ARTful: A model for user-defined schedulers targeting multiple high-performance computing runtime systems

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

ARTful: A model for user-defined schedulers targeting multiple high-performance computing runtime systems*

Alexandre Santana[1], Vinicius Freitas[1], Márcio Castro[1], Laércio L. Pilla[2,3], and Jean-François Méhaut[4]

[1]Federal University of Santa Catarina (UFSC), Florianópolis, Brazil
[2]Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400, Talence, France – email: laercio.lima-pilla@labri.fr
[3]INRIA, LaBRI, UMR 5800, F-33400, Talence, France
[4]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, Grenoble, France

Abstract

Global schedulers are components in parallel runtime libraries that distribute the application's workload across physical resources. More often than not, applications showcase dynamic load imbalance and require customized scheduling solutions to avoid wasting resources. Some libraries lack support for user-defined schedulers and developers resort to unofficial extensions that are harder to reuse and maintain. We propose a global scheduler software design, entitled *ARTful model*, to create user-defined solutions with minimal alterations in the runtime library. Our model uses a component-based design to separate components from the runtime library and the scheduling policy implementation. The *ARTful model* describes the interface of a portable scheduler library, allowing policies to operate on different runtime libraries. We study the overhead induced by our design through our ARTful library implementation MOGSLib using workload-aware scheduling policies. We experiment with two different policies from OpenMP and Charm++ runtime systems, also presenting evaluations of the policies outside of their original library context. We observe that our portable schedulers can sometimes perform decisions faster than their native counterparts with negligible overhead in the execution times of synthetic applications and molecular dynamics kernels.

1 Introduction

High Performance Computing (HPC) applications are built atop of long-lasting standards created by the collaborative efforts of industry and academia such as OpenMP [9], MPI [28] and BLAS [6]. Thanks to these standard libraries, HPC applications may achieve better performance and portability across different platforms. However, these tools alone do not provide the best performance out-of-the-box to all problems and/or systems.

Resource management is paramount to both portability and performance of parallel applications. A well-known technique to improve resource management is *global scheduling*, which aims at (re-)distributing tasks of parallel applications over physical resources, allowing predictable

*Author's accepted version. Definitive version available at `http://doi.org/10.1002/spe.2977`

application performance in different systems [10]. Although a multitude of scheduling solutions have been proposed [26], most runtime libraries have limited support for user-defined schedulers [2, 3]. Additionally, to the best of our knowledge, there is no interoperability between solutions implemented in current runtime libraries.

Future predictions suggest a larger, more diverse HPC ecosystem, with novel hardware architectures [10]. The success of future exascale computers relies, partially, on existing tools for resource mapping and application scheduling. As diversity increases, most of the complexity to orchestrate these resources will be absorbed by standards implemented in runtime libraries. However, even today, integrating novel scheduling solutions to these libraries relies on unofficial and hard to maintain initiatives [15]. Indeed, efforts to create standards for user-defined scheduling can already be observed in OpenMP [14, 5]. While such propositions have been proven useful for discussing the future of HPC standards, there is a lack of efforts to implement these mechanisms into existing runtime libraries.

We propose a framework to create user-defined scheduling solutions that are portable by design, the *ARTful scheduler library model*. ARTful abstracts scheduling policies and their functional requirements as components [18], loosely connected by blackboard entities that compose the library interface. The model disconnects the runtime library scheduling features from the policy algorithm, exposing dependencies as component requests. ARTful allows portable schedulers to specialize to the runtime library, allowing a single user-defined policy implementation to operate on different runtime libraries. We conceive that ARTful is helpful for teams developing runtime libraries, researchers that want to integrate their scheduling algorithms to different runtimes, users that require different schedulers for different situations, among others.

Besides explaining the ideas behind the ARTful scheduler library model, we provide experiments to analyze the model overhead when compared to native solutions in OpenMP and Charm++ runtime libraries. In addition, we analyze how scheduling policies can be adapted to, and how they perform, outside of their original libraries. The ARTful model stems from our previous work on MOGSLib, a portable scheduler library [25]. We continue our previous work by adding new contributions in the form of the following topics:

1. The *ARTful scheduler library model* based on the software architecture implemented in our portable scheduler library;
2. New scheduling overhead experiments using molecular dynamics benchmarks comparing our portable schedulers with runtime native solutions;
3. The first experimental analysis of libGOMP and Charm++ scheduling solutions applied outside of their intended context.

The remainder of this paper is organized as follows. Section 2 describes the background in global scheduling software development as well as our research problem. Next, in Section 3, we present the *ARTful scheduler library model*. In Section 4, we briefly explain how we implemented the *ARTful model* in our library. We then discuss our experiments and results in Section 5 and conclude our work in Section 6.

