ARTful: A specification for user-defined schedulers targeting multiple HPC runtime systems

Alexandre Santana, Vinicius Freitas, Marcio Castro, Laércio Lima Pilla, Jean-François Méhaut

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

ARTful: A specification for user-defined schedulers targeting multiple HPC runtime systems

Alexandre Santana[1] | Vinicius Freitas[1] | Márcio Castro[1] | Laércio L. Pilla*[2] | Jean-François Méhaut[3]

[1]Federal University of Santa Catarina (UFSC), Florianópolis, Brazil
[2]LRI, Univ. Paris-Saclay – CNRS, Orsay, France
[3]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, Grenoble, France

Correspondence
*Laércio Lima Pilla, Laboratoire de Recherche en Informatique, Bât 650 Ada Lovelace, Université Paris-Saclay, 91405 Orsay, France Email: pilla@lri.fr

Abstract

Application performance, developer productivity, and portability to multiple computational environments are some of the desired traits in High Performance Computing (HPC) applications. The evolution of programming models and runtime systems are crucial aspects on enabling applications to run faster on new architectures without requiring large reimplementation efforts. Runtime systems are central software entities in this software stack and have the role of scheduling and distributing the application workload among the processing units. However, most runtime library implementations offer little support for user-defined scheduling and provide only a limited set of scheduling policies. Extensions to provide better scheduling options usually require modifications to the runtime library and are hard to maintain across the rapidly evolving HPC domain. In this paper, we propose a set of *ARTful* specifications for abstracting system-specific scheduling functionalities and achieve global scheduler implementations that can be tested in isolation from the runtime and reused in multiple system libraries. We also showcase an implementation of our *ARTful* specifications as MOGSLib, a metaprogramming-oriented library that integrates generic global schedulers that can be specialized to function both as Charm++ load balancers and OpenMP loop schedulers. We analyze the overhead of schedulers implemented in MOGSLib in comparison to system native solutions and discuss the qualitative benefits of developing system-independent global schedulers. We show that our implementations can sometimes perform scheduling decisions even faster than their original implementations with negligible overhead in the execution times of synthetic applications and molecular dynamics kernels.

KEYWORDS:
scheduling; runtime systems; portability; code complexity; component; self-adaptable

1 | INTRODUCTION

Applications for High Performance Computing (HPC) environments are built atop of long lasting standards created by the collaborative efforts of industry and academy. Achieving performance without totally sacrificing portability in HPC systems is only possible due to tools like OpenMP[1], MPI[2] and BLAS[3]. One of the reasons behind HPC applications' complexity is the expression and distribution of their parallel tasks. Indeed, developers have a wide array of parallel programming libraries

to transform serial applications into parallel or distributed solutions. However, these tools alone do not provide out-of-the-box performance to all problems or systems.

Resource management is paramount to both portability and performance of parallel applications. It can be used to achieve predictable application performance regardless of the execution environment[4]. The management of the distribution of applications tasks over the available resources is done by *global schedulers*. They serve the purpose of mitigating the effects of load imbalance, costly communications or data transfers, resource variations, and others that can affect application performance. Nevertheless, each application, system, and even parallel programming model proposes new challenges for the design of scheduling algorithms[5]. Most, if not all, parallel programming libraries have at least some software mechanism to handle global scheduling, but only a few offer integrated development support for user-defined schedulers[6,7].

Future generation systems are bound to be composed of increasing numbers of compute nodes while showcasing different and diverse hardware architectures[4]. It is likely that these systems will build upon current software tools for exposing these resources to applications. As system diversity increases, most of the additional complexity to orchestrate these resources will be absorbed by runtime libraries. Standards evolve too slowly to accommodate the increasing pace of technological advances and, even today, integrating novel scheduling solutions to these libraries relies on unofficial and hard to maintain initiatives[8]. Indeed, efforts to create standards for user-defined scheduling can be observed in OpenMP[9,10]. While such propositions have proven useful for discussing the future of HPC standards, no studies have implanted these mechanisms into existing runtime libraries.

Scheduling algorithms are bound to be used in different layers of future systems. Ideally, their codes should be reusable among similar runtime libraries, and be adaptable and scalable to different contexts, as this would lead to better productivity when transcribing schedulers between runtime systems, writing new runtime systems that need schedulers, or showing the benefits of a new scheduling algorithm in comparison to the state of the art, among others. Although we see developments using components to reduce application complexity in related scenarios[11,12], we still lack standard solutions for reusable global schedulers.

In order to overcome these issues, we propose a set of specifications (*ARTful*) to shift scheduling solutions from their current orientation to specific runtime libraries to more adaptable and generic solutions. Using our *ARTful* specifications, we implemented a global scheduling library and a set of schedulers, and implanted them on two different runtime libraries, enabling us to compare our schedulers with their equivalent runtime-native counterparts. This work iterates over our previous study over the software relationship between global schedulers and the runtime system[13] and proposes the following contributions:

1. A core set of specifications to compose generic and reusable global schedulers based on Abstraction, Reusability, and Testability (*ART*) principles;
2. A streamlined development and integration process for user-defined scheduling solutions into multiple runtime systems;
3. The construction of a reference implementation of system-independent global schedulers based on compile-time specialization techniques; and
4. A set of experiments using two runtime systems that include extensions to original global scheduling algorithms (OpenMP multi-loop support).

The remainder of this paper is organized as follows. Section 2 further describes the problems in global scheduling software and denotes related work targeting modularity and composability on HPC environments. Next, in Section 3, we present our *ARTful* specifications. In Section 4, we explain how we implemented the proposed *ARTful* specifications in a library based on compile-time specialization techniques. We then discuss our experiments and results in Section 5. Finally, we conclude this work in Section 6.

2 | BACKGROUND ON GLOBAL SCHEDULING

2.1 | Definitions

Global scheduling may be described as the problem of defining *where* to run a *task*, leaving the decision of *when* to run a task to local scheduling[14]. This definition is broad enough to encompass different scheduling activities, such as *load balancing*, *topology mapping*, and *loop scheduling*. A common component in these activities is the **scheduling policy**, an algorithm to decide the mapping of tasks to resources given a specific objective. In its core, a scheduling policy can be regarded as a function that decides the output by solely analyzing its input. For instance, Algorithm 1 illustrates the Longest Processing Time first

(LPT) scheduling policy, a list scheduling algorithm that tries to minimize the load of the most loaded resource (*makespan*) by mapping tasks in order from most to least loaded (longest to shortest processing time).

Algorithm 1 The Longest Processing Time first (LPT) scheduling policy.

1: **procedure** LPT(Tasks T, Task loads (processing times) L_T, Resources R)
2: $L_R \leftarrow 0$ ▷ The resources start with no load
3: $T' \leftarrow T$ ▷ List of tasks to be mapped
4: $M \leftarrow \emptyset$ ▷ The mapping of tasks to resources starts empty
5: **while** $T' \neq \emptyset$ **do** ▷ While there are tasks left to map
6: $t \leftarrow \arg\max_{t' \in T'} L_T(t')$ ▷ Take the unmapped task with the largest load (longest processing time)
7: $r \leftarrow \arg\min_{r' \in R} L_R(r')$ ▷ Take the resource with the smallest load
8: $M(t) \leftarrow r$ ▷ Map the task to the resource
9: $L_R(r) \leftarrow L_R(r) + L_T(t)$ ▷ Update the load of the resource
10: $T' \leftarrow T' \setminus \{t\}$ ▷ Remove the task from the set of tasks to be mapped
11: **end while**
12: **return** M ▷ Return the computed mapping of tasks to resources
13: **end procedure**

As we can notice, the scheduling policy itself does not care: (i) if the tasks are actually objects, threads, processes, loop iterations, or others; (ii) if the resources are cores, processors, compute nodes, or others; nor (iii) if the processing times were measured, estimated, given by a user, or come from somewhere else. All of these details and some more related to the environment where scheduling happens are represent by the **scheduling context**. By combining a scheduling context and one or more scheduling policies, we are able to create a **global scheduler** to provide scheduling decisions in a runtime system.

2.2 | Current state of global scheduling software in runtime systems

Scheduling solutions are commonly implemented at runtime level and are present in popular libraries such as StarPU[7] and Charm++[6]. Standards like OpenMP[1] even describe the common scheduling solutions that must be implemented in a given compliant library. The global scheduling software in these runtime systems is depicted in Figure 1.

The relationship between the runtime components and an instance of a global scheduler native to the runtime, illustrated in Figure 1, shows that the scheduler is contained within the library codebase, and that it accesses runtime data structures and functionalities within the runtime directly. Although data structures and functionalities may be placed in a scheduling API (such as in Charm++), they can only be reused by different scheduling policies in the context of the same runtime system. This direct relationship creates an implicit mutual dependence between the two components. On the scheduler side, it depends on the runtime library syntax to fetch its input data and apply its decision. Moreover, the runtime library depends on the scheduler to employ these data structures and functionalities so the system, as a whole, can achieve its performance goals.

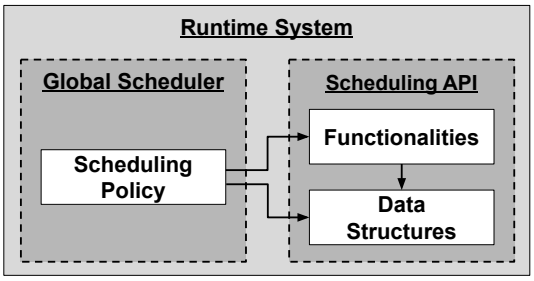


FIGURE 1 Anatomy of the scheduling solutions native to current runtime systems. Schedulers tend to reside inside the runtime system (RTS), and their direct use of RTS functionalities and data structures undermines portability.

