BinLPT: A Novel Workload-Aware Loop Scheduler for Irregular Parallel Loops

Pedro Henrique Penna[1,2,3], **Márcio Castro**[1], **Patricia Plentz**[1], **Henrique C. Freitas**[2], **François Broquedis**[3], **Jean-François Méhaut**[3]

[1] Universidade Federal de Santa Catarina, Florianópolis, Brazil
[2]Pontifícia Universidade Católica de Minas Gerais, Belo Horizonte, Brazil
[3]Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

pedro.penna@posgrad.ufsc.br
{marcio.castro, patricia.plentz}@ufsc.br, cota@pucminas.br
{francois.broquedis,jean-francois.mehaut}@univ-grenoble-alpes.fr

Abstract. *Workload-aware loop schedulers were introduced to deliver better performance than classical strategies, but they present limitations on workload estimation, chunk scheduling and integrability with applications. Targeting these challenges, in this work we propose a novel workload-aware loop scheduler that is called BinLPT and it is based on three features. First, it relies on some user-supplied estimation of the workload of the target parallel loop. Second, BinLPT uses a greedy bin packing heuristic to adaptively partition the iteration space in several chunks. The maximum number of chunks to be produced is a parameter that may be fine-tuned. Third, it schedules chunks of iterations using a hybrid scheme based on the LPT rule and on-demand scheduling. We integrated BinLPT in OpenMP, and we evaluated its performance in a large-scale NUMA machine using a synthetic kernel and 3D N-Body Simulations. Our results revealed that BinLPT improves performance over OpenMP's strategies by up to 45.13% and 37.15% in the synthetic and application kernels, respectively.*

1. Introduction

Evenly distributing the workload among the working threads of an irregular application is a NP-Hard minimization problem known as the Multiprocessor Scheduling Problem. This is a challenge to both academic and industry communities, and it is a recurring subject of research in High Performance Computing (HPC). In shared memory parallel applications, this problem emerges when scheduling iterations of parallel loops [Polychronopoulos and Kuck 1987]. In this scenario, the problem is referred as the Loop Scheduling Problem (LSP), and it can be reduced to the assignment of independent loop iterations such that their load is evenly distributed and the scheduling overhead is minimized.

Several loop scheduling strategies have been proposed to address the previous problem [Hurson et al. 1997], and they mainly rely on two techniques. In the first, called on-demand scheduling, iterations are scheduled to threads on-the-fly at runtime, so that both 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) iteration affinity is exploited. When coupled together, on-demand scheduling and chunk-size tuning may indeed deliver reasonable performance to a wide

range of scenarios. Nevertheless, these techniques 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 [Penna et al. 2016]. To address this weakness, workload-aware strategies were introduced [Banicescu and Velusamy 2001, Kejariwal et al. 2006, Wang et al. 2012]. These strategies rely on some knowledge about the workload to adaptively fine-tune chunk sizes to further amortize load imbalance, and thus deliver superior performance. Although these strategies present better performance gains than workload-unaware strategies (or blind strategies), existing workload-aware loop scheduling strategies still face some drawbacks that should be addressed.

First, these strategies rely on profiling and statistical regression techniques, and thus are inherently designed to well-behaved workloads. To tackle irregular loops, whose workload varies drastically, some alternatives for estimating the workload on-the-fly are necessary. Usually, the loop scheduling strategy itself and the workload-estimation technique should be loosely coupled. This way, HPC engineers may plug into their solutions the workload-estimator that best fits their needs. However, existing knowledge-based strategies do not provide this flexibility. Moreover, existing workload-aware loop scheduling strategies fail to apply their knowledge about the underlying workload of the target irregular loop when scheduling chunks of iterations. They only rely on-demand scheduling technique, which leads to scalability problems [Bull 1998]. Finally, no workload-aware strategy is integrated in a publicly available parallel Application Programming Interface (API) or library, since such integration is usually not trivial [Banicescu 2003].

