Design methodology for workload-aware loop scheduling strategies based on genetic algorithm and simulation

Pedro Henrique Penna, Márcio Castro, Henrique Cota de Freitas, François Broquedis, Jean-François Méhaut

▶ To cite this version:

Pedro Henrique Penna, Márcio Castro, Henrique Cota de Freitas, François Broquedis, Jean-François Méhaut. Design methodology for workload-aware loop scheduling strategies based on genetic algorithm and simulation. Concurrency and Computation: Practice and Experience, 2017, 29 (22), 10.1002/cpe.3933 . hal-01354028

HAL Id: hal-01354028
https://hal.science/hal-01354028

Submitted on 23 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Design Methodology for Workload-Aware Loop Scheduling Strategies Based on Genetic Algorithm and Simulation

Pedro H. Penna[1], Márcio Castro[1], Henrique C. Freitas[2], François Broquedis[3], and Jean-François Méhaut[3]

[1]Universidade Federal de Santa Catarina (UFSC)
[2]Pontifícia Universidade Católica de Minas Gerais (PUC Minas)
[3]Université Grenoble Alpes (UGA), Grenoble INP, CNRS, INRIA

Abstract

In High Performance Computing, the application's workload must be evenly balanced among threads to deliver cutting-edge performance and scalability. In OpenMP, the load balancing problem arises when scheduling loop iterations to threads. In this context, several scheduling strategies have been proposed, but they do not take into account the input workload of the application and thus turn out to be suboptimal. In this work, we introduce a design methodology to propose, study and assess the performance of workload-aware loop scheduling strategies. In this methodology, a Genetic Algorithm is employed to explore the state space solution of the problem itself and to guide the design of new loop scheduling strategies, and a simulator is used to evaluate their performance. As a proof of concept, we show how the proposed methodology was used to propose and study a new workload-aware loop scheduling strategy named Smart Round-Robin (SRR). We implemented this strategy into GCC's OpenMP runtime. We carry out several experiments to validate the simulator and to evaluate the performance of SRR. Our experimental results show that SRR may deliver up to 37.89% and 14.10% better performance than OpenMP's Dynamic loop scheduling strategy in the simulated environment and in a real-world application kernel, respectively.

1 Introduction

In High Performance Computing (HPC), parallel applications can be classified into two groups: *regular applications*, in which the amount of computation required to solve a problem depends mostly on the size of the input data; and *irregular applications*, which have this property strictly related to the

contents of the input data itself [1]. A naive implementation of the matrix multiplication algorithm is a typical example of a highly regular application. In this particular example, the number of operations is constantly proportional to the products of the rows of a matrix by the columns of another matrix, regardless the actual numbers stored in the matrices. In contrast, in a particle simulation application that uses integer sorting to organize the set of particles features an irregular behavior. The amount of computation actually depends on the input sequence, *i.e.,* the input integers themselves.

The difference between these two classes impacts directly on the design of efficient parallel applications. Indeed, regular applications are highly appreciated because their workload can be easily broken up into homogeneous tasks by simply dividing the total workload n by the t working threads. Unfortunately, this strategy is not enough for irregular applications, since it may lead to a heterogeneous set of tasks, thus causing load imbalance among the threads. This may result in a considerable performance and scalability issues, because the overall performance of the application would be bounded by the performance of the most overloaded thread.

Indeed, evenly distributing the input workload among the working threads of an irregular application is an NP-Complete minimization problem known as the Load Balancing Problem [2]. This problem presents high significance to the HPC field and it is a recurring subject of research [3, 4]. For instance, in OpenMP, an industry and academia standard Application Programming Interface (API) for parallel programming on shared-memory architectures [5], this problem emerges when scheduling iterations in parallel loops. In this context, the problem is referenced as the Loop Scheduling Problem and comes down into assigning loop iterations to threads, so as the load of iterations is evenly distributed. In this scenario, several loop scheduling strategies have been proposed to address the problem [6]. However, they do not take into account the input workload of the application and thus turn out to be suboptimal in some scenarios.

The main goal of this work is to introduce a design methodology that may be used to guide the study of new workload-aware loop scheduling strategies. This methodology relies on a Genetic Algorithm (GA) to explore the solution space of the problem and thus uncover new strategies, and a simulator to evaluate their performance. To validate this methodology, we propose a new loop scheduling strategy named Smart Round-Robin (SRR), which considers the input workload at runtime to better balance the workload among the threads of a parallel application without imposing substantial overhead. This paper extends our previous work [7] and delivers the following new contributions to the state-of-the-art:

1. A validation of the proposed simulator through the comparison of its output against the output obtained in a real platform running a synthetic benchmark. On an extensive evaluation based on simulation,

we observed that the SRR strategy may achieve up to 37.80% better performance than OpenMP's Dynamic scheduler.

2. An extensive exploration of the state space of the Loop Scheduling Problem that delivers some insights on the design of new workload-aware loop scheduling strategies. With this analysis we have uncovered that loop scheduling strategies that rely on the average input workload may output near-optimum solutions.

3. An implementation of SRR into the OpenMP's runtime library of the GNU Compiler Collection (GCC), thereby enabling any parallel application that relies on this programming abstraction to benefit from it. Our implementation is open-source and it is publicly available for download. Furthermore, we discuss how one can estimate the workload of an application so that it can effectively use SRR. In a real-world application kernel, the Integer Sort Kernel from the NAS Parallel Benchmarks (NPB), SRR has achieved up to 14.10% and 39.87% better performance than OpenMP's Dynamic and Guided strategies, respectively.

The remainder of this work is organized as follows. In Section 2, we introduce the fundamentals of the Loop Scheduling Problem. In Section 3, we discuss the related work. In Section 4, we present the design methodology that we propose in this work. Then, in Section 5, we present the SRR algorithm and detail how we implemented it into GCC's OpenMP runtime. In Section 6, we expose and discuss our experimental results. Finally, in Section 7, we present the main conclusions of this work and its future perspectives.

2 The Loop Scheduling Problem

The Loop Scheduling Problem is a NP-complete minimization problem that consists in a particular case of the Load Balancing Problem and can be formally stated as follows [2]. Let $X = \{i_1, i_2, \ldots, i_n\}$ be a set of n iterations, and $w_k \in \mathbb{N}^+$ be the workload of iteration i_k. If P_k is an arbitrary subset of X, the workload of this subset can be expressed by $W_{P_k} = \sum\limits_{i_k \in P_k} w_k$.

Given two arbitrary disjoint subsets of X, P_a and P_b, we express the load imbalance φ_{P_a,P_b} among them as the absolute difference between their workloads (Equation 1):

$$\varphi_{P_a,P_b} = |W_{P_a} - W_{P_b}| \tag{1}$$

Therefore, given an integer $k \geq 1$, the Loop Scheduling Problem comes down in partitioning X in k disjoint subsets $\{P_1, P_2, \ldots, P_k\}$ so as to minimize the maximum load imbalance φ within these subsets (Equation 2):

$$f(P_a, P_b) = \min\left(\max\left(\varphi_{P_a,P_b}\right)\right), \quad \forall P_a, P_b \in \{P_1, P_2, \ldots, P_k\} \mid P_a \neq P_b \quad (2)$$

The number of loop iterations and partitions are inherent variables to the Loop Scheduling Problem, so they cannot be left aside. However, other variables may be considered when the problem is analyzed within a real-world context, which turns the minimization function into a multi-objective one. In this new scenario, some important variables are the load of each loop iteration, the frequency in which the scheduling strategy is invoked, and the memory affinity that exists among iterations.