This interaction between schedulers and runtime systems has negative effects on *portability* and *productivity*. Portability is affected by the difficulty to adapt and reuse a scheduler implemented in one runtime system to another, as this requires the reimplementation of accesses to data structures, function calls, and others. Productivity is affected by this too, but also by the time it takes to write a new scheduler in this scenario, and by the additional efforts to test and debug the scheduling codes that have to be, in many cases, integrated directly into runtime system. In a more extreme case, a runtime system's code may need to be modified in order to integrate new global schedulers because it may lack an interface for such additions [15].

The current lack of scheduler portability is aggravated when taking into account the large array of mechanisms being developed in modern scheduling solutions. Nevertheless, these same mechanisms enable solutions capable of evaluating the topology affinity [16,17], controlling processor frequencies [18], assessing dynamic workloads [15], data locality [19] and others in a multitude of systems. The inclusion of new technologies will require further specialization to scheduling solutions and eventually require more portability efforts if not explicitly managed.

2.3 | Related work

Our work targets the problem of global scheduling in HPC runtime systems, particularly aiming for system-independent solutions with better quality of software. Although our discussion employs some concepts found in runtime systems like StarPU and Charm++, the fundamental objectives of this work are different from the objectives of these runtime systems. When analyzing the state of the art, we considered related work that have as primary goals the improvement of modularity, reusability, and interoperability in parallel environments.

On the topic of component-based approaches for modularity, Aumage et al. [12] proposed the combination of task-based decomposition with component-based software models. Their model uses explicit definitions of data transmission to create dependencies and components to define the tasks to be executed. They successfully abstracted segments of code from real applications into a component system over the StarPU runtime system. Through their proposed component system (*COMET*), users can develop the parallel segments of an application as components that will be later mapped to tasks and executed by the StarPU system. Nevertheless, this was planned for the computations of the applications and not to the schedulers.

Grossman et al. [11] proposed that, through a better description of a component's connections, parallel libraries can be composed to collaborate instead of competing for resources. They employ modern language facets such as lambda functions and asynchronous calls to propose a novel modular runtime system that connects components through lambda functions. Their objective is to create a decomposed runtime system that enables different scheduling models, as opposed to creating a specific software for scheduling which constitutes the main difference when compared to our work.

Efforts to propose a way of introducing user-defined scheduling in OpenMP also relate to our work as our approach enables this feature. Kale et al. [9] propose the extension of the OpenMP standard with directives to enable users to declare and define their own strategies to schedule parallel loop iterations. Their work promotes the discussion by proposing a software interface to loop schedulers and leveraging which features would OpenMP library providers support. Meanwhile, Bak et al. [10] propose an API for user-defined schedulers in OpenMP. Our work analyzes the practical aspects of implementing schedulers decoupled from the runtime system. We envision that our work fills the transition gap of current global scheduler development and the adoption of standards such as the one discussed in these works. Among other differences, our work aims to enable a way for scheduling solutions to abstract the system-specific scheduling mechanisms found within runtime libraries.

The demand for variety and user-defined scheduling is also a studied topic in the scope of real-time operating system. Similarly to HPC runtime systems, the kernel must abstract basic functionalities to applications and scheduling is one of them. Mollison and Anderson [20] proposed user-defined schedulers that could be implemented with limited changes to the kernel and be used in multiple operating systems kernels. They applied a common higher level API to enable the user to manipulate schedulers independently of the underlying kernel. Those higher level directives are translated by a driver and forwarded to the kernel and C POSIX library function calls. Their solution enables schedulers to be developed out of the kernel-space with abstract implementations for base functions which involves thread locking, synchronization and other functionalities. It is important to emphasize that the overhead of the technique was acceptable even on real-time constraints, which is the most critical metric for schedulers in real-time operating systems.

Our goal is to enable the reuse of *scheduling policies* in different *HPC contexts* by providing a way to express the relationship between these two entities. We refrain from altering the relationship between those components in favor of compatibility. We also recognize that each individual runtime library have its own scheduling needs that might not necessarily be aligned to the goals of other libraries. Furthermore, we aim to provide a compatibility layer for the inclusion of new scheduling solutions that

have been conceived for different systems with a common subset of characteristics. We provide a set of specifications with these objectives in mind in Section 3.

At a software level, we propose that global schedulers expose their requirements to runtime libraries using software indirection layers (e.g., inheritance, lambdas, components and templates) instead of implicitly conforming to the library's constraints. This allows the expression of a behavior that can be specialized to a different set of components and heuristics, hence new scheduling contexts. Although different indirection layer implementations may be employed, we aim for a compile-time approach using metaprogramming as it yields low runtime overhead. We explain our software implementation in Section 4.

3 | THE ARTFUL SCHEDULER SPECIFICATIONS

The *ARTful scheduler specifications* provide guidelines for the development of portable global schedulers originated from our experience creating a global scheduling library (more details in Section 4). *ART* is an acronym for *Abstraction*, *Reusability* and *Testability* which are the main design pillars behind the specifications. Our approach to obtain reusability is based on the premise that it is possible to implement a scheduling policy agnostic to one or more target runtime libraries. Indeed, by achieving this goal, the specifications also provide support for user-defined scheduling solutions in these runtimes. Furthermore, this reusability should not come at the price of the degradation of scheduler performance, otherwise specialized solutions would still be preferred by users and developers.

We present our specifications as a set of software abstractions that encapsulates the necessary concepts for achieving runtime library-independent global schedulers. The *ARTful* specifications are described in the following sections and are constituted by the following abstractions: (i) the scheduling policy; (ii) the global scheduler; (iii) the scheduling context and (iv) the runtime library adapter.

3.1 | ARTful abstractions overview

The anatomy of an *ARTful* scheduling solution is presented in Figure 2. It is based on a set of abstractions to define a different approach to integrate global schedulers in runtime libraries. The specifications require limited additions to the runtime library codebase and do not impose alterations to its native functionalities or how those are presented to the user. Instead, the support for *ARTful schedulers* is similar to a runtime library extension.

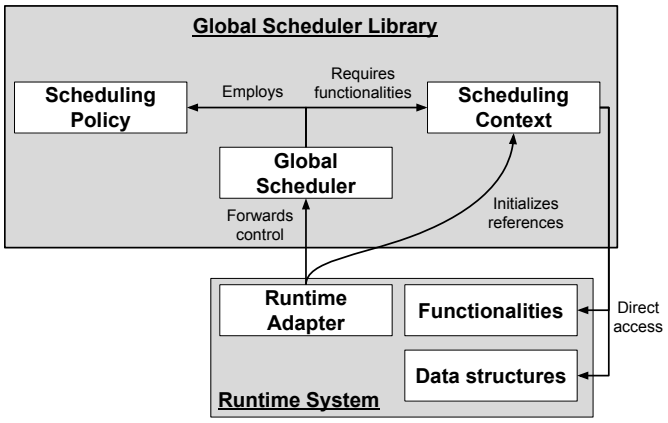


FIGURE 2 The *ARTful* scheduling solution anatomy. The components of the scheduling library can be reused in different scenarios. Changes to the runtime system are limited to a runtime adapter.

A comparison between Figures 2 and 1 reveals several benefits of *ARTful* schedulers. Firstly, we are able to extract the main global scheduling software from runtime libraries, creating the opportunity to reuse it with different runtimes. Furthermore, the only necessary addition to a runtime library is the **runtime adapter** component, meaning that new schedulers, with similar dependencies, can be used without any other changes to the runtime system. Besides that, we can notice that the global scheduling

library is now split into **scheduling policy**, **global scheduler**, and **scheduling context** components. These components can be developed by different stakeholders (e.g scheduling context might be provided by the maintainers of the runtime system), reused in different situations (e.g reuse scheduler on a new context), tested independently, and composed to fit the scheduling requirements of different runtime libraries.

When using *ARTful* schedulers, the runtime library starts scheduling by calling the runtime adapter. Once called, the adapter delegates the scheduling decision to a global scheduler external to the runtime codebase. As the adapter is an internal component of the runtime library, it can access other data structures and functions in its namespace. The adapter proceeds to expose these mechanisms, rather than employing them, to the *ARTful components* outside of the runtime scope. Indeed, this is mandatory to achieve portable implementations while retaining the original behavior of native solutions. A more detailed description of each other abstraction presented in Figure 2 is given in the upcoming sections.

In a nutshell, each *ARTful component* and its specifications are designed to represent one stakeholder of the scheduling solution software stack and its needs. The author of a scheduling policy is interested in describing its algorithm unbounded by current technologies or implementation details. Runtime system and architecture providers aim to express a context optimized for a set of applications along with tools to better distribute their workload (e.g schedulers and/or scheduling APIs). Global scheduler developers evaluate the existing policies and propose adaptations so that these policies can fit the current technologies and implementations. Furthermore, the runtime adapter originates from the segregation of a scheduling solution into the aforementioned components. Equipped with a clear distinction of responsibilities and a defined interface, different software developers can work independently on each of these components. Indeed, this scenario promotes the use of software engineering techniques to enable the reuse not only within a component codebase (e.g a scheduling context composed of several other abstractions) but also among the component implementations (e.g a scheduling context that serves several global schedulers).

It is important to notice that different software modeling techniques can be employed to construct the runtime adapter. In fact, the implementation is highly reliant on how the host runtime system software is structured. We discuss the runtime adapter broad specifications in Section 3.5 and discuss our implementation for OpenMP and Charm++ libraries later in Section 4.4.

3.2 | Scheduling policy abstraction

At the core of any global scheduler are the algorithms that define the mapping of tasks to resources. These algorithms are referred to as **scheduling policies** and they are found in different flavors of goals and heuristics. A given policy may require input data to analyze the platform distributed topology [17], power consumption data from each of its processors [21], the workload of each work unit [22], the memory architecture of the machine [23], or others. This diversity makes the task of describing a unified interface for this entity almost impossible without sacrificing either simplicity or flexibility.