In this context, the main goal of this work is to propose a novel workload-aware loop scheduling strategy for irregular parallel loops in which iterations are independent from one another. This new strategy overcomes the aforementioned weaknesses in workload prediction and chunk-scheduling and was integrated in a widely-used API for parallel programming. In summary, this work delivers the following main contributions to the state-of-the-art:

- *A novel workload-aware loop scheduling strategy entitled BinLPT*. To enable superior performance and flexibility, our strategy is based on three features: (i) a user-supplied estimation about the workload of the target irregular loop; (ii) a greedy bin packing heuristic to adaptively partition the iteration space into several chunks; and (iii) a hybrid scheme based on the Longest Processing Time (LPT) rule and on-demand scheduling [Graham 1969].
- *An integration of our novel workload-aware loop scheduling strategy into the OpenMP runtime system of GCC*. Our implementation is open-source and publicly available for download, thereby enabling any parallel application that relies on this programming abstraction to seamlessly use our strategy.

To evaluate BinLPT, we carried out a performance analysis using a synthetic kernel and a 3D N-Body Simulations application kernel. We ran experiments on a large-scale Non-Uniform Memory Access (NUMA) machine and we contrasted the performance of BinLPT against the default strategies that are shipped with OpenMP. 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 workload-aware loop scheduling strategy proposed in this work. In Section 4, we present our evaluation methodology. In Section 5, we discuss the experimental results. In Section 6, we contrast related work with ours. In Section 7, we draw the conclusions of our work and discuss some future works.

2. Background

In this section we introduce the background on which this work relies. First, we present the LSP, and then we discuss the classical loop schedulers.

2.1. The Loop Scheduling Problem

The LSP is an instance of the NP-Hard minimization problem for multiprocessor scheduling and it down to partitioning $\widehat{x}$ into n non-overlapping chunks and assigning disjoint sets of these n chunks to the p threads; such that the following are minimized: (i) the number of n chunks that are used to partition the loop iteration space $\widehat{x}$; and the maximum load imbalance between any pair of threads in p. This formulation shows the relation of the four core variables of the LSP: (i) the loop iteration space $\widehat{x}$; (ii) the load of iterations w_k; (iii) the number of chunks n in which $\widehat{x}$ will be partitioned; and (iv) the number of threads p that will process the n chunks in parallel.

Additional variables may also play an important role in a real-world context. For instance, the *scheduling overhead* is an important concern for strategies that assign chunks of iterations to idle threads on-the-fly. If contention in synchronization structures is costly, the irregular parallel loop may face scalability issues [Fang et al. 1990]. The performance of memory-intensive irregular loops can also be severely impacted by data locality, so *memory affinity* can be exploited by loop schedulers when through the use of large chunks [Markatos and Le Blanc 1994].

2.2. 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 [Kejariwal et al. 2006]. Static scheduling strategies introduce no runtime overhead, but (i) they are only possible on parallel loops which can have their bounds somehow determined at compile time, and (ii) they are suitable only for parallel loops which feature a compile-time predictable workload. In contrast, runtime strategies are employed to address parallel loops that either do not meet the aforementioned compile-time requirement, or perform computation on a workload that is known only at runtime. In this work, we address the problem of scheduling irregular parallel loops in which the workload is known only at runtime, and in the following paragraphs we discuss the classical scheduling strategies on this scenario.

Pure Dynamic Scheduling (PDS) assigns iterations to threads in unit-sized chunks and on-demand. Whenever a thread becomes idle, an iteration is assigned to it. This strategy achieves good load balancing but at the price of a possibly high runtime overhead.

Chunk Self-Scheduling (CSS) works like PDS, but instead of assigning iterations one by one, it assigns iterations in equally-sized chunks [Fang et al. 1990]. Small chunk sizes deliver good load balancing, but they likely introduce prohibitive runtime overheads. In contrast, large chunk sizes avoid this problem, but may increase load imbalance. When the chunk size is fine-tuned, near-optimum load balancing is achieved [Balasubramaniam et al. 2012], and when it equals to one, this scheduling strategy degenerates to PDS.

Guided Self-Scheduling (GSS) also assigns chunks of iterations to threads on demand, but it dynamically changes their size at runtime (the size of the next chunk is given by the number of remaining iterations divided by the number of threads) [Polychronopoulos and Kuck 1987]. The idea of having a decreasing chunk size is to offer a compromise between achieving good load balancing while reducing runtime overhead.