The load of each iteration is related to the input workload of the application, and it has great influence in the quality of the output scheduling solution. For instance, if the load of iterations follows a Uniform distribution, the load imbalance tends to be small, whereas for a Gaussian one, this feature tends to be higher. The frequency in which the loop scheduling strategy is invoked, on the other hand, is an important concern for runtime strategies, in which iterations are assigned on the fly. This variable states how many times the scheduling strategy is invoked, and thus it impacts directly on the performance of the application. Finally, the memory affinity is related with the temporal and spatial data localities that exist among loop iterations that are assigned to the same partition. When they are strongly present, the memory system is efficiently used, reducing contention in buses and other interconnection structures and thus increasing performance.

3 Related Work

Several strategies for tackling the Loop Scheduling Problem in a wide range of scenarios have been proposed. When comes to large-scale Non-Uniform Memory Access (NUMA) platforms and memory-bound irregular applications, cutting-edge scheduling strategies that are shipped with OpenMP face several scalability issues. Aiming to overcome these problems, Durand *et al.* introduced a new loop scheduler, called *Adaptative* [4]. This strategy relies on a work-stealing algorithm to dynamically adapt the granularity of work in parallel loops to find a compromise between data-access locality of the OpenMP's Static and Dynamic schedulers. They implemented this strategy in OpenMP and assessed its performance with two application kernels, (i) the K-means Clustering, which presents an irregular behavior; and (ii) the Smooth Particle Hydrodynamic, which features a regular computation. Their results pointed out that their scheduling strategy outperforms the Dynamic loop scheduler on irregular applications in about 2.4 times, while obtaining similar performance to the Static scheduler on regular applications. Other scheduling strategies that explore locality-awareness are reported in [3, 8].

In contrast to the aforementioned research efforts that rely solely on runtime information, Thoman *et al.* proposed a hybrid approach that considers compiling information [9]. They implemented this solution in the *Insieme Compiler* and runtime system, and compared its performance against the loop schedulers available on OpenMP. Their results pointed out that a hybrid scheduling strategy may lead to up to 4.51 of performance speedup over OpenMP scheduling solutions. Another work that also relies on compiling information is presented in [10].

With this emerging variety of loop scheduling strategies, the task of selecting the most adequate one for a given application becomes challenging. To address this situation, Sukhija *et al.* proposed an approach that uses supervised Machine Learning techniques to predict the most robust loop scheduling strategy for a target application/platform [11]. They showed that their proposed approach: (i) enables the selection of the most robust loop scheduling algorithm that satisfies a user-specified tolerance on the given application's performance; and (ii) offers higher guarantees regarding the performance of the application using the automatically selected loop scheduling algorithms, when compared to the performance of the same application using an empirically selected loop scheduling algorithm.

Towards a similar goal, Srivastava *et al.* proposed and evaluated an approach based on Artificial Neural Networks (ANNs) to predict the flexibility of dynamic loop scheduling strategies for heterogeneous systems [12]. They used synthetic benchmarks that simulate the behavior of scientific irregular applications to train an ANN. To model the input workload of these synthetic programs, they used three Probability Density Functions (PDFs): Gamma, Gaussian and Exponential. Their results showed that the proposed model can be used to quantify the robustness of dynamic loop scheduling strategies. When used in conjunction with other performance metrics, the robustness metric can be used to select the most robust dynamic loop scheduling strategy which also yields performance improvements.

In respect to the current efforts for evaluating existing scheduling strategies, Srivastava *et al.* proposed a methodology for analyzing the robustness of dynamic strategies [6]. To do so, they implemented eight strategies in an in-house simulator and used a synthetic benchmark to evaluate them. This artificial program has n independent iterations and the time of each of them is modeled by a Gaussian distribution. They concluded that a simulation-based methodology may be applied to assess the performance of load balancing strategies. Another work that also suggests a simulation-based methodology to carry on this analysis is reported in [13].

Our work differs from the previous ones in several points. First, in contrast to the related works that propose a simulation-based methodology for evaluating loop scheduling strategies [6, 13], we propose a methodology that not only enables the evaluation of such strategies, but also the design of new ones. To enable this, we propose the use of a state space searching

technique based on a GA to study the Loop Scheduling Problem itself. Furthermore, (i) we make our methodology publicly available; (ii) we carry out a validation of our simulator using synthetic benchmark programs; and (iii) our simulator is capable of generating synthetic workloads according to five different PDFs. Second, different than related works that target locality-aware [4, 3, 8], compiler-based [9, 10] and portfolio-based [11, 12] scheduling, in this work we focus on workload-aware scheduling.

Finally, as a proof of concept, we show how the proposed methodology assisted us on the design of a new workload-aware scheduling strategy called SRR. This strategy relies on the estimation of the load of loop iterations to better balance the load among the threads. For some applications the load of loop iterations can be determined at runtime as a function of the input workload whereas for others such estimation can be obtained from source code instrumentation or runtime profiling. In addition, in this work, (i) we present an implementation of this strategy in GCC's OpenMP runtime, (ii) we make this implementation publicly available; and (iii) we carry out an extensive evaluation of SRR using simulation and application kernel benchmarking techniques.

4 Proposed Methodology

In this section, we present the design methodology that we propose in this work. This methodology relies on a GA to explore the solution space of the problem and guide the design of new scheduling strategies, and a simulator to evaluate the performance of loop scheduling strategies. First, we introduce a discussion about the problem variables considered in this study. Then, we present our simulator and the GA, in turn.

4.1 Problem Variables

In Section 2, we discussed the Loop Scheduling Problem and the variables related to it. Some of these variables are inherent to the problem itself (*e.g.,* the load of each iteration) whereas others are related to the intricacy characteristics of the application and/or the platform (*e.g.,* memory affinity). The methodology proposed in this work considers the following variables: (i) the number of loop iterations; (ii) the number of working threads; and (iii) the PDF of task loads associated to the loop iterations. Both the number of iterations and threads are core variables of the Loop Scheduling Problem, thus they must be considered. The third variable states the load that is associated to the loop iterations. The PDF does not only affects the imbalance between the application tasks, but also the performance of the loop scheduling strategy.

The other variables that we discussed in Section 2, such as the frequency in which the scheduling strategy is invoked and the memory affinity that

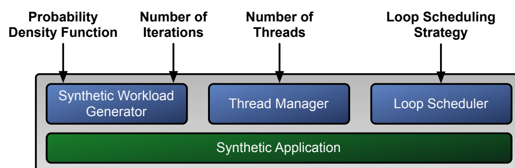


Figure 1: Architectural overview of the simulator.

exists among loop iterations, were not considered in this study. The rationale behind this is that these variables are either application- or platform-dependent, and thus they can be studied in a later step. This is indeed a future work that we intend to accomplish, and we further discuss it in Section 7.

4.2 Simulator

The simulation technique is useful for isolating the core variables of a given problem, precisely controlling the testing environment, and evaluating solutions for that particular problem with minimum efforts. Unfortunately, existing simulators for the Loop Scheduling Problem reported in Section 3 could not be used because they were not publicly available. For all these reasons, we designed and implemented an open-source simulator[1] for assessing loop scheduling strategies.