The *ARTful* specifications for defining a scheduling policy refrain from creating a common description of a scheduler in terms of steps or input. Instead, we propose to define this entity in terms of their most common characteristic — the task mapping output — and define rules to guarantee the support of algorithm diversity. The following are the *ARTful* specifications for the implementation of a scheduling policy as a software component in the global scheduler assembly:

ARTful Scheduling Policy specification

SP-1 It must have only one procedure in its public interface.

SP-2 It must not record state.

SP-3 The output of its procedure must be a global schedule.

SP-4 A given policy must be allowed to define its own input parameters tuple.

SP-5 A given policy must have flexible data types to account for runtime library variations.

SP-1, SP-2 and SP-3 characterize the scheduling policy component as an entity with a single purpose in the global scheduler assembly. They characterize this abstraction as a black box that can be easily tested by employing inputs with predictable outputs. SP-3 and SP-4 account for the well-defined objective of the component and the component's flexibility to define multiple approaches, respectively. In special, SP-4 guarantees that novel policies can conform to this specifications regardless of new

technologies. Finally, SP-5 is a prerequisite for implementation portability. It is not uncommon that runtime libraries use different data types to represent common concepts such as the workload of a task and communication information. A crucial step in portability is assuring that these data types can be adapted when applying the scheduler to a context with different data type definitions. These specifications can be achieved in most general purpose programming languages and are enough to describe a generic scheduling policy.

3.3 | Global scheduler abstraction

If the policies define the algorithmic approach to solve the mapping problem, the **global scheduler** abstraction defines the shape of the scheduling solution. The *ARTful* global scheduler specifications ensure that there is a syntactic agreement between a scheduling context and a scheduling policy. The global scheduler entity expresses which operations must be provided in any given context for supporting its scheduling policies. It must manipulate the scheduling context to gather input and operations to fulfill the requirements of the scheduling policy. The *ARTful* specifications for this entity are as follows:

ARTful Global Scheduler specification

GS-1 It must have only one procedure in its public interface.

GS-2 Its procedure must output a global schedule.

GS-3 Its procedure must employ at least one scheduling policy.

GS-4 All global schedulers must share a common procedure signature.

GS-5 The global scheduler must create an interface for manipulating the scheduling context.

GS-6 The global scheduler must be able to work on any scheduling context that fulfills its interface.

GS-1, GS-2 and GS-3 define the abstraction's goals as a scheduling policy software container. GS-4 creates a common syntax for calling global schedulers which enables the construction of selection procedures for selecting the appropriate scheduler (e.g., lambdas, function pointers and templates). GS-5 defines the relationship between global scheduler and the scheduling context. This is an important step as it states that the global scheduler developer has the freedom of creating its own specifications for a scheduling context implementation. It allows the development responsibilities to be split into two different stakeholders, the scheduler implementation provider and the context implementation provider. Finally, GS-6 enforces that it is the responsibility of the global scheduler developer to design an interface that is expressive enough to provide portability (e.g., make use of appropriate function decorators and attributes).

3.4 | Scheduling context abstraction

The **scheduling context** abstraction represents the environment where the scheduling solution is inserted into. It encompasses any software or hardware abstraction that might affect the behavior of the solution. This might reflect to runtime library functionalities, parallel programming model abstractions, hardware architectural characteristics (e.g NUMA, DVFS) and others. The scheduling context implementation must be flexible enough to represent environments where: (i) the user can provide scheduling input[24]; (ii) the application is instrumented to assist the scheduler[25,26]; or (iii) libraries allows for complex functionalities, such as communicating with other scheduling endpoints[27].

A given implementation of a scheduling context is specialized for the environment it represents and, most notably, to one target runtime system. As such, it should be paired to and employ the functionalities and data structures exposed by a *ARTful runtime library adapter*. Ultimately, the scheduling context plays the role of linking the global scheduler to the semantics of the other HPC software components. We can take into consideration the Charm++ system as an example. Charm++ instruments the applications, so it can calculate the processing time of each individual application task. The system exposes this information for its load balancers, which are expected to employ them as load estimations for the tasks in the next application iteration cycle. Indeed, Charm++ depends on its schedulers to make use of this functionality, otherwise it would not be possible to achieve its

adaptive runtime status. These features are fundamental to Charm++'s scheduling scheme and might be absent or irrelevant on other runtime libraries and even optional in future versions of Charm++. Portable scheduling policies must be isolated from specialized behaviors such as this one, otherwise they become dependent on its semantics (i.e., they require an instrumented application). The explicit definition of a *scheduling context abstraction* relieves the scheduling policy from these issues by taking the role of situating the policy in the appropriate context. This *ARTful abstraction* has the following specifications:

ARTful Scheduling Context specification

SC-1 It encapsulates every context-sensitive feature required by a runtime library scheduler.

SC-2 It implements all the interface requirements defined by one or multiple global schedulers.

SC-3 It can be attached to any global scheduler with compatible requirements.

SC-4 It generates errors if attached to a global scheduler with incompatible requirements.

SC-5 It is accessible within the application and user code.

SC-1 states that the scheduling context is the *ARTful* component responsible form encapsulating the global scheduler transformations to a given runtime system. SC-2 reinforces the role of this abstraction as a functionality provider for global schedulers. Without following an interface defined by a global scheduler, a context implementation has no purpose. SC-3 and SC-4 are meant to guarantee the correct software relationship between scheduler and context implementations. Also, SC-4 states that an error must be issued if an incorrect composition is made as to avoid implementations that fail silently. Finally, SC-5 guarantees that users can access the context instances in the application code. This characteristic allows schedulers to access data from multiple sources regardless of which entities are managing it. As a consequence, users can define scheduler functionalities within scheduling contexts without the need of extending the runtime API, allowing the use of user-defined procedures in runtime systems that have no native support for such features.

3.5 | Runtime library adapter abstraction

The *ARTful* **runtime adapter** is an entity responsible for connecting the other abstractions to a runtime library. Regardless of the support for extending the set of policies in a system, we still have not found HPC runtimes without some internal scheduling module. As such, a runtime adapter can be inserted into the runtime library as a novel user-defined scheduler either by hacking the library or, when available, by making use of the system's development support to schedulers (such as an interface or a scheduling framework). An implementation of this concept is equivalent to a global scheduler native to the runtime system that, instead of making a scheduling decision itself, forwards this responsibility to a decoupled *ARTful* global scheduler. Additionally, the adapter has the role of initializing references to the runtime library data structures, so that those can be exposed to the *ARTful scheduling context*.

It would be too time-consuming or even impossible to create a unified specification for this component. Each runtime portrays a unique design to express its global schedulers and, if creating a common interface for all of them would be feasible, portable solutions would already exist. Nonetheless, runtime adapters must adhere to the system's routines while portraying the following behavior for *ARTful* scheduling solutions:

ARTful Runtime Adapter specification

RA-1 It must act as a native scheduler in the runtime system that presents a set of *ARTful* schedulers.

RA-2 It initializes references to system data structures within *ARTful* scheduling contexts.

RA-3 It forwards the schedule decision responsibility to an *ARTful* global scheduler instance.

RA-4 It employs the computed schedule using the runtime system semantics.

RA-1 defines a runtime adapter that integrates with one runtime system and interfaces all scheduling decisions. RA-2 states that a runtime adapter shields a scheduling context from some interactions with the runtime system by setting itself the references to data structures and functionalities of interest. RA-3 explains that runtime adapters have no responsibilities related to actually computing a schedule, leaving that to global schedulers. Nevertheless, RA-4 sets their responsibility of using the runtime system to enforce the computed schedule.

In cases where a scheduling framework is provided (e.g., Charm++ and StarPU), the runtime adapter looks like a usual user-defined scheduler. By contrast, other systems (e.g., OpenMP runtimes) must rely on alterations in the runtime library functions for expanding the system's global scheduler pool. However, since the adapter yields control to *ARTful* global schedulers, a single adapter can be used to extend the runtime system with multiple scheduling policies. This is an advantage over the current development process where, for each new scheduler addition, more code must be added inside a runtime system.

4 | MOGSLIB AS AN IMPLEMENTATION OF ARTFUL SCHEDULER SPECIFICATIONS

We developed the Metaprogramming-oriented Global Scheduling Library (MOGSLib)[13] to evaluate the performance impact of *ARTful global schedulers* when employed in real runtime systems. MOGSLib is implemented using object-oriented and template metaprogramming directives of the *C++14* language. The *ARTful* abstractions are encapsulated in classes and configured through template parameters, allowing for components to be specialized during compilation. With the objective of reducing the overall number of lines and code, MOGSLib implementations use the standard template library (STL[1]) algorithms and data structures whenever possible. As of now, MOGSLib supports the creation of global schedulers that can be specialized into centralized load balancers for Charm++ and dynamic loop schedulers for LibGOMP (GNU's OpenMP runtime). Each component in MOGSLib can be individually validated using unitary tests through googletest[28] before being integrated into the runtime system. In the remainder of this section, we present how *ARTful* components are implemented in MOGSLib and how they are connected to two runtime systems.

4.1 | MOGSLib scheduling policies

We focus on the class of centralized, workload-aware scheduling policies in this work. More specifically, we work with variations over the LPT scheduling policy (Algorithm 1 in Section 2.1). This class of scheduler makes its decisions based on knowledge over all available application and platform workload data. They are of interest to our study because (i) they are very common choices in runtime systems, (ii) they achieve quasi-linear mapping decision times with little variance, (iii) they do not require many other features or information, making it easier to focus on the important aspects of their implementation.

The scheduling policies of interest in this work are named **BinLPT**[22] and **GreedyLB**[29]. Each policy was originally specialized to function in its respective runtime system, which results in some differences between them. For instance, BinLPT schedules loop iterations to OpenMP threads, while GreedyLB schedules Charm++ chares to operating system threads or processes. Nevertheless, we will refer to loop iterations and chares as **tasks**, and to threads and processes as **processing units** (**PUs**), the latter representing resources.