Factoring Self-Scheduling (FSS) works similarly to GSS, but it differs in the way that chunk sizes are determined [Hummel et al. 1992]. To address the scenarios in which GSS does not perform so well, FSS computes the next chunk size by dividing a subset 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.

Among the aforementioned loop scheduling strategies, OpenMP offers builtin support for FSS and CSS. The OpenMP community pragmatically refers to them as Guided and Dynamic, respectively. Therefore, we will refer to these strategies using the latter notation, unless otherwise stated.

3. The BinLPT Loop Scheduler

In this section, we present our novel workload-aware loop scheduling strategy. First, we discuss the internals of BinLPT, and then we detail our strategy algorithmically. We implemented BinLPT in libGOMP and we made the enhanced version of this runtime system publicly available[1] under the GPL v3 License. Therefore, any parallel application that is built on top of OpenMP may seamlessly use our scheduling strategy.

3.1. Strategy Internals

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 indeed the key features that enable the superior performance of BinLPT.

In the *chunk partitioning* phase, BinLPT splits 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 and iteration affinity may be exploited efficiently. Indeed, this sub-problem could be optimally solved in pseudo-polynomial time using a dynamic programming algorithm for the Linear Partition Problem. Nevertheless, since loop ranges may grow asymptotically, the overhead incurred by this algorithm makes its use prohibitive. Therefore, BinLPT relies on a workload-aware adaptive technique that takes as input a user-supplied threshold k and works as follows. First it computes the average load ω_{avg} for a chunk based on the workload information and k. Next, it uses a greedy bin packing heuristic that bundles into a single chunk the maximum number of iterations whose overall load does not exceed ω_{avg}.

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 works as follows. Initially, chunks are statically scheduled to threads using the LPT rule: which assigns the heaviest chunks to the least overloaded threads, and then iteratively assigns lighter chunks to more heavily loaded threads. Next, threads are unblocked and start computing. Then, whenever a thread finishes computing all its chunks, it steals a chunk from other thread that still has work left. This simple scheme optimally handles load imbalance created by both predictable and unpredictable phenomena. Static scheduling based on LPT ensures a $4/3$-approximation scheduling solution to load imbalance incurred by the workload. On the other hand, on-demand scheduling ensures that unpredictable phenomena, such as communication latencies, external load interference, and poor workload estimation, are optimally tackled in a 2-approximative fashion [Graham 1969].

[1] `www.github.com/lapesd/libgomp`

Algorithm 1 BinLPT loop scheduling strategy.

1: **function** BINLPT(A, k, n)
2: $C \leftarrow$ COMPUTE-CHUNKS(A, k)
3: SORT(C, descending order)
4: **for** i **from** 0 to n **do**
5: $T_i \leftarrow 0$
6: **for** i **from** 0 **to** $\mid C \mid$ **do**
7: $T_j \leftarrow \min T$
8: $P_{T_j} \leftarrow P_{T_j} \cup \{\widehat{c_i}\}$
9: $T_j \leftarrow T_j + \omega(\widehat{c_i})$
10: **return** P
11: **function** COMPUTE-CHUNKS(A, k)
12: $j \leftarrow 0$
13: $C \leftarrow$ empty multiset
14: $\widehat{c_0} \leftarrow$ empty sequence
15: $\omega_{\text{avg}} \leftarrow \dfrac{\sum_{i_j \in A} w_j}{k}$
16: **for** i **from** 0 to $\mid A \mid$ **do**
17: $\widehat{c_j} \leftarrow (\widehat{c_j}, A_i)$
18: **if** $\omega(\widehat{c_j}) > \omega_{\text{avg}}$ **then**
19: $C \leftarrow C \cup \{\widehat{c_j}\}$
20: $j \leftarrow j + 1$
21: **return** C

3.2. Strategy Design

The BinLPT loop scheduling strategy is outlined in Algorithm 1. In the pictured notation, means that iteration A_i is added to $\widehat{c_j}$. 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 (*i.e.* P_j is a set containing all chunks assigned to thread j). The algorithm starts by computing chunks according to the greedy bin packing heuristic detailed in the previous section (line 13). Then, it sorts the produced chunks according to their loads (line 14). Next, chunks are statically scheduled following the LPT rule (lines 15 to 20). Later, during the execution, whenever a thread becomes idle, it steals chunks from other threads.