Figure 1 presents an architectural overview of this simulator, with its three main modules on the top and the synthetic application that is virtually simulated on the bottom. The Synthetic Workload Generator module takes as input parameters the number of loop iterations and the PDF associated to them, and outputs a series of iterations with the given properties. The load of an iteration expresses the time required to process it, and thus the set of all iterations represents the input workload of the synthetic application. The Thread Manager module creates and schedules the threads of the synthetic application. It takes as input parameter the number of threads to be created, and schedules these threads according to a round-robin policy. This module uses two structures to manage threads: (i) a priority queue of running threads, which is ordered by their remaining processing time; and (ii) a list of threads ready for execution. Finally, the Loop Scheduler module assigns loop iterations output by the Synthetic Workload Generator to the threads of the synthetic application, according to the scheduling policy that it takes as input parameter. It exports an interface that allows the study of both static and dynamic scheduling strategies.

This simulator is an open-source software and can be easily extended to support new features. Nevertheless, its current version is shipped with the following features:

[1]Our simulator is publicly available at `www.github.com/cart-pucminas/scheduler`

- The Synthetic Workload Generator can generate an arbitrary number of loop iterations following five different PDFs: Beta, Gamma, Gaussian, Poisson and Uniform.

- The Loop Scheduler supports two well-known strategies, which are both available in the OpenMP [5]: Static and Dynamic. The former divides the loop iterations into equal-sized chunks, and statically assigns these chunks to threads in a round-robin fashion. The latter uses an internal work queue to dynamically assign chunk-sized blocks of loop iterations to threads. In addition, as we discuss later on (Section 5), the workload-aware strategy proposed in this paper (SRR) is also implemented in the Loop Scheduler module.

- The simulator outputs a detailed trace file containing the following information about the execution of the synthetic application: (i) the number of threads; (ii) the number of iterations; (iii) the load of each iteration; and (iv) the total load assigned to each thread.

4.3 Genetic Algorithm

The simulator presented in the previous section enables us to evaluate the performance of loop scheduling strategies. Oftentimes, however, it may be difficult to know if these strategies are outputting a good scheduling (*i.e.,* evenly balanced load) for a given input workload. In this context, the state space search technique can be applied to explore the solution space of a particular instance of the Loop Scheduling Problem and to better understand some characteristics of its optimal solution. To perform this exploration, the problem is faced as a minimization problem, and a heuristic search algorithm, which is guided by the corresponding minimization objective function, is then applied (recall Equation 1).

Several heuristic search algorithms are known and have already been applied in different domains [14, 15, 16]. In this work, however, we used a GA to perform the state space exploration of the Loop Scheduling Problem. We chose this algorithm over the others due to two main reasons: first, it has shown promising results in various other contexts [17]; and second, it does not require any knowledge on how new states are generated from a given start state, which is true for our particular problem under study. In the following paragraphs we present a general GA framework, and then we discuss how we adapted it for the Loop Scheduling Problem.

GAs are based on search heuristics that mimic Darwin's Evolutionary Theory [18]. They usually encode the solution of a problem as a tuple (organism): an n-ary array (chromosome) that represents the solution itself; and a value (*fitness*) that corresponds to the evaluation of the problem's objective function for that particular solution. Based on this abstraction, the algorithm builds an initial population of organisms and evolves it with

the course of time, by applying genetic operators such as *selection*, *crossover* and *mutation*.

Algorithm 1 outlines the GA framework that we have used in this work. This algorithm takes two input parameters: the size of the population (*popsize*) and a stop condition (*stop*). It first generates an initial population (*pop*) of solutions (line 2), and then selects some organisms to form up the mating pool (lines 5 and 6). Organisms with higher *fitness* have higher probabilities of being selected, thus having a greater chance to spread their genes to future populations. Then, organisms that were selected for mating are chosen at random to form up couples and crossover with a high probability α (lines 8 to 10). In this process, some offspring (*children*) are generated by merging the chromosomes of their parents. New organisms may undergo to some mutation in their chromosomes, with a low probability β (lines 12 and 13). Finally, to form up a new population, some organisms of the old population are chosen to be replaced by the new offspring (line 15), and then a new generation starts. This process terminates when *stop* condition is met and the best solution found is returned (lines 16 and 17).

Algorithm 1 A general framework for a Genetic Algorithm.

```
 1 function GENETIC-ALGORITHM(popsize, stop)
 2     pop ← INITIAL-POPULATION(popsize)
 3     repeat
 4         newpop ← ∅, parents ← ∅
 5         for i from 1 to 2 × popsize do
 6             parents ← parents ∪ SELECTION(pop)
 7         for i from 1 to popsize do
 8             couple ← random pair (x, y) ∈ parents
 9             if high probability α then
10                 children ← CROSSOVER(couple)
11                 for all child ∈ children do
12                     if low probability β then
13                         child ← MUTATION(child)
14                     newpop ← newpop ∪ child
15         REPLACEMENT(pop, newpop)
16     until stop condition is not met
17     return BEST-ORGANISM(pop)
```

To illustrate how we adapted Algorithm 1 for the Loop Scheduling Problem, let us consider a scenario with 2 threads and 8 loop iterations, which is depicted in Figure 2. Each chromosome is encoded as an n-ary array (8, in this example) that indicates the thread/iteration assignment, and the *fitness* of each chromosome is computed accordingly to Equation 1. To build the initial population, the chromosome of each organism is randomly generated, according to a uniform probability density function (Figure 2-a). The selection operation selects pairs of organisms according to the Roulette Wheel

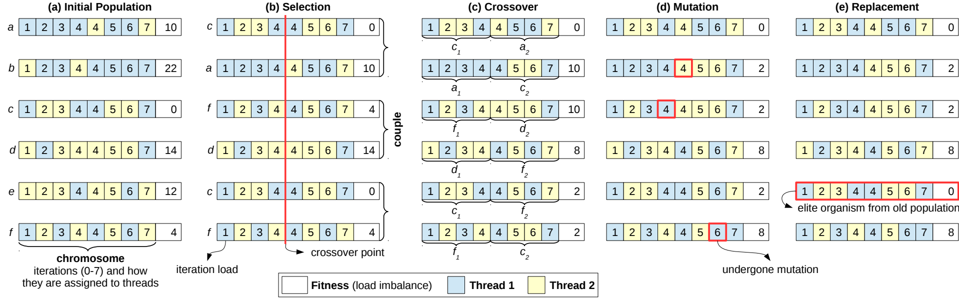


Figure 2: Overall functioning of the GA for an instance of the Loop Scheduling Problem with 2 threads and 8 loop iterations.

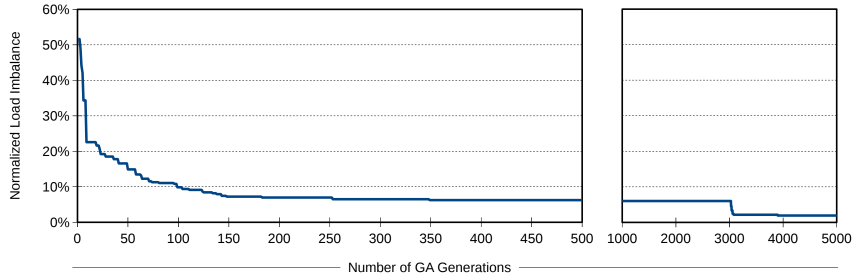


Figure 3: Genetic algorithm evolution for a Beta distribution with 96 iterations.

algorithm (Figure 2-b), in which the probability of an organism of being selected is proportional to its fitness value [18]. Then, pairs of organisms are selected at random to crossover with each other. This operation uses a single crossover point and generates two organisms (Figure 2-c). The mutation operation randomly changes parts of the chromosome (Figure 2-d). The replacement operation (a) applies the elitism technique in the old population [19], extracting from it the k best organisms, and (b) chooses randomly among the remaining organisms those that will form up the new population, considering both the remaining old population and the new organisms that were generated during the crossover operation (Figure 2-e).