BinLPT partitions the tasks in chunks (contiguous lists of tasks) through a greedy bin packing heuristic to obtain better cache locality for each PU. Later, it schedules the chunks into processing units using the LPT rule. BinLPT was designed as a loop scheduling policy for the shared memory OpenMP runtime, and it was implemented in an enhanced version of LibGOMP[15]. The library is customized to portray the BinLPT policy and a few extra functions so that users can register parallel loops and inform their iterations' workload. Our new BinLPT implementation is contained within MOGSLib and the extra required functions are implemented within the scheduling context data structures in MOGSLib, which diminishes the amount of code added to LibGOMP.

GreedyLB utilizes a *max-heap* to order tasks by their workload and a *min-heap* to order PUs by their current load. The policy iteratively designates the task atop of the *max-heap* to the processor atop of the *min-heap* until the task heap gets empty. GreedyLB is implemented in Charm++'s load balancing framework. It uses the system's *load balancer database* (LBDB) to gather dynamic data about the application and processing unit's workloads. Our version in MOGSLib preserves every aspect of this policy, but the access to the system's structures is abstracted by context data structures in order to conform to the *ARTful* specifications.

[1]The C++ STL reference can be found at https://en.cppreference.com/w/cpp/container.

Algorithm 2 The LPT scheduling policy and its template structure interface in MOGSLib.

```
1  template<typename Id, typename Workload>
2  struct LPT
3  {
4    public:
5      using Out = vector<Id>;
6      static void map(Out &map, vector<Workload> tasks, Id npus)
7      {
8        auto tasks = create_workload_heap<true>(task_loads); // max-heap of tasks
9        vector<Workloads> pu_loads(npus);          // Allocate zero-initialized entries
10       auto pus = create_workload_heap<false>(pu_loads);    // min-heap of resources
11
12       while(!tasks.empty()) // While there are tasks left to map
13       {
14         auto &task = tasks.front(); // Take the unmapped task with the largest load
15         auto &pu = pus.front();     // Take the resource with the smallest load
16
17         map[task.id] = pu.id; // Map the task to the resource
18         pu.load += task.load; // Update the load of the resource
19
20         // Remove the task from the set of tasks to be mapped
21         pop_heap(tasks.begin(), tasks.end(), Compare<Workload, true>());
22         tasks.pop_back();
23         // Update the resources' heap
24         pop_heap(pus.begin(), pus.end(), Compare<Workload, false>());
25         push_heap(pus.begin(), pus.end(), Compare<Workload, false>());
26       }
27     }
28 };
```

In MOGSLib, every scheduling policy is implemented as *C++ static functions* within *template structures* in order to follow our ARTful specifications (Section 3.2, SP-1 and SP-2). The template parameters are used to abstract system-sensitive data types allowing the policies to work with generic definitions of *workload* and *identifiers* data types (SP-5). In Charm++, workload is defined by a system type, `LBRealType`, that represents the execution walltime (in seconds) of a task in the previous application iteration. In LibGOMP, the workload data is informed by the user and represents an arbitrary numeric unit originally implemented as integers.

Algorithm 2 displays the template structure signature of a generic LPT policy in MOGSLib. In MOGSLib, every policy structure must declare its only public procedure as a static function called `map` (line 6). The first parameter is used as the procedure's global schedule output (SP-3), while the remaining parameters must represent the policy's requirements (SP-4). As such, the LPT policy requirements, as represented in the algorithm, are: (i) a collection with the tasks' workload values and (ii) the processing unit count. With the correct definition of the LPT structure template parameters, this policy implementation can be used in any runtime system capable of fulfilling the policy's requirements.

The `map` function itself follows a structure very similar to the LPT scheduling policy from Algorithm 1 (Section 2.1). The main difference between this code and the original algorithm is that we use efficient max- and min-heaps to store the loads of tasks and resources, respectively, while the original algorithm represents tasks and resources with sets, and it obtains the task and resource of interest with "arg max" and "arg min" operations. Finally, MOGSLib's organization streamlines the development and integration of schedulers by enabling developers to write scheduling policies once and in a runtime system-independent manner, to test their algorithms in isolation, and to express the context requirements for reusing a given implementation in other systems.

4.2 | MOGSLib scheduling contexts

The scheduling context is the next abstraction to be developed (if not yet available) in the creation of new scheduler following a bottom-up process. The implementation of a scheduling context depends on the global schedulers they will be used with, as suggested by the ARTful specifications SC-2 and SC-3 (Section 3.4). For instance, GreedyLB requires the number of PUs in the platform and the workload of each task in the application. These are the bare minimum requirements of workload-aware schedulers, so a generic context structure interface can be expressed in MOGSLib as illustrated in lines 1–8 in Algorithm 3. Meanwhile, BinLPT requires one additional information, which is the maximum number of chunks to be created, which can be

Algorithm 3 The scheduling context interfaces required by GreedyLB and BinLPT in MOGSLib.

```
1  template<typename tId, typename tLoad>
2  struct WorkloadCtx
3  {
4    using Id = tId;
5    using Load = tLoad;
6    Id npus() { /* ... */ }
7    vector<Load> workloads() { /* ... */ }
8  };
9
10 template<typename tId, typename tLoad>
11 struct BinLPTCtx : public WorkloadCtx<tId, tLoad>
12 {
13   Id chunks() { /* ... */ }
14 };
```

obtained from OpenMP internal data structures. A compliant interface for the BinLPT scheduling context (which will encapsulate every context-sensitive featured required as per SC-1) can be defined by extending the default `WorkloadCtx` as depicted in lines 10–14 in Algorithm 3.

Scheduling context structures must fulfill the *global scheduler* class requirements at the same time they help achieve the scheduling objectives of the runtime system where they act. For instance, Charm++ applications may contain unmigratable tasks (rigid jobs) and the platform may have unavailable PUs which cannot receive new tasks. Charm++ load balancers must filter the related input data to ignore these elements in the platform and avoid illegal mappings. Additionally, Charm++ also instruments the application in order to obtain dynamic workload measurements of each task. These measurements are the reason behind Charm++ adaptive scheduling solutions, meaning that workload-aware strategies in this system should employ these metrics as to conform to Charm++'s objectives.

Algorithm 4 showcases the scheduling context structure for centralized workload-aware schedulers in Charm++ (`CharmCentralizedWL`). The context uses Charm++'s LBDB to query the workloads of tasks and PUs. This database in Charm++ is a reference to a system-specific data structure that is accessible in the system's global namespace. It can be accessed in MOGSLib context structures through a pointer (line 3) that must be set up by an *ARTful system adapter*.

The scheduling context structure in Algorithm 4 is responsible for filtering the unavailable PUs (lines 9–14), unmigratable tasks (lines 16–20), and calculate the workload of the tasks (line 22) following Charm++'s semantics. These functionalities are implicitly delivered to the workload-aware global schedulers through their requirements, the `npus` (lines 25–29) and `workloads` (lines 31–36) functions. Since these steps are required for many Charm++ load balancers, this context integrates some Charm++ constraints into the scheduler and improves the software quality by avoiding code replication throughout scheduling policies.

In the case of LibGOMP, the automatic definition of task workloads is originally not present. The authors of BinLPT extended the OpenMP specifications to portray a new API function call named `omp_set_workload`, so that the workload could be informed by the user to the loop schedulers in the runtime system. We recreated this feature in MOGSLib within the OpenMP context structure, depicted in Algorithm 5, without the need of extending LibGOMP's API. In the `LibGOMPWorkload` context structure, the number of PUs in the environment is determined by the OpenMP API function `omp_max_threads` (lines 5–8). The number of chunks must be gathered from the runtime system's internal structure, which is set up by an *ARTful system adapter* (abstracted in lines 15–18). Finally, the workload of each task is informed by the user, who can access and manipulate the `workloads` public variable (line 3). More details on how a user can access an instance of this structure are discussed in Section 4.5.

In a later work by the authors of BinLPT [15], their custom version of LibGOMP was once again extended. A new feature called *multiloop support* was added to allow the runtime to associate a schedule calculated by BinLPT to a parallel loop. The runtime can then use the previously calculated schedule when executing the same loop again. Multiloop is used to reduce the scheduler overhead by conditionally avoiding new calculations based on user-defined criteria. This feature is specially useful in iterative applications that display little variation from one iteration to another.

This multiloop feature is not native to LibGOMP nor to the OpenMP specifications. It has been developed in this custom version of LibGOMP and is currently available only for the BinLPT scheduler. However, we envision that this is a context-sensitive scheduling functionality that could be applied to other library implementations of OpenMP and loop schedulers. We implemented this feature in MOGSLib within a specialized scheduling context structure that extends the default behavior of

Algorithm 4 MOGSLib's scheduling context implementation for centralized, workload-aware Charm++ load balancers.

```
 1 struct CharmCentralizedWL : public WorkloadCtx<unsigned, LBRealType>
 2 {
 3   LDStats *lbdb = RTS::Charm::lbdb_ref;
 4
 5   private:
 6     vector<Id> PUs, chares;
 7     vector<Load> workloads;
 8
 9     void filterPUs()
10     {
11       PUs.clear();
12       for(Id i = 0; i < lbdb->nprocs(); ++i)
13         if(lbdb->procs[i].available) PUs.push_back(i);
14     }
15
16     void filterChares()
17     {
18       for(auto i = 0; i < lbdb->n_objs; ++i)
19         if(lbdb->objData[i].migratable) chares.push_back(i);
20     }
21
22     void calculateWorkloads() { /* Omitted for simplicity */ }
23
24   public:
25     Id npus()
26     {
27       filterPUs();
28       return PUs.size();
29     }
30
31     vector<Load> workloads()
32     {
33       filterChares();
34       calculateWorkloads();
35       return workloads;
36     }
37 };
```

the LibGOMP scheduling context. Due to the *ARTful specifications*, this feature is available for all schedulers in MOGSLib when associated to this context. We strongly believe user-defined functionalities should not be implemented within runtime system libraries, as they create unnecessary complexity to an already extensive codebase. The *ARTful specifications* allow implementations to both expand the scope of a scheduling technique and incur in less alterations to the runtime system library.