4. Evaluation Methodology

To evaluate BinLPT, we employed a synthetic kernel and an application kernel. The former was used to assess the upper bound performance of our strategy, whereas the latter was employed to enable such analysis in a realistic scenario.

The synthetic kernel performs embarrassingly parallel computations on private variables as proposed in [Bull 1998], and thus it benchmarks the load balancing performance of a scheduler. The kernel takes as input four parameters: the iteration space size, the input workload w, the scheduling strategy, and the number of working threads. The application kernel performs N-Body Simulations and it was chosen for two main reasons. First, it has great importance to the scientific community as a whole, since it finds application on different domains, such as Computation Fluid Dynamics and Molecular Dynamics [Springel et al. 2005]. Second, it is a typical irregular kernel that is frequently studied within the context of loop scheduling [Banicescu 2003, Wang et al. 2012]. The N-Body Simulations kernel that we considered (code-named LavaMD) was extracted from the Rodinia Benchmarks Suite [Che et al. 2009].

Table 1 summarizes the parameters we used in each set of experiments. The workloads are frequently studied by related works, and they were generated using the tool proposed by [Penna et al. 2016]. Problem sizes were chosen so as to reflect the full processing capacity of the experimental platform. Baseline strategies were selected to be

Table 1. Parameters for experiments.

Parameters	Synthetic Kernel	Application Kernel
Workload PDF	Exponential, Gamma and Gaussian	Exponential, Gamma and Gaussian
Loop Size	{384, 768, 1536, 3072, 6144}	$11 \times 11 \times 11$
Chunk Size	Guided {1}, Dynamic {1,2}, BinLPT {288, 384}	Guided {1,2,3}, Dynamic {1,2,3}, BinLPT {384, 576, 768}

consistent with related works. Chunk sizes were selected based on earlier experiments that revealed them to be the optimal values.

For the *synthetic kernel*, we adopted a full factorial experimental design, thereby resulting in 75 different scenarios. For these experiments, we set the number of threads to 192 to reflect the full computational power of the experimental platform. For the *application kernel*, on the other hand, we adopted a fractioned experimental design, where we varied the number of threads from 24 to 192, with a constant step of 24. We considered 27 different scenarios for each experimental configuration. We carried out five replications of each configuration to account for the inherent variance of the measures. 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%.

We considered the following performance metrics[2] [Luke et al. 1998, Cariño and Banicescu 2008]: (a) *Parallel Time*, which is the overall execution time of the parallel loop; (b) *Cost*, which is the aggregate time spent to execute the parallel loop, and thus quantifies the waste of processor time; (c) *Performance*, which is the ratio of the total amount of work to the parallel time; (d) *Coefficient of Variance (C.o.V)*, which is the ratio between the standard deviation and the mean execution time of the threads; (e) *Slowdown*, which is the ratio between the execution time of the slowest thread to the fastest one.

All experiments were performed on a SGI Altix UV 2000 machine, which has 24 cache coherent NUMA nodes interconnected through SGI's 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 our experiments, hyper-threading was disabled and we used a first-touch memory allocation strategy coupled with a compact thread affinity policy to mitigate runtime NUMA effects.

5. Experimental Results

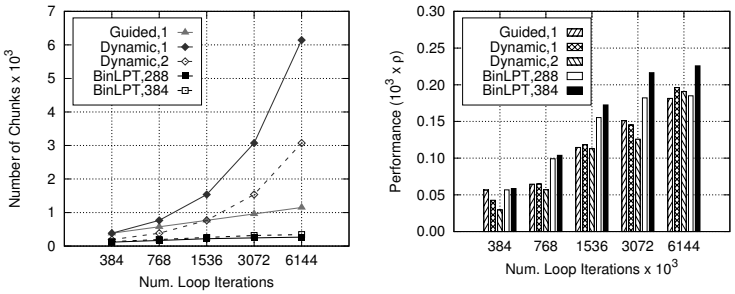
First, we show results for the synthetic kernel, and then we analyze results for the application kernel. All experimental results are publicly available for download[3].

