the minimum/maximum/mean times taken by the simulator to compute the local problems, whereas the blue ones correspond to the minimum/maximum/mean times taken by the simulator for post-processing. Moreover, to ease the analysis we plotted results at each 10 simulation time-steps. Simulations with `guided` unveiled similar performance to those with the `dynamic` scheduler, and thus they were omitted due to space limitation. These plots concern the compilation of values obtained from all MPI tasks across 2000 simulation time-steps. Recall that in order to apply BinLPT, in the first 100 simulation time-steps we profiled the execution when running with `dynamic`, and in the remainder 1900 time-steps we applied BinLPT with its multiloop feature. This is the reason why the red and blue bars in Figure 14 are higher in its leftmost part, corresponding to the first 100 simulation time-steps (when the execution of the loops is not using BinLPT).

The results unveil significant differences when employing the `dynamic` and BinLPT schedulers in the elastodynamics simulations. When analyzing the parallel loop in which post-processing is applied for computing the final numerical solution, the following analysis holds. When using BinLPT instead of `dynamic`, the mean execution time of a simulation time-step reduces from 271.15 to 246.44 ms; and the mean C.o.V decreases from

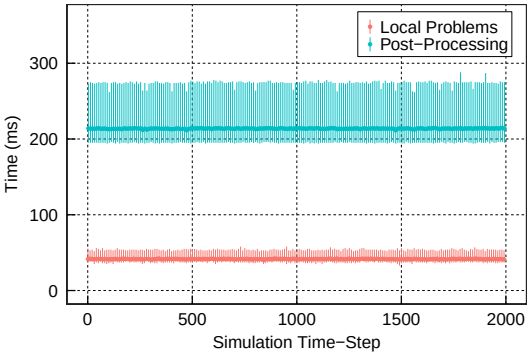


FIGURE 13 Execution profiling while using `dynamic` (original).

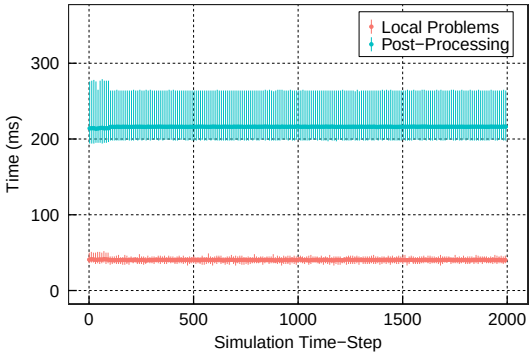


FIGURE 14 Execution profiling while using BinLPT (original).

11.58% to 7.67%. It is worth mentioning that even though the blue points corresponding to the mean times of the post-processing loop in the remainder 1900 time-steps in Figure 14 are higher than the ones in Figure 13, the top of the blue bars corresponding to the maximum values in Figure 14 are lower than the ones in Figure 13. Since such loops need to be completely finished for the simulator to proceed with the next time-step, the worst iteration (the one with the maximum value) will block the whole simulator. On the other hand, for the loop parallel loop in which local problems are solved, the use of BinLPT implies in a drop of 41.50 to 40.67 ms and 8.50% to 7.40% for the aforementioned metrics. Overall, our strategy yielded to 9.11% and 5.01% of performance gains over `dynamic` in the execution of local problem solving and post-processing loops.

6.5 | Summary of Results

Overall, we can summarize our findings as follows:

Simulation The workload analysis unveiled that BinLPT better balances the underlying workload of the target parallel loop, regardless the type of the workload. The highest gains over the baseline strategies were observed for Exponential-generated workloads, due to its inherently high irregular pattern. In this scenario, BinLPT delivered up to 1.27× superior load balancing, on average. Furthermore, BinLPT showed up to be the least sensitive strategy in what concerns the way in which the underlying workload of the target parallel loop is shuffled. The reason for this is based on the fact that, in contrast to baseline strategies, BinLPT accounts for the workload when scheduling chunks to the threads.