2 Problem Definition

Global scheduling may be described as the problem of defining *where* to run a set of *tasks*, leaving the decision of *when* to run them to local scheduling [7]. This definition is broad enough to encompass different scheduling activities in modern parallel programming models such as *load balancing*, *topology mapping* and *loop scheduling*.

```
1: procedure LPT(Tasks T, Task loads (processing times) L_T, Resources R)
2:     L_R ← 0                                   ▷ The resources start with no load
3:     T' ← T                                    ▷ List of tasks to be mapped
4:     M ← ∅                                     ▷ The mapping of tasks to resources starts empty
5:     while T' ≠ ∅ do                           ▷ While there are tasks left to map
6:         t ← arg max_{t'∈T'} L_T(t')           ▷ Take the unmapped task with the largest load (largest
   processing time)
7:         r ← arg min_{r'∈R} L_R(r')            ▷ Take the resource with the smallest load
8:         M(t) ← r                              ▷ Map the task to the resource
9:         L_R(r) ← L_R(r) + L_T(t)              ▷ Update the load of the resource
10:        T' ← T' \ {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
```

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

The **scheduling policy** is an algorithm for deciding the mapping of tasks to resources, given specific objectives. Scheduling policies are unbounded by technological aspects of runtime libraries, system architecture, and parallel programming models. To emphasize this concept, we present the *Largest Processing Time First (LPT)* scheduling policy [11] in Algorithm 1. This list scheduling algorithm tries to reduce the *makespan* by iteratively taking an unmapped task with the largest load and mapping it to the resource with the smallest load.

The LPT algorithm does not define implementation aspects like: (i) what is a task (e.g., object, thread, process or loop iteration); (ii) what is a resource (e.g., core, processor or compute node); or (iii) how a workload is measured (e.g., static model or introspection). These characteristics denote *how* and *where* policies are implemented, being important practical aspects of the taxonomy of scheduling problems and solutions [17]. In this work, we explore this disparity between the general concept and a specialized definition of scheduling policies. We denote these practical definitions as being part of a **scheduling context**, a component that specializes a scheduling policy to a scheduling problem through runtime library or user-defined components.

Parallel runtime libraries like StarPU [3] and Charm++ [2] support scheduling solutions as software modules. They expose data structures and endpoints to decouple the scheduling service from other runtime features. This design minimizes the implementation effort of scheduling strategies at the cost of explicit software dependencies with regards to the runtime interface. We refer to these as *native schedulers* because they fulfill the library scheduling needs, are implemented with native components, and frequently reside in the same codebase as the whole library itself.

The dependency between runtime library and native scheduler is bi-directional. The library relies on specialized schedulers to expose its execution model. For instance, Charm++ applications decompose the application domain into a number of chares (i.e. Charm++ notation for tasks) far greater than the number of processors. This strategy, known as over-decomposition, requires the runtime system to re-distribute chares and avoid load imbalance. This task is especially difficult with an application-agnostic framework that may host irregular workloads or communication patterns. Nonetheless, Charm++ and other similar libraries, achieve scalable high performance due to a combination of introspection and dynamic global scheduling. In other words, Charm++ relies on schedulers to consume data from its introspective analysis tools to secure an efficient adaptive execution model. It is important to notice that these requirements are closely related to our definition of the **scheduling context**, in the sense that they define *how*

(i.e., use introspection data as input) and *where* (i.e., compilation unit linked to the Charm++ scheduler module) to implement the scheduling policy algorithm.

State-of-the-art scheduling solutions for HPC applications evaluate topology affinity [24, 13], decide processor frequencies [21], assess dynamic workloads [23], data locality [27] and others. As system architectures transit to the exascale era, new strategies will emerge to handle load imbalance in different levels of the system. Therefore, it is increasingly important to design schedulers in such a way that enables: (i) source code reuse without performance penalties; (ii) individual testing, and (iii) a structured runtime integration process.

2.1 Related work

Popular parallel programming models, such as StarPU [3] and Charm++ [2], implement global schedulers as runtime services encapsulated within the library. These systems support new policies by enforcing a common interface on the scheduler components. New policies are compiled alongside the runtime library and act as *native schedulers*. Some standards like OpenMP [9] describe a minimal set of standard-compliant global schedulers (e.g., static and dynamic). Recent works [14, 5] identify the problem of the limited set of schedulers in standard OpenMP. These works proposed different ways to extend the standard to allow user-defined loop schedulers.

On the topic of component-based approaches for modularity, Aumage et al. [4] 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. This work showcases the importance of providing interfaces for high performance tools to assist the expression and reuse of user-defined components.

Grossman et al. [12] designed a runtime system that allows parallel libraries to co-exist through resource scheduling. In this work, the authors modeled solver libraries as components and proposed a communication interface so libraries can collaborate instead of competing for resources. Their solution uses lambda functions for communication, resulting in low overheads and an extensible interface for new resource scheduling strategies.