5.1. Synthetic Kernel Results

The actual number of chunks produced by BinLPT varies according to the workload itself and it be fine-tuned by its k parameter. For instance, `BinLPT,288` produces at most $k = 288$ chunks, pragmatically. On the other hand, for the Guided and Dynamic strategies, the number of chunks that are generated depends on the iteration space $|\widehat{x}|$. For the former strategy, the number of chunks grows proportionally to $O(\log |\widehat{x}|)$, whereas for the latter

[2]We will refer hereafter to these metrics using capitalization along with their symbol.

[3]Experimental results available at: `https://doi.org/10.6084/m9.figshare.4742272`



(a) Number of chunks. (b) Performance (ρ) results.

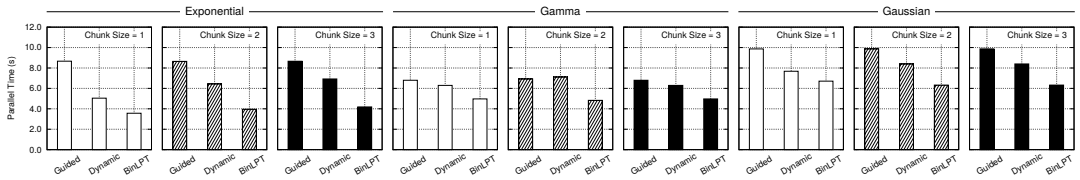
Figure 1. Synthetic kernel benchmarking results for Exponential workload.

strategy it grows with $O(|\widehat{x}|)$. In both latter strategies, the granularity of the chunk-sizes may be fine-tuned according to a parameter b. For instance, `Dynamic,1` will cause Dynamic to use unit-sized chunks ($b = 1$). On the other hand, `Guided,2` instructs Guided to generate chunks which are not smaller than 2 ($b = 2$).

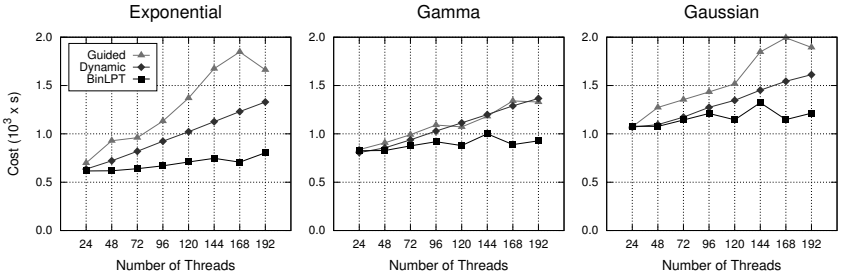
Figure 1(a) presents the number of chunks generated by each strategy for an Exponential-generated workload, when varying the size of the iteration space (*i.e.* $|\widehat{x}|$) at a constant ratio ($2\times$). We observed similar behaviors for the other workloads (*i.e.* Gamma- and Gaussian-based), and thus we omitted them due to space limitations. Overall, the results show that the number of chunks produced by BinLPT are far fewer than the ones produced by both Guided and Dynamic, regardless of the values assigned to parameters k and b. The number of chunks generated by Guided grows linearly and for Dynamic grows exponentially. Since the scheduling overhead of runtime on-demand scheduling techniques depends on the number of chunks [Cariño and Banicescu 2008], it turns out that BinLPT is the most scalable scheduling strategy for the chosen k values. Indeed, if we used larger values for k, the number of chunks produced by BinLPT could be bigger. However, in the performance analysis that follows, we reveal that the values used for k are sufficient for BinLPT to deliver superior performance.

Figure 1(b) presents Performance (ρ) results for a Exponential-generated workload, when varying the size of the iteration space (*i.e.* $|\widehat{x}|$) at the same constant ratio of $2\times$. Overall, BinLPT achieved the best results. The highest Performance (ρ) observed for `BinLPT,288` was for the scenario with 768 iterations, where it delivered 34.75% superior Performance (ρ) than the best configuration for the other two strategies (`Dynamic,1`). The highest Performance (ρ) observed for `BinLPT,384` was also in the same scenario, where it delivered 37.65% superior Performance (ρ).