In this specific example, all organisms selected for mating have generated some offspring (crossover rate of 100%), three organisms have undergone some mutations (mutation rate of 37.5%), and one organism has been directly placed in the new population (elitism rate of 1%). However, for the results that we present in this work, we have used the following parameters for the GA: a crossover rate of 80%, a mutation rate of 10%, a replacement rate of 90% and an elitism rate of 1%. Furthermore, we halted the GA's execution (*stop* condition) when the best solution found remained unchanged for $10,000$ successive generations.

These parameters were defined empirically and impact directly on the convergence of the algorithm. Figure 3 presents the evolution over the time for an instance of the problem with 96 loop iterations following a Beta PDF, showing how adequate the chosen parameters were. The best solution of the initial population has a fairly unbalanced thread/iteration assignment. However, as the number of GA generations increase, the best solution evolves towards the optimal one. After 500 generations, the best solution found by the GA presents 6% of load imbalance, and with $3,058$ generations this value decreases to 2%. Overall, we carried out the previous analysis on $3,780$ instances of the problem, varying the number of threads, loop iterations and probability density functions (Beta, Gamma, Gaussian, Poisson and Uniform). In all these scenarios, our GA presented a very similar behavior, which validates our approach.

5 The Smart Round-Robin Loop Scheduling Strategy

Existing loop scheduling strategies available in OpenMP are blind to the application workload. They schedule loop iterations regardless their load. While this approach may lead to a good performance on some applications, on others it may lead to load imbalance and thus to a performance degradation. To overcome this problem, we propose a new loop scheduling strategy that considers this information. Indeed, with this extra knowledge we could actually find several near-optimal thread/iteration assignments. However, the challenge was to come up with a strategy that would efficiently output such solution without imposing substantial overhead to the application.

To overcome this barrier, we thus considered the solutions output by the GA. For all the $3,780$ scenarios, we gathered the workload assigned to threads. Figure 4 presents these results, for 96 loop iterations and 12 threads for all the five PDFs. Grey bars indicate the workload assigned to each thread. Based on these results, the idea was to design a heuristic that would mimic such solution.

Indeed, several heuristics are possible. Nevertheless, we restricted our analysis to those that would rely on some statistical information about the input workload, more precisely on the *average*, *median* and *mode*. The coloured lines in Figure 4 point out the expected workload assigned to threads when each of these three statistical measures are considered. We observed that, regardless the PDF of the input workload, the *average* statistical measure is likely to lead to a loop scheduling similar to the one output by the GA. To confirm this finding, we considered other scenarios with different numbers of threads and loop iterations and all of them pointed out to the same conclusion. This result motivated us to design the SRR strategy, which relies on the *average* heuristic and it is further discussed in the next

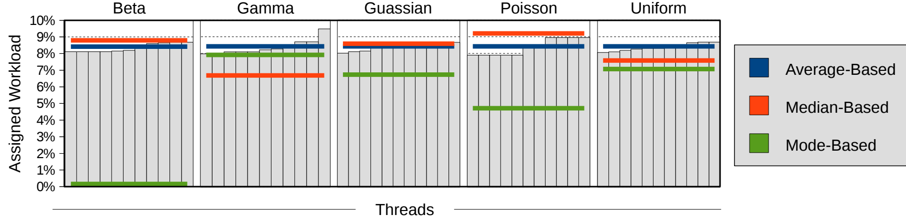


Figure 4: Workload assigned to threads and possible strategies.

sections.

5.1 SRR Algorithm

The SRR loop scheduling strategy is outlined in Algorithm 2. The idea is to assign pairs of iterations to threads following a round-robin scheme, so that in the end, each thread is assigned to a workload that is near to the total average workload. For this, each pair is formed up with iterations not yet assigned to threads that have the highest and lowest loads.

This algorithm takes two input parameters: an array that gives the load of each iteration (A) and the number of working threads (n). Then, it returns a multiset (P) that indicates the thread/iteration assignment (*i.e.*, P_j is a set containing all iterations assigned to thread j).

The algorithm starts by invoking a Sorting Algorithm (line 2) before scheduling iterations in pairs using a round-robin scheme (lines 3 to 9). It is worth noting that we distinguish between two possible cases, depending on whether $|A|$ is odd or even. The simplicity of this algorithm leads to low requirements for both, time and space. The space complexity is restricted to the number of iterations to schedule, therefore being linear ($O(n)$). The time complexity, on the other hand, is dominated by the time to actually sort the iterations according to their load, which is logarithmic ($O(n \log n)$) when using the Quick-Sort Algorithm.

Algorithm 2 SRR loop scheduling strategy.

1 **function** SRR(A, n)
2   SORT(A, ascending order)
3   $j \leftarrow 0 \quad b \leftarrow 0 \quad d \leftarrow 1$
4   **if** $|A|$ is odd **then**
5     $P_0 \leftarrow P_0 \cup 0$
6     $b \leftarrow 1 \quad d \leftarrow 0$
7   **for** i **from** b **to** $|A|/2 - d$ **do**
8     $P_j \leftarrow P_j \cup \{i\} \cup \{|A| - i - d\}$
9     $j \leftarrow (j+1) \mod n$
10   **return** P

As stated before, SRR must know in advance the load of each iteration

to correctly balance the load among the application threads. Fortunately, there are several approaches to obtain or estimate the load of iterations. The most straightforward one is to instrument the target application and to profile it, either offline or online. This solution is likely to lead to the most accurate estimation, but it may demand an extra execution of the application (offline profiling) or impose a great overhead (online profiling). Moreover, this approach may not be applied to non-deterministic applications, such as simulations in which the seed value varies from one execution to another. The second approach is to use compiling information to perform this estimation based on code analysis. Indeed, this alternative may output a high-quality approximation for the load of iterations [9]. However, it may significantly increase compilation time and it will not work for irregular applications in which irregularity arises from the input data itself, such as Integer Sorting. Finally, for some applications, the load of iterations may be determined at runtime as a function of the input workload. However, it may may not be highly accurate for some applications. In Section 6.4, we show how we can apply this third approach in a real-world application. It is important to note, however, that SRR may also be coupled with the first and second approaches.

5.2 SRR Implementation in Libgomp

We implemented the SRR strategy in the *libgomp* library, which is the GCC's OpenMP runtime. This way, any parallel application that is built on top of OpenMP may seamlessly use our scheduling strategy. The enhanced version of this library is publicly available at `www.github.com/lapesd/libgomp`.

We made three main changes to the mainstream *libgomp* implementation. First, we changed the `GOMP_loop_runtime_start()` and `gomp_loop_init()` functions. The former one decides which scheduler to invoke, and we added the SRR strategy as a new option there. The latter function handles the scheduling task, and we inserted into it all the code for performing the SRR scheduling (Algorithm 2). Second, we added the `gomp_iter_srr_next()` function to the library. This function lookups the iteration/thread map output by the SRR strategy and effectively assigns iterations to the corresponding threads. Finally, we provided a new runtime function named `omp_loop_srr_set_workload()`, which sets the workload information for the next loop. The SRR strategy relies on this information to run.

To invoke the SRR strategy, the programmer should set the OpenMP environment variable `OMP_SCHEDULE` to `srr`, and select the `runtime` scheduler in the OpenMP `schedule` clause. Furthermore, the application should call the `omp_loop_srr_set_workload()` runtime function to inform the SRR strategy about the load of iterations in the next parallel loop. The application should correctly provide this information, otherwise the behavior is undefined.