Synthetic Kernel Benchmarking The overhead scaling analysis pointed out that the number of chunks produced by BinLPT is far smaller than the ones produced by `guided`, `dynamic` and HSS loop schedulers. The kernel complexity benchmark uncovered that BinLPT performance gains over baseline strategies are strengthened by more complex kernels. The greatest Performance (ρ) gap was observed for a Quadratic kernel when BinLPT performed up to 1.22× better than the best baseline strategy (HSS). Finally, the loop size scaling assessment uncovered that BinLPT surpasses baseline strategies in performance when the Load Imbalance Amortization Principle is not strongly present. The highest Performance (ρ) observed for BinLPT was for the scenario with 1536 iterations Exponential-generated workload, where it delivered at least 21.25% superior Performance (ρ) than the best baseline strategy (`dynamic`).

Application Kernel Benchmarking Strong scaling analysis uncovered that BinLPT achieves better performance than baseline strategies in all three kernels considered in this paper. Significant Performance (ρ) gains were observed for Exponential-generated workloads and when the full processing capacity of the experimental platform was used. In the SMV, MST and LavaMD kernels we observed up to 15.78%, 25.0% and 64.91% of performance improvement, respectively, in contrast to the best baseline strategy in each scenario. In addition, our experimental results unveiled further support to the conclusions that we previously drew with the Synthetic Kernel Benchmarking experiments. Conversely to the Load Imbalance Amortization Principle[4], the more threads execute a parallel loop the stronger is load imbalance, once that less iterations are to be actually assigned and thus bad/good scheduling decisions become more critical to performance. Strong scaling experiments with application kernels unveiled that BinLPT delivers scalable Performance (ρ) in this pathological scenario.

Elastodynamics Simulator Execution profiling unveiled that BinLPT may deliver up to 9.11% better performance than the OpenMP's `dynamic` loop scheduler. More importantly, C.o.V results pointed out that BinLPT may balance the workload of a target parallel loop in up to 33.76% better. In a production environment, a better usage of the of the underlying platform implies on significant savings in terms of financial costs.

7 | CONCLUSIONS

To deliver high performance to large-scale engineering and scientific applications, particular intricacies of the application and the underlying platform should be considered, so that tailored techniques can be employed. In this context, execution irregularity is an important feature that should be taken into account when scheduling tasks in a parallel application. While in regular applications, a naive static strategy that assigns an even number of tasks to the working threads may lead to an optimal solution, in highly irregular applications smarter heuristics are required to enable high performance. Indeed, task scheduling in multiprocessors plays an important role in HPC, and thus it is a recurring subject of research. For instance, this problem emerges when scheduling independent iterations of parallel loops in shared-memory-based applications. In this context, the problem is referred to as LSP, and it is reduced to the assignment of independent loop iterations such that their load is evenly distributed, and thus execution time reduced, and the scheduling overhead is minimized.

Several loop scheduling strategies were proposed to address the aforementioned problem, and they mainly rely on on-demand scheduling and chunk-size tuning. When coupled together these techniques may deliver reasonable performance to a wide range of scenarios: the former dynamically handles load imbalance and runtime variations, whereas the latter mitigates scheduling overheads and enables iteration affinity exploitation. However, on-demand scheduling and chunk-size tuning do not consider any knowledge about the underlying workload of the target parallel loop, and thus scheduling strategies built upon them naturally turn out to be suboptimal. To address this limitation, workload-aware strategies were introduced. These strategies rely on some workload knowledge to adaptively fine-tune chunk sizes, and thus further amortize load imbalance and deliver superior performance. Nevertheless, existing workload-aware loop scheduling strategies fall short on several points that still should be addressed, such as workload-estimation, chunk scheduling, and integration with applications.