The demand for variety and user-defined scheduling is also a studied topic in the scope of the real-time operating system. Similar 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 system kernels. They applied a common higher-level API to enable the user to manipulate schedulers independently from 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 that involve 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.

3 ARTful scheduler software model

We observe that native global schedulers in different runtime libraries have a common program flow composed of three steps: (1) requirements resolution; (2) task-to-resource mapping computation; and (3) task assignment/migration.

Step 1 is characterized by interactions where the policy acquires input data from software entities in the scheduling context. Some libraries facilitate these interactions providing software endpoints for obtaining workload prediction, communication matrix, or topology input data and communication services for distributed schedulers. In Step 2, the scheduling policy calculates the task-to-resource mapping using the data collected from the scheduling context. Then, the scheduler enacts the workload distribution in Step 3, often by yielding the control back to the runtime library and returning a schedule.

Our model, entitled *ARTful scheduler library software model*, aims to isolate the activities of the scheduling policy (Steps 1 and 2) and the scheduling context (Steps 1 and 3). The isolation characterizes two practical scheduler development phases, algorithm implementation and context specialization. Separated by design, artifacts from these phases can live on different codebases, the runtime library and a portable scheduling policy library. We discuss in Section 3.1 how to separate the activities using a component-based model. Then, we detail the entities in the portable policy library interface in Section 3.2.

3.1 Component model for scheduler requirements

The *ARTful model* relies on a component-based approach to represent scheduling requirements. More specifically, ARTful components are lightweight adapters to functionalities already provided by other entities in the scheduling context. They encapsulate common interactions between the policy (e.g., algorithm) and runtime or user-defined interfaces (e.g., system features). This portability strategy is common among HPC libraries like BLAS [6] and oneAPI [1], where mathematical entities (e.g., matrices and tensors) and manipulation routines are specified through a set of standard function interfaces and structures.

The ARTful model defines the software structure of *scheduling requirement concepts*, common data types and routines employed on scheduling policies across runtime libraries. Software entities that provide access to a scheduler requirement k are described by a tuple of parameters known as the *scheduling concept descriptor* $D_k = (I_k, O_k, P_k, C_k)$. The tuple D_k is associated to a single scheduling concept but can be used to describe many different implementations (i.e., software components [18]). The software interface I_k is a set of functions to access the data or routine that the concept represents. The origin O_k is a flag to indicate whether the functionality is provided by the runtime library or user-defined components. The descriptor parameter set P_k describes the component implementation-specific characteristics relevant to the concept. It can characterize, for instance, how the data was gathered, how it is presented in memory, or other aspects that may assist in characterizing a scheduling solution. The check function C_k defines a process to compare two *concept descriptors*. It compares the P_k parameter values of each instance and returns *true* if both descriptors detail compatible concept implementations (i.e., interchangeable). A software component that implements a concept must be described by the concept descriptor and is called an *ARTful component*.

In this work, we experiment with workload-aware scheduling policies from Charm++ and OpenMP. These policies depend on workload predictions for each of the application tasks and may also require the workloads associated with the processing units (e.g, the measured system noise in a core). The workload prediction w is a common requirement concept across scheduling solutions, often stored in memory as an array of numbers with abstract unit. We define P_w as a pair of values (S, M), describing the subject and model type respectively. The subject S states the meaning of the workload values (it can be the *application tasks* or the *system processing units*). The M parameter describes the workload model type, which can be either *dynamic*, when obtained via introspection, or *static* when the values stem from offline analysis. As the M parameter usually does not affect the policy algorithm behavior, we also allow it to assume the

```
1   class WorkloadDataIface {
2       virtual double* values() const = 0;
3       virtual int size() const = 0;
4   };
5
6   class WorkloadDataDesc {
7       typedef WorkloadDataIface iface;
8
9       enum Subject { tasks, pus };
10      enum Model { is_static, is_dynamic, any };
11
12      const Subject subject;
13      const Model model;
14
15      WorkloadDataDesc(Subject s, Model m) : subject(s), model(m) {}
16
17      bool compare(const WorkloadDataDesc& o) const {
18          return subject == o.subject &&
19                  (model == any && o.model != any ||
20                  model == o.model);
21      }
22  };
```

Algorithm 2: A C++ component descriptor and interface implementation for the workload prediction dependency concept. The *WorkloadDataDesc* class describes implementations in terms of the workload model type and what it represents, tasks or processing units. The *WorkloadDataIface* class describes the component interface, the methods available to scheduling policies to access the workload predictions in a context.

value *any*, a value signaling that this parameter is irrelevant to the policy. The origin O_w is only defined by component implementations and are ignored when designing concept descriptors. We define I_w as a function having no parameters and returning a list of numbers. The comparison function C_w checks the parameters S and M of two component descriptors. It returns *true* when the subject S and model type M are the same on both components, with the exception that one of the components is allowed to have *any* as the model type and match with other M values.