Snippet 1: Usage of the SRR scheduler in OpenMP.

```
1     double cosinesum(int a, int b) {
2       double sum = 0.0;
3       unsigned tasks[b - a];
4
5       for (int i = a; i < b; i++)
6         tasks[i - a] = i;
7       omp_loop_srr_set_workload(tasks, b - a);
8
9       #pragma omp parallel for reduction(+:sum) schedule(runtime)
10      for (int i = a; i < b; i++)
11        for (int j = a; j < i; j++)
12          sum += cos(i);
13
14      return (sum);
15    }
```

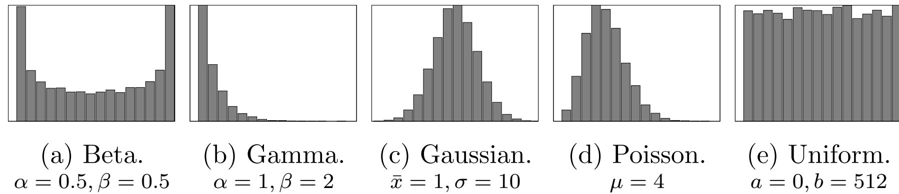
Snippet 1 illustrates the use of our scheduler. In this example, we perform a nested sum of cosines, which may be encountered in engineering and mathematics applications. In this case, we can determine how many additions each iteration in the outer loop will perform, and thus we can estimate their load. Therefore, we fill the `tasks` array with that information (lines 5 to 6), and we inform the SRR scheduler about this by calling `omp_loop_srr_set_workload()` (line 7). Next, we invoke the SRR strategy by choosing the runtime scheduler (line 9).

As final remark, it is worth noting that this is a proof-of-concept implementation and we intend to enhance it to make it even more user-friendly. In particular, we intend to (i) add `srr` as a new keyword to the `schedule()` clause; and (ii) add a `workload()` clause to the `#pragma omp` directive. This way, users can invoke our strategy as follows: `#pragma omp for schedule(srr) workload(tasks)`.

6 Experimental Evaluation

In this section, we discuss the results for the proposed methodology. First, we present validation results for our simulator. Then, we present an evaluation of the SRR strategy using our simulator. Finally, we assess its performance in a real-world application.

We considered the following parameters and configurations throughout the experimental evaluation. We generated synthetic workloads with the following PDFs: Beta, Gamma, Gaussian, Poisson and Uniform. Figure 5 presents these functions and their parameters. We considered several synthetic workloads by varying the initialization seed value of PDFs (from 1 to 20) and the number of loop iterations. To ensure consistency on the average workload output by each PDF we applied a correcting multiplying factor on each of them. The number of loop iterations was chosen according to a loop iteration phenomenon that we observed when using the GA to



(a) Beta. $\alpha = 0.5, \beta = 0.5$ (b) Gamma. $\alpha = 1, \beta = 2$ (c) Gaussian. $\bar{x} = 1, \sigma = 10$ (d) Poisson. $\mu = 4$ (e) Uniform. $a = 0, b = 512$

Figure 5: Probability Density Functions (PDFs) and their parameters.

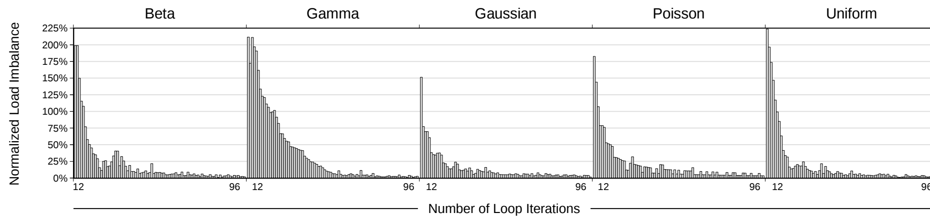


Figure 6: Impact of the number of loop iterations on the load imbalance with 12 threads.

explore the solution space of the Loop Scheduling Problem. We analyze this peculiar behavior in details in Section 6.1. We carried out the experiments on a machine powered by 2 Intel Xeon E5-2620 (6 cores each) with 64 GB of RAM. We fixed the number of threads to 12 (one thread per core) in all experiments, unless otherwise stated.

To provide a comparative evaluation, we studied two other loop scheduling strategies that are available in OpenMP: Static and Dynamic. Other strategies were not considered because the Dynamic one results in similar performance gains when an optimal chunk size is used [13]. In our experiments, we considered the best results for chunk sizes of 1, 2, and 4.

6.1 Impact of the Number of Loop Iterations on the Load Imbalance

The load imbalance is a property of an irregular application and its input workload. To better understand how this property behaves when varying the number of loop iterations, we carried out several experiments with the GA. Figure 6 shows the load imbalance for all the five PDFs we considered in this study, using 12 threads.

As it can be observed, when the number of loop iterations equals the number of working threads, the load imbalance anomaly is strongly present, being up to 223% (Uniform distribution). However, when the number of loop iterations is equal to or more than eight times the number of threads (*i.e.,* 96 iterations), this unwanted behavior significantly drops to 3% (Poisson distribution). Put it in other words, the more loop iterations, the lower is the load imbalance, regardless the input workload. We observed the same

phenomenon when we used different initialization seed values for the PDFs. Based on this observation in the following experiments we varied the number of loop iterations from 48 to 192 threads (4× and 12 threads).

6.2 Simulator Validation

Figure 7 shows the workload assigned to each thread by the Static, Dynamic and SRR strategies obtained from the simulator. These results concern to those scenarios where the latter strategy has performed the best possible, however we observed a similar behavior in most of the other scenarios. The blue line points out the optimal scheduling solution, which is based on the average-heuristic (recall Section 5), and it is presented for an upper-bound limit comparison. In all PDFs but Gamma, we observed that the SRR strategy performs similar to the optimum strategy, thus uncovering the potentials of workload-aware loop scheduling strategies. On the other hand, we observed that the Dynamic strategy overloads a small number of threads and assigns the remainder workload evenly to the others. This leads to a load imbalance and thus to a poor performance in contrast to our strategy. For the Gamma distribution, we noticed that both strategies faced difficulties on scheduling the loop iterations efficiently. Nevertheless, this behavior is explained by the characteristics of the workload itself, which presents many loop iterations with small loads, and few iterations with very large loads. This causes some threads to be assigned to heavy-loads and leads to a strong load imbalance.

In order to validate the simulator results, we designed and implemented a synthetic benchmark. This artificial program is outlined in Algorithm 3 and it performs the actual computation that the simulator simulates: it computes a single parallel loop using OpenMP. This synthetic benchmark takes as input five parameters:

- f: a PDF to generate the input workload;
- n: the number of loop iterations;
- s: the scheduling strategy to use;
- k: the number of threads;
- l: the load of one operation in the synthetic kernel.