To overcome the aforementioned weaknesses, we proposed a novel workload-aware loop scheduling strategy called BinLPT and we presented an integration of this strategy into the the OpenMP runtime system of GCC. Our strategy is based on three features for delivering superior performance than existing loop scheduling strategies. First, it relies on some user-supplied estimation of the workload of the target irregular loop. Such estimation may be derived either from the problem structure or through online/offline profiling, thus enabling maximum flexibility. Second, BinLPT uses a greedy bin packing heuristic to adaptively partition the iteration space in several chunks. The maximum number of chunks that may be produced is a parameter of our strategy, and it may be fine-tuned to better meet the characteristics of the irregular parallel loop. Third, it schedules chunks of iterations using a hybrid scheme based on the LPT rule and on-demand scheduling, thereby ensuring that load imbalance and runtime variations are handled in the best possible known way.

This work extends our previous one and delivers two new contributions to the state-of-the-art. First, we introduce multiloop support to BinLPT. This new functionality enables the HPC engineer to reuse workload estimations across different parallel loops as well as to have different workload estimations for each one. Based on this feature, we integrated BinLPT into an Elastodynamics Simulator that is employed by the oil and gas industry for identifying extraction hotspots on a seabed. In addition, aiming at promoting a discussion on how BinLPT could be employed on other scenarios, we discuss how we actually integrated BinLPT into the aforementioned real-world application as well as we present its performance analysis on Santos Dumont Supercomputer. Second, we present a rich performance evaluation of BinLPT using three different techniques: (i) simulation; (ii) synthetic kernel benchmarking; and (iii) application kernel benchmarking. We employed simulations to assess the upper bound performances of our loop scheduling strategy under a great variety of workloads. We used the synthetic kernel benchmarking to uncover the potentials of BinLPT in a realistic environment, under different algorithmic configurations of the target parallel loop. Finally, we considered application kernel benchmarking to analyze the effective performance of our strategy in practice. In this third assessment, we selected three different application kernels to study. These kernels present great importance to the scientific community, span over different domains and feature an irregular computation: Sparse Matrix by Vector Multiplication (SMV); Minimum Spanning Tree Clustering (MST); and N-Body Simulations (LavaMD). To deliver a rich comparative analysis, we considered three baseline strategies and we run all experiments in a large-scale NUMA machine.

Our performance analysis with the Elastodynamics Simulator unveiled significant execution time differences between `dynamic` and BinLPT. Our loop scheduler took in average 9.11% and 5.01% less cycles than the baseline strategy for computing local problems and carrying out the post-processing, respectively. Furthermore, we observed that BinLPT conducted to smaller variances on running times, thereby further strengthening our hypothesis that BinLPT better balances the workload of a parallel loop among its working threads. On the other hand, our experimental results consistently unveiled that BinLPT delivers superior performance than the baseline strategies, whether the assessment was made via simulation, synthetic kernel benchmarking or application kernel benchmarking. In the simulations, we observed that BinLPT better balances the underlying workload of the target parallel loop, regardless the characteristics of the workload. BinLPT achieved the best performance on Exponential-generated workloads, delivering $1.27\times$ superior load balancing than baseline strategies. In the synthetic kernel benchmarking, we observed that the potentials of BinLPT are further strengthened when the algorithmic complexity of the target parallel loop increases. Furthermore, BinLPT presented superior performance when scaling the size of the underlying workload, and it also stood out as the loop scheduling strategy with smaller scheduling overheads. Finally, application kernel benchmarking experiments uncovered that BinLPT features superior stronger scaling capabilities

than baseline strategies for Exponential-generated workloads. In the SMV, MST and LavaMD kernels we noted up to 15.78%, 25.0% and 64.91% of performance improvement, respectively, in contrast to the best baseline strategy in each scenario.

The significant performance gains that we observed with the Elastodynamics Simulator, synthetic kernel and application kernels motive us to push even further our enhancements on BinLPT. In future work, we intend to improve our implementation of BinLPT to provide additional extensions and features, such as a profiling module and a built-in regression-based workload estimator, for further assisting engineers to integrate BinLPT into their applications. Furthermore, we intend to port BinLPT to other OpenMP runtime environments and also integrate our scheduler to other parallel programming frameworks such as TBB, Cilk and Charm++.