4.3 | MOGSLib global schedulers

In MOGSLib, global scheduler components are implemented as *C++ template classes*. This is inspired by both Charm++ and StarPU, where schedulers are represented by classes. By comparison, the template parameter is exclusive to the MOGSLib implementation, being used to both abstract and bind the class declaration to an external data type (the scheduling context) respecting GS-6 (Section 3.3). Through this, a global scheduler class can only be realized when paired with a definition of a scheduling context data type. The global scheduler class can then use the scheduling context data type to access information and functions in order to fulfill its scheduling policies requirements and call them to calculate a schedule.

Algorithm 6 showcases the implementation of an *ARTful* global scheduler abstraction that employs the previously described LPT scheduling policy. In lines 1–2, the template class is declared along with its template parameter, representing the scheduling context `Ctx`. The template data type is used in lines 4–5 to query concrete data types for the task/processing units identifiers `Id` and the tasks' workload data type `Load` (GS-5). The *ARTful global scheduler* public procedure to decide a schedule (GS-1) is the `work` function (lines 10–20) which takes one reference to an instance of its scheduling context data type. The syntax to call the `work` function is the same for any scheduler class as the parameter for this function is tied to the template class definition (GS-4). The `work` function manipulates the context instance to call functions that must be present in its implementation. This is observed in lines 12–13, where the scheduling context is queried to obtain the tasks' workload and the PU count, respectively.

Algorithm 5 MOGSLib's scheduling context implementation for workload-aware loop schedulers in LibGOMP.

```
1  struct LibGOMPWorkload : public BinLPTCtx<unsigned, unsigned>
2  {
3    vector<Load> workloads;
4
5    Id npus()
6    {
7      return omp_max_threads();
8    }
9
10   vector<Load> workloads()
11   {
12     return workloads;
13   }
14
15   Id chunks()
16   {
17     return RTS::OpenMP::chunks;
18   }
19 };
```

Ultimately, following GS-3, the global schedule decision is forwarded to the *ARTful scheduling policy* (lines 15–17) and the output is passed on to the *scheduling context* in line 19 (GS-2).

The *C++* language allows any data type to be passed as a template parameter, and this is true for MOGSLib global scheduler classes too. However, the compilation will only succeed as long as the template parameters has definitions for all types and functions required within the global scheduler class (ARTful Scheduling Context specification SC-4). This purposefully creates a binding between the two classes where the global scheduler expresses its dependencies and the scheduling contexts fulfills them. Therefore, the global scheduler class is capable of abstracting the runtime system at the same time it express its dependencies in a small segment of code, which also makes it possible to reuse a global scheduler in a different runtime system by changing only its context.

4.4 | MOGSLib runtime library adapters

In order to expose MOGSLib components to the runtime system, we need to create a native global scheduler that acts as a proxy in the runtime system (RA-1, Section 3.5). The adapter serves as a layer of abstraction, isolating the system from the schedulers and vice versa. Each adapter must be developed within the runtime system and, once it is available, multiple MOGSLib global

Algorithm 6 A MOGSLib global scheduler that employs the LPT scheduling policy.

```
1  template<typename Ctx>
2  class LPTScheduler
3  {
4    using Id = typename Ctx::Id;
5    using Workload = typename Ctx::Load;
6
7    using P = LPT<Id, Workload>;
8    using Out = typename P::Out;
9
10   void work(Ctx& ctx)
11   {
12     auto workloads = ctx.workloads();
13     auto npus = ctx.npus();
14
15     Out out = Out();
16     out.resize(size(workloads));
17     P::map(out, workloads, npus);
18
19     ctx.set_schedule(out);
20   }
21 };
```

schedulers can be exposed through a single adapter. Indeed, an immediate benefit of this design is the reduction of the total code alterations in the runtime system library needed to implement multiple user-defined scheduling solutions.

In Charm++, the development of user-defined centralized load balancers is achieved by extending its `BaseLB` class. Scheduler developers must encapsulate the policy logic and interaction with the system within the derived class and provide an implementation to a virtual `work` function. Indeed, our MOGSLib runtime adapter for Charm++, named `MOGSLibLB`, follows this standard approach to integrate new scheduling solutions in Charm++. Algorithm 7 showcases the `work` function of our runtime adapter that can be used for centralized, workload-aware scheduling policies. It initializes a static reference to the system's load balancing database (line 3), enabling MOGSLib scheduling contexts to manipulate it as previously seen in Algorithm 4 (RA-2). Later, it calls a function pointer in MOGSLib API, `selected_scheduler`, that decides which scheduler class will be executed. This function's default behavior is to call the first scheduler class informed in the precompilation script, however, user functions can be assigned to this pointer for deciding the scheduler. In line 6, the control is forwarded to the `work` method of the selected *ARTful global scheduler* class in MOGSLib, which will calculate the schedule using the data in the scheduling context (RA-3). Finally, the adapter assigns the schedule using Charm++'s API (lines 11–13) in accordance with RA-4.

Algorithm 7 Main function (`work`) in MOGSLib's runtime adapter for Charm++ centralized load balancers.

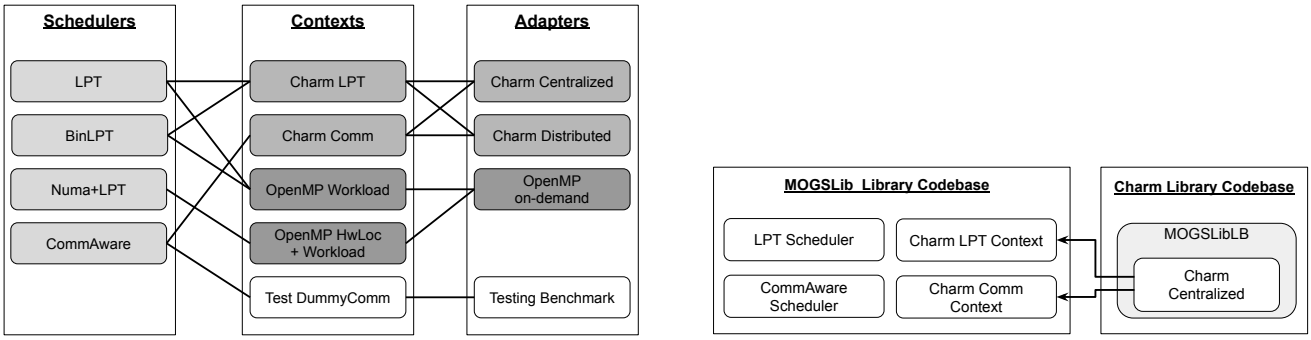
```
1  void MOGSLibLB::work(LDStats* stats)
2  {
3    MOGSLib::RTS::Charm::lbdb_ref = stats;
4
5    auto strategy = MOGSLIB::API::selected_scheduler();
6    auto map = MOGSLib::API::work(strategy);
7
8    auto &chare_ids = MOGSLib::RTS::Charm::chare_ids;
9    auto &pu_ids = MOGSLib::RTS::Charm::pu_ids;
10
11   auto i = 0;
12   for(auto chare : chare_ids)
13     stats->assign(chare, pu_ids[map[i++]]);
14 }
```

The development of user-defined loop schedulers for LibGOMP is different. Without explicit support from the library, users must manually integrate new policies by modifying internal structures that handle the loop scheduling. These structures are presented as a set of functions responsible for passing control from the runtime system to the selected scheduling policy when a parallel loop is detected. The creation of a MOGSLib adapter for LibGOMP is achieved through the modifications in two LibGOMP functions, `gomp_loop_runtime_start` and `gomp_loop_init`. The first function is executed when the application reaches an OpenMP `parallel for` construct and it decides which policy will execute. In order to add MOGSLib as a loop scheduler, a new option is added to the `gomp_loop_runtime_start` internal decision mechanism, the `mogslib` policy. The other function, `gomp_loop_init`, is called before the parallel loop starts and it is where static loop schedulers decide their schedule. This function is altered to call MOGSLib and apply its calculated schedule when the `mogslib` policy is chosen. Another function is also added to the LibGOMP implementation, the `gomp_iter_mogslib_next`. The `gomp_iter_*_next` functions are called when a thread finishes performing its work pool. The added function in the adapter serves the purpose of dynamically stealing chunks from other threads as to minimize the dynamic overhead of static scheduling.

The aforementioned MOGSLib runtime adapter is based on the adaptations required to execute the BinLPT scheduler[22]. As such, MOGSLib schedulers exposed to LibGOMP through this adapter apply a static schedule and portray a dynamic balancing phase where chunks are stolen from overloaded threads. In contrast to other OpenMP user-defined schedulers, these alterations are enough so that LibGOMP can call any policy developed within MOGSLib, avoiding extra costs of integrating different strategies to the library.

4.5 | MOGSLib assembling tools

The *ARTful* specifications guides the development of generic components that can be composed to form a specialized global scheduler. Indeed, a global scheduler library that follows the *ARTful* specifications is comprised of multiple separate components



(a) Before compilation. (b) After compilation.

FIGURE 3 Hypothetical interaction among MOGSLib components before and after the compilation process.

as depicted in Figure 3. However, there must be a streamlined way to compose the abstractions and attach them to the runtime system, so they can perform as native global schedulers without requiring additional efforts from developers.