First, the benchmark generates a synthetic workload w according to f (lines 1 to 4). Then, it sets the loop scheduling strategy and number of working threads to use to s and n, respectively. Finally, it performs dummy computations over the input workload (line 8). Note that the number of operations that the synthetic kernel actually executes is proportional to both, w and l. We use the latter parameter to adjust the load of loop iterations,

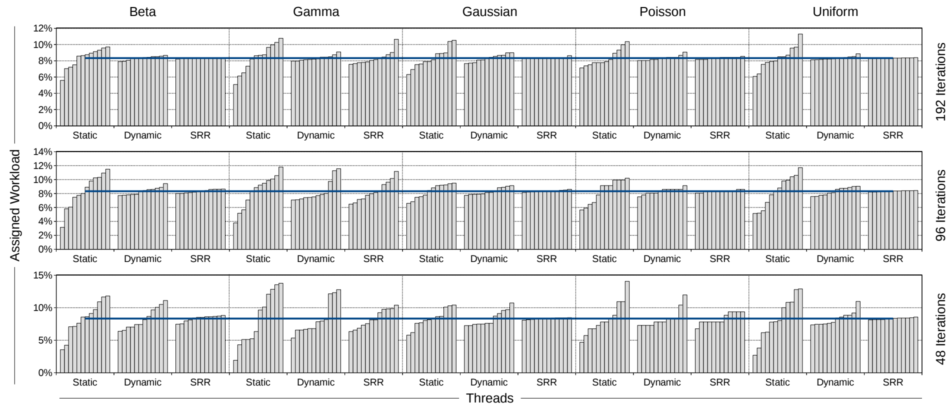


Figure 7: Workload assigned by the Static, Dynamic and SRR strategies in the simulator.

so as they are costly enough to step out from the overhead imposed by the OpenMP runtime environment itself. We set this value to 2×10^8 in our experiments.

Algorithm 3 Synthetic benchmark.

1 **function** SYNTHETIC-BENCHMARK(f, n, s, k, l)
2 $\quad$ w is an array size n
3 $\quad$ **for** i **from** 0 **to** n **do**
4 $\quad\quad$ $w_i \leftarrow$ random number according to f
5 $\quad$ set scheduler to s
6 $\quad$ set number of threads to k
7 $\quad$ **parallel for** i **from** 0 **to** n **do**
8 $\quad\quad$ SYNTHETIC-KERNEL(w_i, l)
9
10 **function** SYNTHETIC-KERNEL(w, l)
11 $\quad$ $a \leftarrow 0$
12 $\quad$ **for** i **from** 0 **to** w **do**
13 $\quad\quad$ **for** j **from** 0 **to** l **do**
14 $\quad\quad\quad$ $a \leftarrow a + 1$

We executed the synthetic benchmark, collected thread/iteration assignments, computed the load imbalance, and compared the results for this metric with the ones output by the simulator. Table 1 details the results for all the scenarios. For the Static and SRR strategies, we observed that the load imbalance output by our simulator agrees 100% with the results output in the synthetic benchmark. For the Dynamic strategy, on the other hand, we observed that our simulator is on average 99.90% accurate. The difference in the load imbalance between simulations and synthetic benchmarking experiments for the Dynamic comes from its non-deterministic behavior.

Loop Size	Beta	Gamma	Gaussian	Poisson	Uniform
48	99.96%	100.00%	99.96%	100.00%	99.91%
96	99.67%	99.98%	99.95%	100.00%	99.91%
192	99.70%	99.60%	99.90%	100.00%	99.90%

Table 1: Simulator accuracy for the Dynamic strategy.

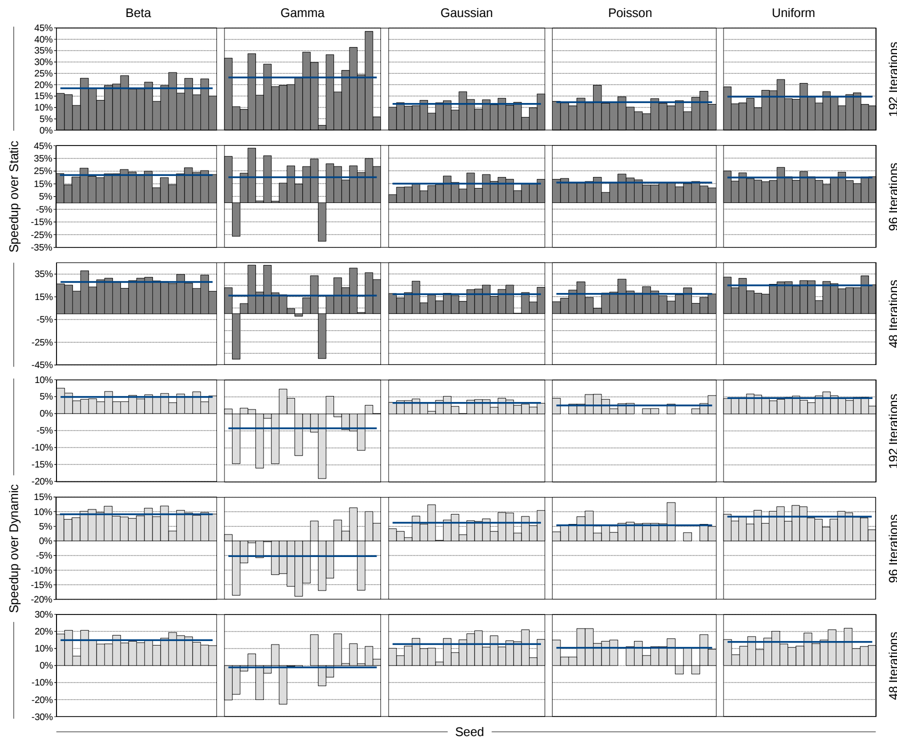


Figure 8: Performance gains of SRR when varying the initialization seed values of PDFs in the simulator.

6.3 SRR Performance

To evaluate the performance of the SRR strategy, we used the simulator proposed in our methodology. The rationale for choosing the simulation technique over synthetic benchmarking is three-fold: it enables us to (i) isolate the core variables of the Loop Scheduling Problem, (ii) precisely control the testing environment, and (iii) evaluate strategies for this problem with minimum efforts.

Figure 8 presents the performance gains of SRR over the Dynamic and Static strategies when we used different seed values to initialize each PDF on the simulated environment. The blue line outlines the mean speedup that we observed. Overall, SRR presented better performance than Static and Dynamic for all distributions. We observed the following average speedups

for the Beta, Gamma, Gaussian, Poisson and Uniform PDFs over the Static strategy, respectively: 28.80%, 11.12%, 14.56%, 15.18% and 19.83%. On the other hand, the average speedups for the Beta, Gaussian, Poisson and Uniform PDFs over the Dynamic strategy were, respectively: 9.63%, 7.37%, 6.09% and 8.96%. The maximum speedup obtained over the Static and Dynamic strategies were 37.89% (Uniform distribution, 48 loop iterations) and 21.74% (Poisson distribution 48, iterations), respectively. For the Gamma distribution, however, we observed a significant performance degradation for some seed values. The rationale behind this behavior comes from the nature the PDF itself: it has many iterations with very small load and a few with very large loads. This property causes the latter iterations to be assigned to threads at the beginning of the scheduling process, leading to a load imbalance. Indeed, the SRR strategy performs better on PDFs that have a balanced number of iterations with low and large loads. Finally, we observed less than 5% of performance loss over the Dynamic strategy for the Poisson distribution with 48 iterations.