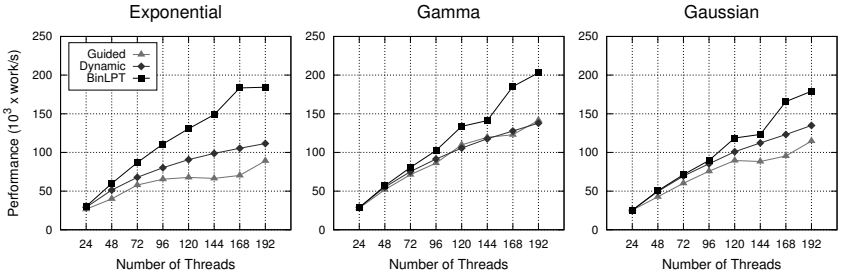
The worst Performance (ρ) observed for BinLPT was for the scenario with 6144 loop iterations and $k = 288$, where we noted 6.09% of Performance (ρ) degradation. Although `BinLPT,288` did perform slightly worse, the weight of chunks generated by BinLPT was not fine-grained enough to amortize load imbalance in this scenario. This behavior is confirmed by Figure 1(a), which shows that BinLPT generated much less chunks than the other strategies for this scenario (`BinLPT,288` generated 266 chunks, whereas `Guided,1`, `Dynamic,1` and `Dynamic,2` generated $4.33\times$, $23.09\times$ and $11.54\times$ more chunks, respectively). Thus, a fair comparison between BinLPT and the other strategies accounts for the equivalence between the maximum number of chunks k produced by our strategy and the number of chunks generated by them. Recall that `BinLPT,288` splits



(a) Parallel Time (τ) vs. chunk size when running with 192 threads.



(b) Cost (γ).



(c) Performance (ρ).

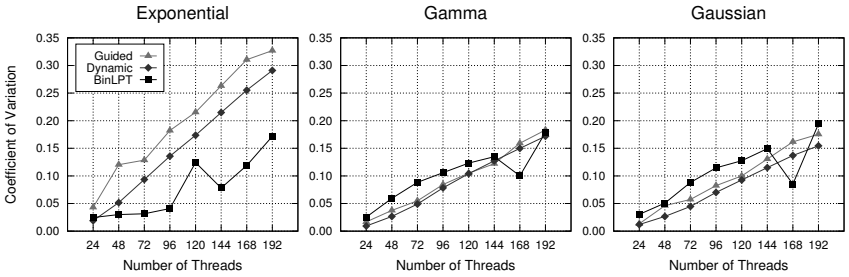
Figure 2. Experimental results for the N-Body Simulations application kernel.

the iteration space in at most 288 variable-size chunks, whereas `Dynamic,1` produces 6144 unit-sized chunks ($b = 1$). Therefore, for instance, a fair comparison would be in the scenario with 768 iterations, between `BinLPT,384` (at most 384 chunks) against `Dynamic,2` ($768/2 = 384$ chunks). In this case, BinLPT would deliver 45.13% better Performance (ρ). We observed similar results for the Gamma- and Gaussian generated workloads (they were omitted due to space limitations), in which BinLPT delivered 29.94% and 32.81% better Performance (ρ) over `dynamic,2`, respectively.

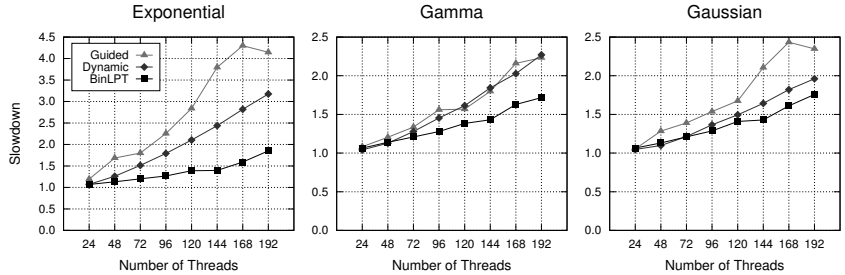
5.2. Application Kernel Results

Figure 2(a) presents Parallel Time (τ) results for the N-Body Simulations application when using 192 threads and varying both the chunk size and the workload. In this plot, the chunk size for the BinLPT means that the parameter k of our strategy was chosen so as it would lead a fair-comparison with Guided and Dynamic in each scenario. Overall, the results unveiled that BinLPT delivers better Parallel Time (τ) regardless the scenario.