MOGSLib scheduling solutions are configured by a combination of a *global scheduler class*, a *scheduling context structure* and a *runtime adapter*. Some abstractions may not fit together and a solution is only valid if a *global scheduler template class* can be compiled given a *scheduling context structure* template parameter. This validation at compile time is available due to the relationship between the two classes discussed in Section 4.3. MOGSLib uses a precompilation script to query the user about the data types that will be composed into a set of global schedulers. The user must inform the path of at least three *C++* header files with the template definitions of: (i) an *ARTful global scheduler* class; (ii) an *ARTful scheduling* structure; and (iii) the *ARTful runtime adapter* implementation for the desired system. This information enables MOGSLib to create the necessary links between the data structures and create a static library API that exposes its global schedulers. The generated library API is a *C++* header file which must be attached and compiled alongside the runtime system or application.

An overview of the library after the precompilation and compilation process is depicted in Figure 3(b). In this figure, MOGSLib exposes two hypothetical strategies, the *LPT* and *CommAware scheduling policies*. Those policies are each supported by different contexts: (i) `CharmLPT`, a structure that queries the workload of the tasks from Charm++ LBDB and (ii) `CharmComm`, a structure that queries the LBDB to gather the tasks' communication data. Both of these *scheduling solutions* are available in Charm++ through a runtime adapter class `CharmCentralized`, which makes Charm++ functionalities available in MOGSLib context structures.

After the development of *ARTful* schedulers using MOGSLib, the only thing remaining is for the user to employ them to improve application performance. As the overhead of using a scheduling library external to the runtime system can be intimidating for new users, we explore different experimental scenarios in the next section to demystify this.

5 | EXPERIMENTS WITH MOGSLIB SCHEDULERS AND NATIVE COUNTERPARTS

We organized a set of experiments in order to quantify the effects of our global scheduling library MOGSLib (based on our *ARTful* specifications) instead of the global schedulers native to runtime systems. The objective of these experiments are: (i) to measure the scheduling overhead (i.e., the time it takes to compute a schedule during the execution of an application) with and without using MOGSLib for the same scheduling algorithms; and (ii) to measure the impact on the total execution time of applications when using MOGSLib or the equivalent schedulers native to the runtime systems.

The experiments include the use of MOGSLib in two different scenarios: one with Charm++ and the other with LibGOMP. On Charm++, we compare the implementation of GreedyLB in MOGSLib to its original Charm++ version using a synthetic benchmark (*lb_test*) and a molecular dynamics benchmark (*LeanMD*). Meanwhile, for LibGOMP, we compare the implementation of BinLPT in MOGSLib and its original version using a synthetic benchmark (*SchedCost* based on characteristics of LULESH[30]) and a molecular dynamics benchmark (*LavaMD*). Additionally, we evaluate the performance of the multiloop feature introduced by BinLPT. Finally, these experiments include variations in the number of tasks to be scheduled (problem size) because they directly affect the execution time of global schedulers.

All comparisons where based on a confidence threshold of 95% (significance of 5%) for the different statistical methods used. We start our evaluations by checking if the samples follow normal distributions (Kolmogorov-Smirnov test) and if they have the same variance (F test). If we do not reject the null hypothesis in any tests (i.e., all p-values > 0.05), then we use parametric methods for our comparisons (Welch Two Sample t-test), or else we have to use nonparametric methods (Mann-Whitney U test). For both kinds of methods, a p-value < 0.05 means that we reject the null hypothesis that the compared versions perform the same (in other words, they perform differently).

The execution platform for all experiments is the *Ecotype* cluster from the Grid5000 distributed environment[2]. All Charm++ tests are deployed on four nodes whereas the OpenMP tests run over a single shared memory node. Each node in *Ecotype* includes two Intel Xeon E5-2630L v4@1.80GHz processors (10 cores per CPU with hyper-threading) and 128GB of DDR3 memory. They are interconnected using a Gigabit Ethernet network @10Gbps. The nodes run the Debian 9 operating system and the compiler used throughout the experiments is g++ v7.3.0. We used Charm++ v6.9.0[3] and the custom version of LibGOMP used in this paper is publicly available online with both the original BinLPT and MOGSLib schedulers[4].

In the following sections, we discuss the details and results of the experiments with synthetic benchmarks, with the multiloop feature, and with the molecular dynamics benchmarks.

5.1 | Experiments with synthetic benchmarks

Synthetic benchmarks are best suited for abstracting application details, so we use them to assess the *schedule decision cost* (or *scheduler overhead*) of the schedulers based solely on the volume of data processed (number of tasks). In other words, we measure the scheduling overhead in all scenarios and analyze the impact of MOGSLib.

5.1.1 | lb_test on Charm++

lb_test is a load balancing synthetic benchmark distributed with Charm++ whose tasks perform dummy floating point operations. It supports multiple configuration parameters, but most of the parameters have no impact on the scheduler overhead. We fixed the benchmark parameters to simulate tasks in a 2D mesh or ring topology for 150 iterations, and to call a load balancer every 40 iterations. The tasks were set with fixed workloads that vary from $10\mu s$ to $3000\mu s$ per iteration, and they run over the 80 available processing units (160 virtual cores over 4 compute nodes). The number of tasks was varied from 800 to 3200 in steps of 800, and additional scenarios with 8000 and 16000 were tested. For each number of tasks and scheduler version, lb_test was executed 20 times for a total of 60 load balancing calls.

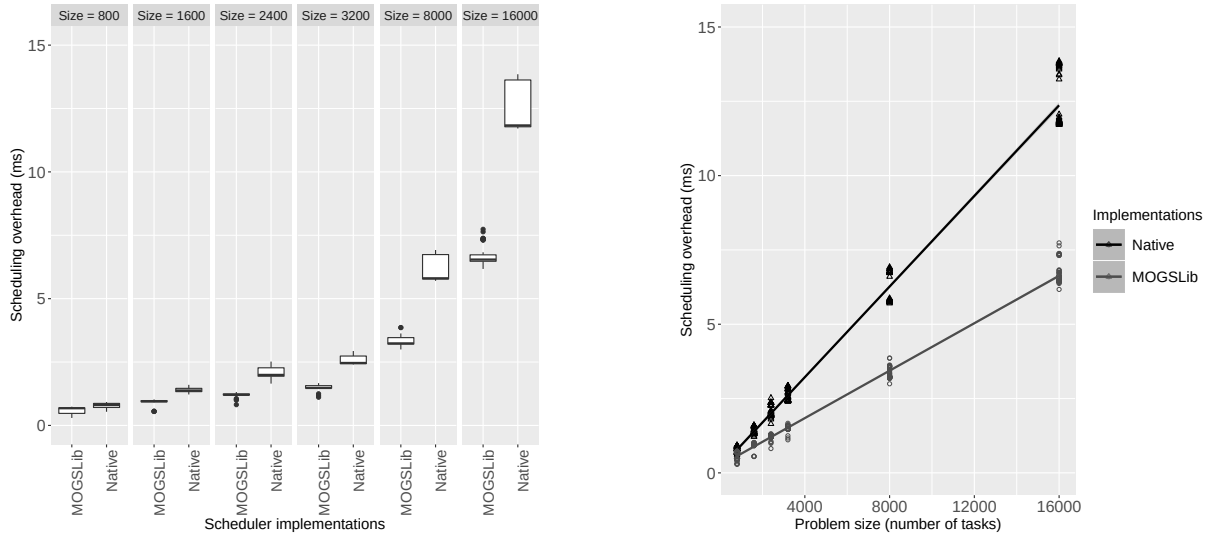
Figure 4(a) displays the collected scheduler overhead for the different scenarios as boxplots[5]. The horizontal axis showcases the different application sizes, each portraying the data for both implementations of the GreedyLB load balancer, and the vertical axis represents the measured times in milliseconds. As we can see in Figure 4, MOGSLib shows a smaller overhead than its native counterpart, and the difference seems to be more noticeable as we increase the number of tasks to be scheduled. Indeed, for all cases, MOGSLib shows a different and smaller time (all Mann-Whitney U tests returned p-values < 0.05). Upon further investigation, we concluded that the smaller overhead of MOGSLib comes from its smart use of STL algorithms and data structures.

Figure 4(b) showcases the linear regression of these scenarios as to extrapolate the curve in our analyzed range. This analysis suggests that MOGSLib version scales better than the native Charm++ version. More specifically, although we see a higher base overhead, the MOGSLib version has coefficient value 47% smaller for the predicted schedule decision time.

5.1.2 | SchedCost on LibGOMP

SchedCost simulates an iterative application where each of its N iterations are composed of one parallel loop as depicted on Algorithm 8. The schedule decision cost (`cost`) is calculated in line 10 based on the difference between the time when the first instruction within the loop is computed (`fi` in line 7) and the time right before the entering the parallel loop (`pi` in line 3). The sum of the schedule decision costs of N iterations represents the total scheduler overhead for the application (line 11).

[2]Information about Grid5000 is available at https://www.grid5000.fr/mediawiki/index.php/.
[3]Information about Charm++'s software is available at http://charm.cs.illinois.edu/software.
[4]Code available on GitHub at https://github.com/alexandrelimassantana/libgomp.
[5]Boxes extend from the 1st to 3rd quartiles of the samples. Lines represent the median values. Whiskers represent the data within 1.5 IQR from the lower or upper quartile.



(a) Scheduling overhead as boxplots.

(b) Scheduling overhead as points with linear regressions.

FIGURE 4 GreedyLB scheduler overhead on its native form and in MOGSLib for experiments with lb_test.