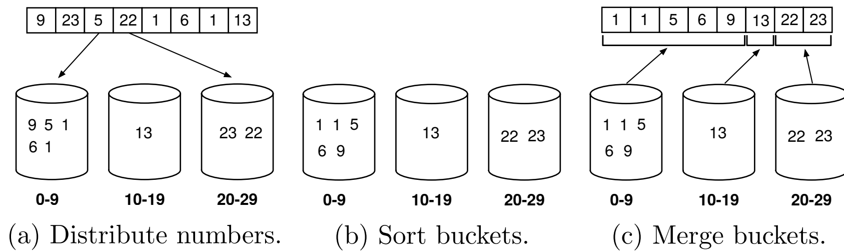
When considering the three different numbers of loop iterations, the maximum average speedup observed over both Static and Dynamic strategies occurred with 48 loop iterations. It is interesting to point out that these results agree with the results output by the GA and lead us to the following ultimate conclusion. The fewer is the number of loop iterations, the stronger is the load imbalance property and the higher is the performance gain of the SRR strategy. With this number of loop iterations, we observed an average performance gain of 19.94% and 12.95% over the Static and Dynamic strategies, respectively.

Finally, when analyzing the seed value, we observed that on average this variable has little impact on the performance gains of the SRR strategy. When considering the Beta, Gaussian, Poisson and Uniform PDFs, we observed a standard deviation value ranging from 0.92% (Uniform distribution over Dynamic) to 7.57% (Poisson distribution over Dynamic). Putting it in other words, when comes to workload-aware loop scheduling, the number of loop iterations and the PDF associated with the load of iterations, regardless its seed, are the most important core variables of the problem.

6.4 Real-World Application Kernel

In the previous section we uncovered the potential performance gains of SRR in a simulated environment. In this section, we illustrate the use of this strategy in a real-world application kernel, the Integer Sort (IS) kernel from the well-known NPB.

The IS kernel plays an important role in several applications, such as particle simulation [20]. It sorts an array of n integer numbers in parallel as follows. First, all numbers are divided according to their range into a fixed number of buckets (Figure 9a). Then, the numbers in each bucket are



(a) Distribute numbers. (b) Sort buckets. (c) Merge buckets.

Figure 9: Example of the computation performed by the IS kernel.

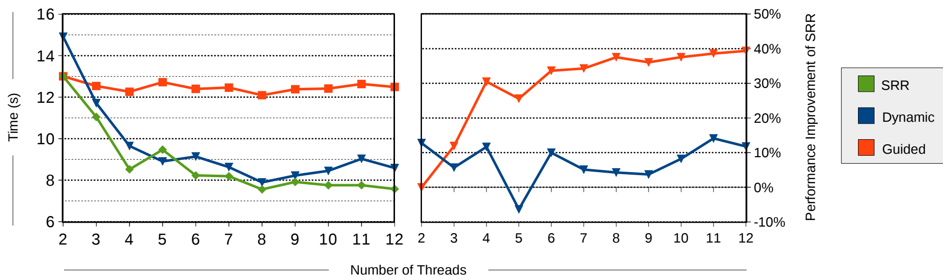


Figure 10: Time (left) and performance improvement of SRR over Guided and Dynamic (right).

sorted using the Counting Sort algorithm (Figure 9b). Finally, the contents of the buckets are merged to produce the final sorted sequence (Figure 9c).

The irregular behavior of this kernel comes from the fact that the amount of integer numbers in each bucket may differ from one to another, thus leading to different execution times to sort each bucket. Fortunately, however, it is possible to estimate the actual cost (load) for performing this computation for each bucket. The amount of time to sort a bucket is proportional to the number of numbers in there, and thus we can simply bookkeep this information while dividing numbers into the buckets. Based on this observation, we adapted the original IS implementation to support the SRR strategy.

Figure 10 presents the execution times obtained with the Guided, Dynamic and SRR strategies for the IS kernel, when sorting 2^{31} beta-generated integers using 32 buckets and varying the number of threads from 2 to 12. The maximum error that we observed, according to *Student*'s t-distribution, was 3.04% (Dynamic with 11 threads). Since IS kernel is fairly irregular, the Static strategy presented a very poor performance (30.63% worse than SRR on average) and thus it has been omitted from the figure. We also present the performance improvements obtained with SRR over the Guided and Dynamic strategies.

When varying the number of threads, we observed that SRR outperforms Guided in up to 39.37% (12 threads), and Dynamic in up to 14.10% (11 threads). We correlate this behavior with the thread/iteration schedul-

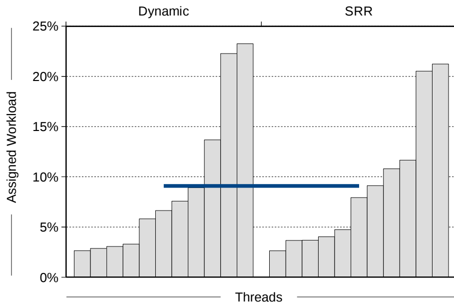


Figure 11: Assigned workload with 11 threads.

Thread #	Dynamic	SRR
1	6.45%	6.46%
2	6.23%	5.42%
3	6.03%	5.41%
4	5.79%	5.06%
5	3.28%	4.36%
6	2.45%	1.14%
7	1.52%	0.03%
8	0.17%	1.71%
9	4.58%	2.57%
10	13.18%	11.43%
11	14.16%	12.14%
AVG	5.80%	5.07%

Table 2: Deviation to the average workload.

ing that is output by each strategy. Since the Guided and Dynamic strategies are blind to the input workload, they may lead to a work distribution that overloads some threads with heavy loads, thus leading to a strong load imbalance in many cases. The SRR strategy, on the other hand, takes into account the input workload and distributes heavy loads evenly, thus decreasing the load imbalance anomaly and leading to a better overall performance.

We can confirm these findings with the following results. In Figure 11, we present the assigned workload for the Dynamic and SRR strategies when using 11 threads. The blue line outlines the best theoretical load balance, which would assign the average workload to all threads. As it can be observed, SRR assigns less work to the most overloaded thread than Dynamic. Furthermore, SRR assigns a workload to the threads that is closer to the average workload.

Table 2 shows how far the workload assigned to each thread is from the best theoretical load balance (the closer to zero the closer to the best theoretical load balance). For the Dynamic scheduler, the workload assigned to all threads is, on average, 5.8% away from the best theoretical load balance whereas it is, on average, 5.07% for the SRR scheduler.

If we consider only the most overloaded thread in Table 2, we can observe that its load is decreased from 14.16% (Dynamic) to 12.14% (SRR), thus reducing the overall execution time achieved by SRR. According to Figure 11, the only case in which SRR presented a performance degradation compared to Dynamic was with 5 threads. After performing the same analysis we concluded that, for this specific scenario, Dynamic achieved a slightly better load balance among the threads. The load of the most overloaded thread was increased from 10.31% (Dynamic) to 12.64% (SRR).

7 Conclusions and Future Work

When comes to loop scheduling, current strategies do not take into account the input workload of the application and thus turn out to be suboptimal in some scenarios. Based on this observation, we introduced a design methodology that allows researchers to study and assess the performance of workload-aware loop scheduling strategies. This methodology relies on (i) a GA to explore the solution space of this problem and to guide the design of new scheduling strategies; and (ii) a simulator to evaluate their performance. To validate the simulator, we compared its results with results obtained from a synthetic benchmark implemented in OpenMP. To validate the design methodology, on the other hand, we proposed a new scheduling strategy named SRR. We evaluated the performance of SRR in the simulated environment and with IS kernel from the NPB.

Our experimental results pointed out that the proposed simulator is 100% accurate for the Static and SRR strategies. For the Dynamic strategy, however, we observed that the proposed simulator is 99.90% accurate, on average. Concerning the design methodology itself, the proposed simulator and GA helped us to propose a new workload-aware loop scheduling strategy named SRR. In contrast to other loop scheduling strategies available in OpenMP, SRR considers the input workload to better assign loop iterations to threads. The SRR strategy was implemented into GCC's OpenMP runtime, so that any parallel application that is built on top of OpenMP may seamlessly use it. We made our implementation publicly available under the GPL 3 License, thus enabling other researchers to further enhance it.