ACKNOWLEDGEMENTS

This work was supported by FAPEMIG, FAPESC, CNPq, CNRS and INRIA. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. The authors acknowledge LNCC/MCTI, for providing access to the SDumont Supercomputer and thank Arnaud Legrand (Université Grenoble Alpes) for his help with the evaluation of BinLPT.

References

1. Graham Ronald. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*. 1969;17(2):416–429.
2. Fang Zhixi, Tang Peiyi, Yew Pen-Chung, Zhu Chuan-Qi. Dynamic Processor Self-Scheduling for General Parallel Nested Loops. In: IEEE Transactions on Computers, vol. 39: :919–929; 1990.
3. Polychronopoulos Constantine, Kuck David. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*. 1987;C-36(12):1425–1439.
4. Hummel Susan, Schonberg Edith, Flynn Lawrence. Factoring: A Method for Scheduling Parallel Loops. *Communications of the ACM*. 1992;35(8):90–101.
5. Tzen Ten, Ni Lionel. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Transactions on Parallel and Distributed Systems*. 1993;4(1):87–98.
6. Markatos Evangelos, Le Blanc Thomas. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*. 1994;5(4):379–400.
7. Hurson A.R., Lim Joford T., Kavi Krishna M., Lee Ben. Parallelization of DOALL and DOACROSS Loops - A Survey. In: Advances in Computers, vol. 45: Elsevier 1997 (pp. 53 - 103).
8. Penmatsa S., Chronopoulos A. T., Karonis N. T., Toonen B. R.. Implementation of Distributed Loop Scheduling Schemes on the TeraGrid. In: :1-8; 2007.
9. Han Yiming, Chronopoulos Anthony T.. Scalable Loop Self-Scheduling Schemes for Large-Scale Clusters and Cloud Systems. *International Journal of Parallel Programming*. 2017;45(3):595–611.
10. Balasubramaniam M., Sukhija N., Ciorba F. M., Banicescu I., Srivastava S.. Towards the Scalability of Dynamic Loop Scheduling Techniques via Discrete Event Simulation. In: :1343-1351; 2012.
11. Penna Pedro Henrique, Castro Márcio, Freitas Henrique, Broquedis François, Méhaut Jean-François. Design Methodology for Workload-Aware Loop Scheduling Strategies Based on Genetic Algorithm and Simulation. *Concurrency and Computation: Practice and Experience*. 2016;29(22):1–18.
12. Bull J. Mark. Feedback Guided Dynamic Loop Scheduling: Algorithms and Experiments. In: International European Conference on Parallel and Distributed Computing (Euro-Par):377–382; 1998; Southampton, UK.
13. Banicescu Ioana. On the Scalability of Dynamic Scheduling Scientific Applications with Adaptive Weighted Factoring. *Journal of Cluster Computing (JCS)*. 2003;6(3):215–226.

14. Kejariwal Arun, Nicolau Alexandru, Polychronopoulos Constantine. History-Aware Self-Scheduling. In: International Conference on Parallel Processing (ICPP):185–192; 2006; Columbus, USA.

15. Wang Yizhuo, Ji Weixing, Shi Feng, Zuo Qi, Deng Ning. Knowledge-Based Adaptive Self-Scheduling. In: International Conference on Network and Parallel Computing (NPC):22–32; 2012; Gwangju, Korea.

16. Penna Pedro Henrique, Castro Márcio, Plentz Patricia, Freitas Henrique, Broquedis François, Méhaut Jean-François. BinLPT: A Novel Workload-Aware Loop Scheduler for Irregular Parallel Loops. In: Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD):220–231; 2017; Campinas, Brazil.

17. Donfack S., Grigori L., Gropp W. D., Kale V.. Hybrid Static/dynamic Scheduling for Already Optimized Dense Matrix Factorization. In: :496-507; 2012.

18. Kale V, Randles AP, Kale V, Gropp WD. Locality-optimized scheduling for improved load balancing on SMPs. In: :1063–1074ACM; 2014.