Algorithm 8 Computation of the schedule decision cost in OpenMP by SchedCost.

```
1 void schedcost(int N) {
2   for(int i = 0; i < N; i++) {
3     int pi = TIME();
4     #pragma omp parallel for
5     for(int j = 0; j < N; j++) {
6       if(j < nthreads)
7         fi[omp_get_thread_num()] = TIME();
8       C[j] = A[j] + B[j];
9     }
10    cost[i] = MAX(fi)-pi;
11    ovh += cost[i];
12  }
13}
```

We selected the SchedCost parameters in order to simulate the same volume of input data of a small and medium use case of the LULESH application[30]. LULESH is an iterative proxy hydrodynamics shock application where each iteration contains multiple parallel loops to calculate various particle interactions. LULESH has parallel loops with large iteration counts and the simulation spans over multiple discrete steps. In this context, the parameters for SchedCost were derived from the application's usage guidelines. They showcase two scenarios: (i) a small test case with 937 parallel loop calls with 30^3 iterations (tasks) each and (ii) a medium use case with 1477 parallel loop calls with 45^3 iterations each. For each number of tasks and scheduler version, SchedCost was executed 100 times in a single shared memory compute node (40 OpenMP threads).

Figure 5 shows the scheduling overhead for different scenarios as boxplots. The horizontal axis showcases the different application sizes (small test on the left and medium test on the right), each portraying the data for both implementations of BinLPT, and the vertical axis represents the accumulated scheduling overhead in milliseconds. The boxplots look mostly like horizontal lines because the measured times were all very close to their median. Also, we can see that the overhead with MOGSLib is smaller than its native counterpart, and that was confirmed by our statistical analysis (both Mann-Whitney U tests returned p-values < 0.05).

In our experimental setup, the average scheduling overhead for the small and medium use cases for the BinLPT native scheduler are 283 and 1534 ms, respectively. Meanwhile, the MOGSLib implementation achieved average scheduling overheads of 117 and 514 ms. This represents speedups of 2.42 and 2.98 for the small and medium cases. As such, we conclude that the portability of MOGSLib does not have to worsen the performance of global schedulers on Charm++ and OpenMP.

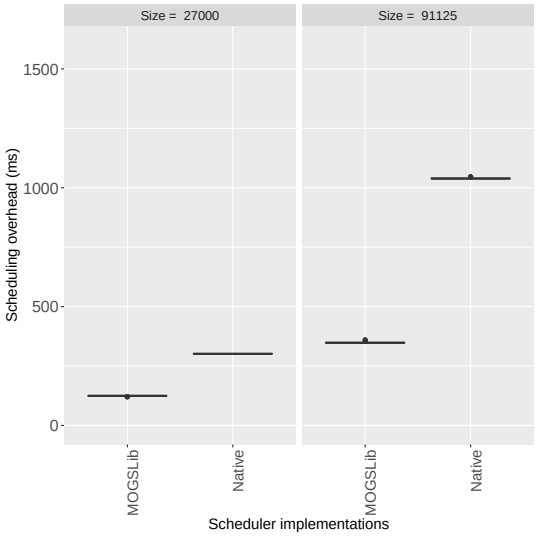


FIGURE 5 Boxplot of BinLPT's scheduling overhead on its native form and in MOGSLib for experiments with SchedCost.

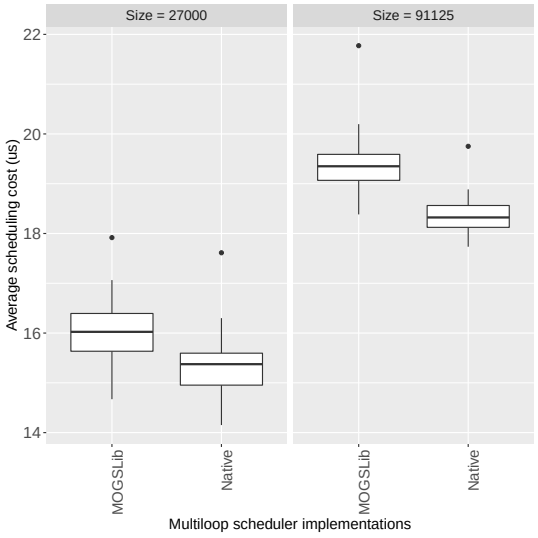
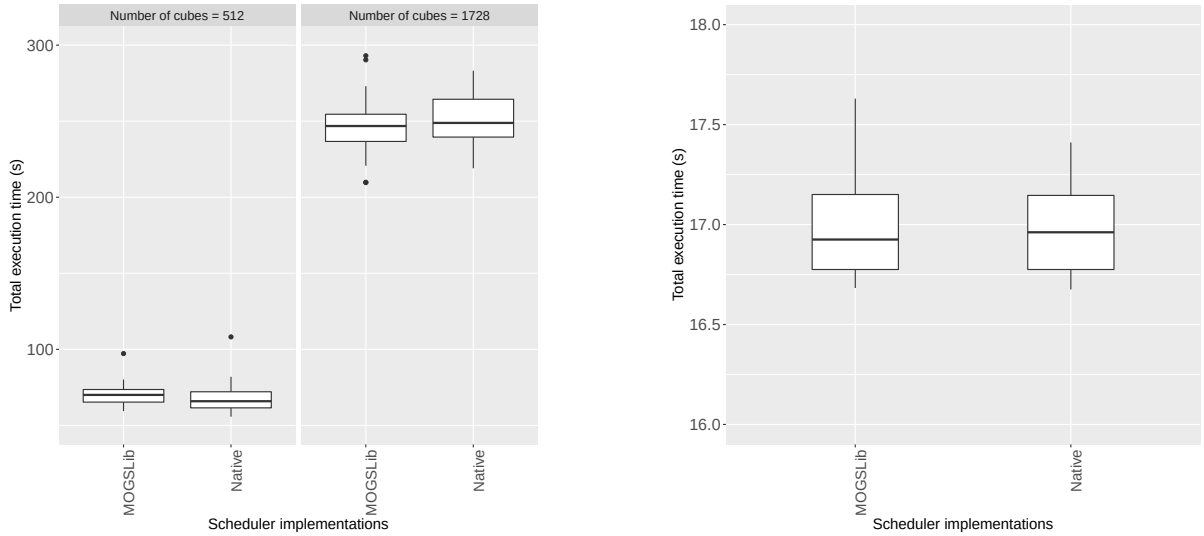


FIGURE 6 Boxplot of BinLPT's schedule cost on its native form and in MOGSLib for multiloop experiments with SchedCost. The vertical axis start at 14 μs to emphasize the difference between scheduler versions.

5.1.3 | Multiloop support

After the experiments with BinLPT without the multiloop feature from the previous section, we experimented with this feature by implementing it out of the runtime library scope as a MOGSLib scheduling context feature. We reused the SchedCost benchmark with the same parameters to assess possible performance differences between both implementations. We configured both schedulers to calculate the schedule once and reuse the same schedule for the remaining loop executions.

Figure 6 presents the schedule cost for different multiloop scenarios as boxplots. It differs from Figure 5 as it is based on the average cost in microseconds for the scheduler calls on each execution of the benchmark (instead of the accumulated scheduling overhead). In this scenario, we discarded the null hypothesis that both implementations achieve the same performance (Welch's t-tests returned p-values < 0.05), meaning that the native implementation performs better than the MOGSLib implementation. Nevertheless, as the difference was around 1 μs on average, we believe that this different is still irrelevant to the overall performance of applications.



(a) Times for LeanMD using GreedyLB on Charm++.

(b) Times for LavaMD using BinLPT on LibGOMP.

FIGURE 7 Total execution time when using native and MOGSLib schedulers. The vertical axes start at different points to emphasize the difference between scheduler versions.

5.2 | Experiments with molecular dynamics benchmarks

Our analysis with synthetic benchmarks suggested that MOGSLib schedulers perform their decisions similarly or faster than their versions native to the runtime systems studied. However, this assertion does not mean that MOGSLib can reduce the application makespan more just by simply employing a different implementation of a scheduling policy. Both versions of a scheduler compute the exact same schedule and, thus, they should have a similar influence on application performance.

Molecular dynamics benchmarks are known to display dynamic load imbalance due to the way particles disperse and interact through the simulated 3D space. Therefore, we use these benchmarks to assess the impact of the schedulers on the *total execution time* of applications.

5.2.1 | LeanMD on Charm++

LeanMD[31] is a Charm++ mini-app that calculates the force interactions among particles using the Lennard-Jones potential computation[6]. Besides its default parameters, we fixed the benchmark to run for 300 iterations, and to call a load balancer after 20 iterations and every 100 iterations after that. We set a small and a medium use cases by simulating the space with 8^3 and 12^3 boxes, respectively. LeanMD was executed 30 times for each number of tasks and scheduler version.

Figure 7(a) portrays the distribution of total execution times for the different scenarios as boxplots. The left boxplot is associated with the small use case (8^3 cubes) and the one to the right with the medium use case (12^3 cubes). The vertical axis represents the time measured in seconds.

As we can see in this figure, the total execution times with MOGSLib and with the native Charm++ implementation of GreedyLB are very similar. Indeed, our statistical analysis indicates that we cannot reject the hypothesis that both results originate from the same distribution — i.e., they perform the same (both Mann-Whitney U tests returned p-values > 0.05). This result shows us that the MOGSLib implementation, despite the better performance on deciding the schedule shown in Section 5.1.1, does not tend to affect the total execution time of applications.

5.2.2 | LavaMD on LibGOMP

LavaMD[32] is a benchmark with an OpenMP version that simulates the pressure-induced solidification of molten tantalum and uranium atoms[7]. For these experiments, we recreated a subset of the experiments done by Penna et al.[15] where the BinLPT

[6]LeanMD is available at http://charmplusplus.org/miniApps/.
[7]LavaMD is available at https://rodinia.cs.virginia.edu/.

scheduling policy was thoroughly evaluated. We configured LavaMD to decompose the 3D domain into 11^3 cubes. Each cube contains a random number of particles generated through an exponential distribution with $\gamma = 0.2$. BinLPT was configured to generate up to 80 chunks of tasks, and each configuration was executed 100 times.