Figure 2(b) presents Cost (γ) results when varying the number of threads, for a scenario with the following configurations (worst-case scenario for BinLPT): `Guided,3` `Dynamic,3` and `BinLPT,384`. When analyzing the results, we observed that BinLPT delivers near constant scalability in a large-scale NUMA machine, for all the three workloads. On the other hand, the Cost (γ) scales up quasi-linear for Guided and linear for



(a) C.o.V (γ) results.



(b) Slowdown (χ) results.

Figure 3. Experimental results for the N-Body Simulations application kernel.

Dynamic. Nevertheless, it is worthy to note that Cost (γ) results suggest that BinLPT performs better on the Exponential and Gaussian results. Indeed, we confirmed this finding with Performance (ρ) results showed in Figure 2(c). In the case of Exponential- and Gaussian-generated workloads, BinLPT delivers up to 37.15% (168 threads) and 34.45% (192 threads) better Performance (ρ), when considering Dynamic as baseline. In, contrast, for the Gamma workload, we noted 30% of improvement.

So far, we considered metrics that rely on the performance of the slowest thread executing the parallel loop. Thus, we now analyze the other two metrics that consider the performance of all threads: C.o.V (λ) and Slowdown (χ). When we analyzed C.o.V (λ), we found out that BinLPT presented lower values than Guided and Dynamic on the Exponential-generated workload; and it showed up worst C.o.V (λ) results for the other two distributions. Indeed, these results at glance suggest that BinLPT does not deliver load balancing. Nevertheless, since our strategy did present better overall Cost (γ) and Performance (ρ) results than Guided and Dynamic in all the three workloads regardless the number of threads, we drawn the following conclusion. Even though there is a great difference between the execution times of threads in BinLPT, the actual absolute difference is smaller than it is for Guided and Dynamic. We confirmed this finding by plotting histograms of execution times of threads, when running with 96 to 192 threads. However, we had to omit these plots due to space limitations. This conclusion is also pictured in Figure 3(b), which presents Slowdown (χ) results. This plot evidences that the difference between the slowest and fastest threads in BinLPT is actually smaller than in Guided and Dynamic for all thread configurations. Considering Dynamic as baseline, and when using 192 threads, gains are 45.54%, 26.32% and 12.62% for the Exponential-Gamma- and Gaussian-generated workloads, respectively.

6. Related Work

Targeting time-step applications with irregular parallel loops, Banicescu [Banicescu 2003] 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 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 [Cariño and Banicescu 2008]. 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.* [Kejariwal et al. 2006] proposed History-Aware Self-Scheduling (HSS). Unlike AWF, HSS relies on statistical information collected offline via profiling to carry out a smarter scheduling. Based on this extra knowledge, at every scheduling round, HSS computes chunk sizes in a decreasing fashion like FSS, but also considering 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.* [Wang et al. 2012] introduced Knowledge-Based Adaptive Self-Scheduling (KASS). This strategy works on 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 makes the total workload to be equally distributed to the threads, approximately. In the second phase, iterations on local work queues are scheduling with decreasing sizes likewise 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, and 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.

The workload-aware loop scheduling strategy that we proposed in this work differs from the related strategies discussed above in several points, thereby delivering contributions to the state-of-the-art. First, unlike all these strategies, BinLPT 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. Con-

sequently, 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 and KASS, which rely on offline profiling and online regression techniques. Second, even though the aforementioned strategies do use their estimations on the workload to partition the iteration space in several chunks, they lack in using this knowledge to actually schedule chunks of iterations. 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 in a 2-approximation optimally [Graham 1969]. Finally, existing strategies lack on integrability with applications. For instance, their source-code is not available for download, and their reported algorithmic description is not detailed enough to enable an in-house implementation of them. Our solution, on the other hand, is open-source and is built into GCC's OpenMP runtime.