19. Chen Quan, Chen Yawen, Huang Zhiyi, Guo Minyi. WATS: Workload-Aware Task Scheduling in Asymmetric Multi-Core Architectures. In: International Parallel and Distributed Processing Symposium (IPDPS):249–260; 2012; Shanghai, China.

20. Kruskal Clyde, Weiss Alan. Allocating Independent Subtasks on Parallel Processors (TSE). *IEEE Transactions on Software Engineering.* 1985;SE-11(10):1001–1016.

21. Thoman Peter, Jordan Herbert, Pellegrini Simone, Fahringer Thomas. Automatic OpenMP loop scheduling: A combined compiler and runtime approach. In: Lecture Notes in Computer Science, vol. 7312: 2012 (pp. 88–101).

22. Cariño Ricolindo, Banicescu Ioana. Dynamic Load Balancing with Adaptive Factoring Methods in Scientific Applications. *Journal of Supercomputing.* 2008;44(1):41–63.

23. Asanović Krste, Bodik Ras, Catanzaro Bryan Christopher, et al. *The Landscape of Parallel Computing Research: A View from Berkeley.* UCB/EECS-2006-183: EECS Department, University of California, Berkeley; 2006.

24. Buluç Aydin, et al . Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks. In: Annual Symposium on Parallelism in Algorithms and Architectures (SPAA):233–244; 2009; Calgary, Canada.

25. Wu Chao-Chin, Lai Lien-Fu, Yang Chao-Tung, Chiu Po-Hsun. Using Hybrid MPI and OpenMP Programming to Optimize Communications in Parallel Loop Self-Scheduling Schemes for Multicore PC Clusters. *Journal of Supercomputing.* 2009;60(1):31–61.

26. Wu Chao-Chin, Yang Chao-Tung, Lai Kuan Chou, Chiu Po-Hsun. Designing Parallel Loop Self-Scheduling Schemes Using the Hybrid MPI and OpenMP Programming Model for Multi-Core Grid Systems. *Journal of Supercomputing.* 2012;59(1):42–60.

27. Bailey D. H., Barszcz E., Barton J. T., et al. The NAS Parallel Benchmarks - Summary and Preliminary Results. In: Supercomputing '91:158–165ACM; 1991; New York, NY, USA.

28. Grygorash Oleksandr, Zhou Yan, Jorgensen Zach. Minimum Spanning Tree Based Clustering Algorithms. In: IEEE International Conference on Tools with Artificial Intelligence (ICTAI):73–81; 2006; Arlington, USA.

29. Springel Volker, White Simon D M, Jenkins Adrian, et al. Simulations of the Formation, Evolution and Clustering of Galaxies and Quasars. *Nature.* 2005;435(7042):629–636.

30. Che S., Boyer M., Meng J., et al. Rodinia: A benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization (IISWC):44-54; 2009.

31. Araya R., Harder C., Paredes D., Valentin F.. Multiscale Hybrid-Mixed Method. *SIAM Journal of Numerical Analysis.* 2013;51(6):3505-3531.

32. Gomes Antonio T. A., Pereira Weslley S., Souto Roberto P., Valentin F.. A Multiscale Hybrid-Mixed Method for the Elastodynamic Model with Rough Coefficients. In: Ibero-Latin American Congress on Computational Methods in Engineering (CILAMCE); 2017; Florianópolis, Brazil.

33. Srivastava Srishti, Malone Brandon, Sukhija Nitin, Banicescu Ioana, Ciorba Florina. Predicting the Flexibility of Dynamic Loop Scheduling Using an Artificial Neural Network. In: International Symposium on Parallel and Distributed Computing (ISPDC):3–10; 2013; Bucharest, Romania.

34. Penna Pedro H., Inacio Eduardo C., Castro Márcio, et al. Assessing the Performance of the SRR Loop Scheduler with Irregular Workloads. *Procedia Computer Science*. 2017;108:255 - 264. International Conference on Computational Science (ICCS).