Figure 7(b) portrays the total execution times of LavaMD (in seconds in the vertical axis) with the two implementations of BinLPT as boxplots. As in the case for LeanMD (Figure 7(a)), the median times for both implementations were very similar (close to 17 s), and again our statistical analysis indicates that both versions of BinLPT perform the same (Mann-Whitney U test returns a p-value > 0.05). In other words, the benefits of MOGSLib do not come at a cost of slower scheduling decisions nor increased application execution times.

6 | CONCLUSIONS

In this work, we discussed the current state of scheduling software, and proposed the *ARTful scheduling specifications* in order to build more adaptable and generic global schedulers implementations that can be used in multiple runtime systems. Our proposed design allows for the development of user-defined schedulers that can be specialized to work within different runtime systems. We envision that this approach can improve the code quality by isolating schedulers from the remainder of the runtime system library code.

We showcased an implementation of our specifications through MOGSLib, our global scheduler library and development framework. MOGSLib employs object-oriented and generic metaprogramming directives in order to express schedulers and their relationship to the runtime system. We recreated two centralized, workload-aware scheduling solutions in MOGSLib (BinLPT used in OpenMP, and GreedyLB used in Charm++). We integrated these schedulers indirectly to their respective runtime systems and compared the scheduler overhead of our implementation in comparison to their system native counterparts. Synthetic benchmark experiments showcased that our implementations can sometimes decide the schedule even faster than their original implementations. However, these performance variations were not relevant enough to alter the total execution times of molecular dynamics applications. We envision that libraries like MOGSLib, where components are adapted to work along the remainder of the software stack, should become a standard in future HPC runtime systems.

We gradually rewrote MOGSLib to pinpoint what caused the performance gains of our schedulers in comparison to the native implementations in Section 5.1. Our additional analysis suggests that the performance improvements originate from the use of the *C++* STL algorithms in favor of a custom implementation. When using the original implementations for sorting algorithms and data structures, both versions of the same scheduler obtained similar performance. Therefore, we consider the use of STL algorithms as an advantage of MOGSLib schedulers acting as LibGOMP loop schedulers, as LibGOMP is built using the C language and has no direct access to STL or other modern tools of *C++*. As the logic of MOGSLib policies are developed in *C++* and linked to LibGOMP, MOGSLib could extend the tools available for developing loop schedulers in LibGOMP. Indeed, this observation does not hold true for all runtime systems and, in fact, Charm++ load balancers are already developed in *C++*. Our conclusion is that MOGSLib is specially well-suited for runtime systems that offer little customization support for the development of user-defined global schedulers.

Our future work aims towards the integration with new runtime systems, the development of new global schedulers, and the evolution to handle global schedulers in runtime systems that employ task-based execution models. MOGSLib also can be expanded to handle distributed scheduling solutions or communication-aware policies. Ultimately, MOGSLib is a work in progress that represents an alternative take on developing global schedulers that can be expanded in many ways.

ACKNOWLEDGMENT

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by INRIA and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). This work was partially supported by the *Conselho Nacional de Desenvolvimento Científico e Tecnológico* – Brasil (CNPq) under the Universal Program (grant number 401266/2016-8) and by the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* – Brasil (CAPES) under the Capes-PrInt Program (grant number 88881.310783/2018-01).

References

1. Dagum L, Menon R. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 1998; 5(1): 46–55.

2. Walker DW, Dongarra JJ. MPI: A standard message passing interface. *Supercomputer* 1996; 12: 56–68.

3. Blackford LS, Petitet A, Pozo R, et al. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software* 2002; 28(2): 135–151.

4. Dongarra J, Beckman P, Moore T, et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications* 2011; 25(1): 3–60.

5. Thoman P, Dichev K, Heller T, et al. A taxonomy of task-based parallel programming technologies for high-performance computing. *Springer Journal of Supercomputing* 2018; 74(4): 1422–1434.

6. Acun B, Langer A, Meneses E, et al. Power, reliability, and performance: One system to rule them all. *IEEE Computer* 2016; 49(10): 30–37.

7. Augonnet C, Thibault S, Namyst R, Wacrenier PA. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience* 2011; 23: 187–198.

8. Kale V, Gropp WD. A user-defined schedule for OpenMP. In: . 11. Proceedings of the 2017 Conference on OpenMP. ; 2017; Stonybrook, New York, USA: 2017.

9. Kale V, Iwainsky C, Klemm M, Müller Kordörfer JH, Ciorba FM. Towards A Standard Interface for User-Defined Scheduling in OpenMP. In: Proceedings of the International Workshop on OpenMP (iWomp). ; 2019; Auckland.

10. Bak S, Guo Y, Balaji P, Sarkar V. Optimized Execution of Parallel Loops via User-Defined Scheduling Policies. In: . 48. Proceedings of the International Conference on Parallel Processing. ACM; 2019; Kyoto, Japan: 38:1–38:10

11. Grossman M, Kumar V, Vrvilo N, Budimlic Z, Sarkar V. A pluggable framework for composable HPC scheduling libraries. In: Proceedings of the International Parallel and Distributed Processing Symposium Workshops. IEEE; 2017; Orlando, FL, US: 723–732.

12. Aumage O, Bigot J, Coullon H, Pérez C, Richard J. Combining both a component model and a task-based model for hpc applications: a feasibility study on gysela. In: Proceedings of the International Symposium on Cluster, Cloud and Grid Computing (CCGrid). IEEE/ACM; 2017; Madrid, Spain: 635–644.

13. Santana A, Freitas V, Pilla LL, Castro M, Méhaut JF. Reducing Global Schedulers Complexity through Runtime System Decoupling. In: Proceedings of the Brazilian Symposium on High Performance Computing Systems (WSCAD). IEEE; 2018; São Paulo, Brazil: 38–44.

14. Casavant TL, Kuhl JG. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.* 1988; 14: 141–154.

15. Penna PH, A. Gomes AT, Castro M, et al. A Comprehensive Performance Evaluation of the BinLPT Workload-aware Loop Scheduler. *Concurrency and Computation: Practice and Experience* 2019: e5170. doi: 10.1002/cpe.5170

16. Pilla LL, Ribeiro CP, Coucheney P, et al. A topology-aware load balancing algorithm for clustered hierarchical multi-core machines. *Future Generation Computer Systems* 2014; 30: 191–201.

17. Jeannot E, Meneses E, Mercier G, Tessier F, Zheng G. Communication and Topology-aware Load Balancing in Charm++ with TreeMatch. In: Proceedings of the International Conference on Cluster Computing (CLUSTER). IEEE; 2013; Indianapolis, United States.

18. Padoin EL, Diener M, Navaux POA, Méhaut J. Managing Power Demand and Load Imbalance to Save Energy on Systems with Heterogeneous CPU Speeds. In: Proceedings of the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). IEEE; 2019: 72-79

19. Unat D, Dubey A, Hoefler T, et al. Trends in data locality abstractions for HPC systems. *IEEE Transactions on Parallel and Distributed Systems* 2017; 28(10): 3007–3020.

20. Mollison MS, Anderson JH. Bringing theory into practice: A userspace library for multicore real-time scheduling. In: Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE; 2013; Philadelphia, PA, US: 283–292.

21. Frasca M, Madduri K, Raghavan P. NUMA-aware graph mining techniques for performance and energy efficiency. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC). IEEE Computer Society; 2012; Salt Lake City, Utah.

22. Penna P, Castro M, Plentz P, Freitas H, Broquedis F, Méhaut JF. BinLPT: A Novel Workload-Aware Loop Scheduler for Irregular Parallel Loops. In: Proceedings of the Brazilian Symposium on High Performance Computing Systems. IEEE; 2017; Campinas, Brazil.

23. Durand M, Broquedis F, Gautier T, Raffin B. An efficient openmp loop scheduler for irregular applications on large-scale numa machines. In: International Workshop on OpenMP (iWomp). Springer; 2013; Camberra, Australia: 141–155.

24. Bhatele A, Fourestier S, Menon H, Kale LV, Pellegrini F. Applying graph partitioning methods in measurement-based dynamic load balancing. tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA; Lawrence Livermore National Laboratory: 2011.

25. Fattebert JL, Richards D, Glosli J. Dynamic load balancing algorithm for molecular dynamics based on Voronoi cells domain decompositions. *Computer Physics Communications* 2012; 183(12): 2608-2615.

26. Mei C, Sun Y, Zheng G, et al. Enabling and Scaling Biomolecular Simulations of 100 Million Atoms on Petascale Machines with a Multicore-optimized Message-driven Runtime. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE/ACM; 2011; Seattle, USA: 61:1-61:11.

27. Freitas V, Santana A, Castro M, Pilla LL. A Batch Task Migration Approach for Decentralized Global Rescheduling. In: Proceedings of the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). IEEE Computer Society; 2018; Lyon, France: 1-12.

28. Sen A. A quick introduction to the Google C++ Testing Framework. *IBM DeveloperWorks* 2010; 20: 1–10.

29. Zheng G, Bhatelé A, Meneses E, Kalé LV. Periodic hierarchical load balancing for large supercomputers. *The International Journal of High Performance Computing Applications* 2011; 25(4): 371-385.

30. Karlin I, Bhatele A, Keasler J, et al. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In: Proceedings of the International Parallel & Distributed Processing Symposium (IPDPS). IEEE; 2013; Boston, USA.

31. Mehta V. *LeanMD: A Charm++ framework for high performance molecular dynamics simulation on large parallel machines*. PhD thesis. University of Illinois at Urbana-Champaign, 201 North Goodwin Avenue, Urbana, IL 61801-2302; 2004.

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

How to cite this article: A. Santana, V. Freitas, M. Castro, L. Pilla, and JF. Méhaut (2020), ARTful: A specification for user-defined schedulers targeting multiple HPC runtime systems, *XXXXXXXXXXXXXXXXXX*, *20XX;00:1–XX*.