Finally, we compared the performance of SRR with the Static and Dynamic strategies available in OpenMP. In the simulated environment, we considered five PDFs, with 20 different initialization seed values for each one. We observed that SRR improved the performance of Static and Dynamic strategies in up to 37.89% and 21.74%, respectively. For a real-world application kernel, the IS kernel from NPB, SRR led to a better load balancing, achieving up to 39.37% and 14.10% better performance than Guided and Dynamic strategies, respectively.

As future work, we intend to further improve the simulator and SRR to consider application- and platform-dependent variables. For instance, the memory affinity between loop iterations could be considered while scheduling loop iterations to threads to avoid (when possible) remote memory accesses in NUMA platforms. SRR could also be extended to work with OpenMP tasks, making it suitable for parallel applications that feature task parallelism. Moreover, processor heterogeneity could also be considered to better balance the load among heterogeneous processors (*e.g.,* ARM big.LITTLE). We also intend to assess the SRR performance with other real-world applications and PDFs, and to study workload prediction techniques based on Machine Learning that would enable workload-aware loop scheduling strate-

gies to be used in a wide range of applications. Additionally, we plan to add an oracle scheduler based on the GA in OpenMP, so that for any particular application we can estimate the upper-bound load balancing performance one strategy can achieve. Finally, we intend to employ the proposed methodology to guide the design of new workload-aware loop scheduling strategies in both homogeneous and heterogeneous platforms.

Acknowledgements

This work was supported by FAPEMIG, FAPERGS and INRIA under the ExaSE cooperation project grant APQ-03206-13, by CNPq under the projects grants 458530/2014-0 and 233223/2014-2, and by STIC-AmSud/CAPES scientific-technological cooperation programs under EnergySFE research project grant 99999.007556/2015-02.

References

[1] Kulkarni M, Burtscher M, Inkulu R, Pingali K, Casçaval C. How much parallelism is there in irregular applications? *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, ACM: New York, NY, USA, 2009; 3–14, doi: 10.1145/1504176.1504181.

[2] Skiena SS. *The Algorithm Design Manual*. 2nd edn., Springer, 2008.

[3] Ding W, Zhang Y, Kandemir M, Srinivas J, Yedlapalli P. Locality-aware mapping and scheduling for multicores. *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Shenzhen, China, 2013; 1–12, doi:10.1109/CGO.2013.6495009.

[4] Durand M, Broquedis F, Gautier T, Raffin B. An efficient openmp loop scheduler for irregular applications on large-scale numa machines. *OpenMP in the Era of Low Power Devices and Accelerators*, *Lecture Notes in Computer Science*, vol. 8122, Rendell A, Chapman B, Müller M (eds.). Springer Berlin Heidelberg, 2013; 141–155, doi: 10.1007/978-3-642-40698-0_11.

[5] Dagum L, Menon R. Openmp: An industry standard api for shared-memory programming. *IEEE Computational Science Engineering* 1998; **5**(1):46–55, doi:10.1109/99.660313.

[6] Srivastava S, Sukhija N, Banicescu I, Ciorba F. Analyzing the robustness of dynamic loop scheduling for heterogeneous computing systems. *International Symposium on Parallel and Distributed Computing (ISPDC)*, Munich, Germany, 2012; 156–163, doi:10.1109/ISPDC.2012.29.

[7] Penna PH, Castro M, Freitas HC, Broquedis F, Méhaut JF. Uma metodologia baseada em simulação e algoritmo genético para exploração de estratégias de escalonamento de laços. *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD-SSC)*, Sociedade Brasileira de Computação, 2015.

[8] Olivier SL, Porterfield AK, Wheeler KB, Spiegel M, Prins JF. Openmp task scheduling strategies for multicore numa systems. *Int. J. High Perform. Comput. Appl.* 2012; **26**:110–124, doi:10.1177/1094342011434065.

[9] Thoman P, Jordan H, Pellegrini S, Fahringer T. Automatic openmp loop scheduling: A combined compiler and runtime approach. *OpenMP in a Heterogeneous World*, *Lecture Notes in Computer Science*, vol. 7312, Chapman B, Massaioli F, Müller M, Rorro M (eds.). Springer Berlin Heidelberg, 2012; 88–101, doi:10.1007/978-3-642-30961-8_7.

[10] Hajieskandar A, Lotfi S. Parallel loop scheduling using an evolutionary algorithm. *International Conference on Advanced Computer Theory and Engineering (ICACTE)*, vol. 1, Chengdu, China, 2010; 314–319, doi:10.1109/ICACTE.2010.5579010.

[11] Sukhija N, Malone B, Srivastava S, Banicescu I, Ciorba F. Portfolio-based selection of robust dynamic loop scheduling algorithms using machine learning. *IEEE International Parallel Distributed Processing Symposium Workshops (IPDPSW)*, Phoenix, USA, 2014; 1638–1647, doi:10.1109/IPDPSW.2014.183.

[12] Srivastava S, Malone B, Sukhija N, Banicescu I, Ciorba F. Predicting the flexibility of dynamic loop scheduling using an artificial neural network. *IEEE International Symposium on Parallel and Distributed Computing (ISPDC)*, Bucharest, Romania, 2013; 3–10, doi: 10.1109/ISPDC.2013.10.

[13] Balasubramaniam M, Sukhija N, Ciorba F, Banicescu I, Srivastava S. Towards the scalability of dynamic loop scheduling techniques via discrete event simulation. *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Shanghai, China, 2012; 1343–1351, doi:10.1109/IPDPSW.2012.171.

[14] Gu WX, Wen LH. A new method for parallel planning via heuristic search. *International Conference on Machine Learning and Cybernetics (ICMLC)*, vol. 6, 2007; 3128–3132, doi:10.1109/ICMLC.2007.4370685.

[15] Chen C, Rickert M, Knoll A. Path planning with orientation-aware space exploration guided heuristic search for autonomous parking and maneuvering. *IEEE Intelligent Vehicles Symposium (IV)*, 2015; 1148–1153, doi:10.1109/IVS.2015.7225838.

[16] Hatamlou A, Abdullah S, Othman Z. Gravitational search algorithm with heuristic search for clustering problems. *Conference on Data Mining and Optimization (DMO)*, 2011; 190–193, doi:10.1109/DMO.2011.5976526.

[17] Konfrst Z. Parallel genetic algorithms: Advances, computing trends, applications and perspectives. *IEEE International Parallel Distributed Processing Symposium (IPDPS)*, Santa Fe, New Mexico, 2004; 162–, doi:10.1109/IPDPS.2004.1303155.

[18] Goldberg DE, Deb K. A comparative analysis of selection schemes used in genetic algorithms. *Foundations of Genetic Algorithms*, Morgan Kaufmann, 1991; 69–93.

[19] Bhandari D, Murthy CA, Pal SK. Genetic algorithm with elitist model and its convergence. *International Journal of Pattern Recognition and Artificial Intelligence* 1996; **10**(06):731–747, doi:10.1142/S0218001496000438.

[20] Bailey DH, Barszcz E, Barton JT, Browning DS, Carter RL, Dagum L, Fatoohi RA, Frederickson PO, Lasinski TA, Schreiber R, *et al.*. The nas parallel benchmarks. *International Journal of High Performance Computing Applications* 1991; **5**(3):63–73.