7. Conclusions

In this work we proposed a novel workload-aware loop scheduling strategy called BinLPT. To enable superior performance and flexibility, our strategy is based on three features. 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 number of chunks to be produced is a parameter of our strategy that may be fine-tuned. Third, it schedules chunks of iterations using a hybrid scheme based on the LPT rule and on-demand scheduling.

We integrated BinLPT into OpenMP, and we evaluated its performance in a large-scale NUMA machine using a synthetic kernel and a 3D N-Body Simulations application kernel. We considered several workloads and we contrasted the performance of our strategy against other strategies available in OpenMP (Guided and Dynamic). Our results unveiled that BinLPT achieves up to 45.13% and 37.15% better performance in the synthetic and application kernels, respectively. As future work, we intend to enhance BinLPT so that it also accounts for data locality when scheduling chunks; and to extend our strategy to emerging manycore platforms which feature a distributed shared memory.

Acknowledgment

This work was supported by FAPEMIG, FAPESC, CAPES, CNPq, CNRS and INRIA.

References

Balasubramaniam, M., Sukhija, N., Ciorba, F., Banicescu, I., and Srivastava, S. (2012). Towards the Scalability of Dynamic Loop Scheduling Techniques via Discrete Event Simulation. In *International Parallel and Distributed Processing Symp. Workshops*, pages 1343–1351.

Banicescu, I. (2003). On the Scalability of Dynamic Scheduling Scientific Applications with Adaptive Weighted Factoring. *Journal of Cluster Computing*, 6(3):215–226.

Banicescu, I. and Velusamy, V. (2001). Performance of scheduling scientific applications with adaptive weighted factoring. In *International Parallel and Distributed Processing Symp.*, pages 791–801.

Bull, J. M. (1998). Feedback Guided Dynamic Loop Scheduling: Algorithms and Experiments. In *International European Conf. on Parallel and Distributed Computing*, pages 377–382.

Cariño, R. and Banicescu, I. (2008). Dynamic load balancing with adaptive factoring methods in scientific applications. *Journal of Supercomputing*, 44(1):41–63.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S. H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *International Symp. on Workload Characterization*, pages 44–54.

Fang, Z., Tang, P., Yew, P.-C., and Zhu, C.-Q. (1990). Dynamic Processor Self-Scheduling for General Parallel Nested Loops. In *IEEE Transactions on Computers*, volume 39, pages 919–929.

Graham, R. (1969). Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429.

Hummel, S., Schonberg, E., and Flynn, L. (1992). Factoring: a method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101.

Hurson, A., Lim, J., Kavi, K., and Lee, B. (1997). Parallelization of DOALL and DOACROSS Loops - A Survey. *Advances in Computers*, 45:53–103.

Kejariwal, A., Nicolau, A., and Polychronopoulos, C. (2006). History-Aware Self-Scheduling. In *International Conf. on Parallel Processing*, pages 185–192.

Luke, E. A., Banicescu, I., and Li, J. (1998). The optimal effectiveness metric for parallel application analysis. *Information Processing Letters*, 66(5):223–229.

Markatos, E. and Le Blanc, T. (1994). Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400.

Penna, P. H., Castro, M., Freitas, H., Broquedis, F., and Méhaut, J. (2016). Design Methodology for Workload-Aware Loop Scheduling Strategies Based on Genetic Algorithm and Simulation. *Concurrency and Computation: Practice and Experience*.

Polychronopoulos, C. and Kuck, D. (1987). Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439.

Springel, V., White, S. D. M., Jenkins, A., Frenk, C. S., Yoshida, N., Gao, L., Navarro, J., Thacker, R., Croton, D., Helly, J., Peacock, J. A., Cole, S., Thomas, P., Couchman, H., Evrard, A., Colberg, J., and Pearce, F. (2005). Simulations of the formation, evolution and clustering of galaxies and quasars. *Nature*, 435(7042):629–636.

Wang, Y., Ji, W., Shi, F., Zuo, Q., and Deng, N. (2012). Knowledge-Based Adaptive Self-Scheduling. In *International Conf. on Network and Parallel Computing*, number 60973010 in Lecture Notes in Computer Science, pages 22–32.