The code in Listing 2 showcases a possible implementation of the workload prediction concept using C++ classes. The implementation allows scheduling policies to describe a workload prediction data requirement through an instance of the class `WorkloadDataDesc`. The expressed requirement can be compared with other concept descriptors, enabling component implementations to be matched against the policy requirement request. Furthermore, components that perform the workload prediction concept implement its interface, represented by the virtual class `WorkloadDataIface`. In other words, the scheduling policy requires only a pointer to an object of this class to access its functionalities, never relying on the actual software entities that provide the functionality. Registering components, passing references, managing policy requests, and others, is carried out by entities in the *ARTful library interface*.

3.2 ARTful interface

The *ARTful interface* is composed of two components, the *scheduler context* and the *scheduling policy* handlers. Both entities embody the blackboard design pattern, employed in systems composed of heterogeneous and independent software modules, and responsible for coordinating component interactions. Handlers are responsible for registering, indexing, and fetching concept and scheduling policies components, respectively. Together, these entities abstract and isolate the scheduling policy and the scheduling context.

The *scheduler context handler* uses concept descriptors to register and index ARTful compo-

nent instances. The handler requires a concept descriptor instance when both registering and fetching components. Furthermore, runtime and user components must be registered into the scheduling context handler so they can be used to serve scheduling policy requests. The latter are fulfilled using the comparison function C_k against the registered components for the concept k. If a match is possible, the handler returns a reference to the ARTful component to the policy. The handler raises an error if the match is impossible, signaling that the context cannot fulfill the scheduling policy requirements. The components with origin O_k in the runtime library are compared first, allowing precedence to the native functionalities of the runtime library. At the same time, this scheme allows user-defined components to extend the runtime library functionalities where no component with that origin would be registered.

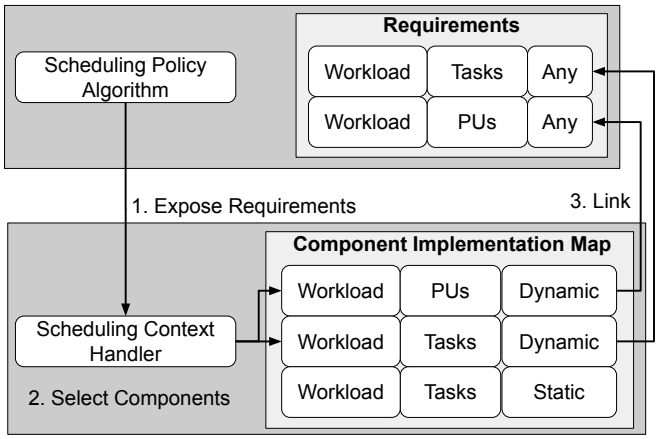


Figure 1: The three-step conversation between the scheduling policy and the scheduling context handler. The relationship between both components is characterized by the policy exposing its requirements in terms of component descriptions. The context handler answers the request selecting the components that best match the description, favoring components registered by the runtime library.

Figure 1 exemplifies the relationship between context, policy, and the *scheduling context handler*. In this example, we continue our example of a workload-aware scheduler framework in the Charm++ system. The library uses introspection to monitor the time spent computing application tasks and background tasks, creating dynamic workload prediction models for tasks and processing units. The extra static workload component is added to showcase that other components, specially user-defined ones, can still be registered in the library interface without conflicts. The exemplified policy requires workload input data for processing units and application tasks. The model type is irrelevant for this policy, so it assumes the value *any*. The policy exposes these requirements in Step 1 by signaling the *scheduler context handler*. The handler evaluates the descriptive parameters of its components during Step 2 using the comparison function. Finally, in Step 3, the handler links the components to the scheduling policy, passing a reference that the policy can use to compute its algorithm.

The *ARTful model* abstracts *scheduling policies* as special components. Unlike the scheduler requirements, the policies do not require a component descriptor. Instead, all policies share the same component interface. This interface takes as input a reference to the *scheduler context handler*, computes the resource map, and returns it as the output. An ARTful scheduler library must therefore define a standard representation for the context handler, the resource map out-

put, and compose these definitions into a unified scheduling policy interface. We explain our approach to defining these points in Section 4, where we discuss our implementation library named *MOGSLib*.

The *scheduling policy handler* allows the runtime library to call the scheduling algorithms. Unlike native runtime schedulers, *ARTful scheduling policies* do not share the same codebase as the runtime. The policy handler registers scheduling policies defined by the user associates them with a name, and provides an interface for the runtime library to access them. In a way, runtime libraries with more than one scheduling solution already contain a policy handler that functions in a similar fashion.

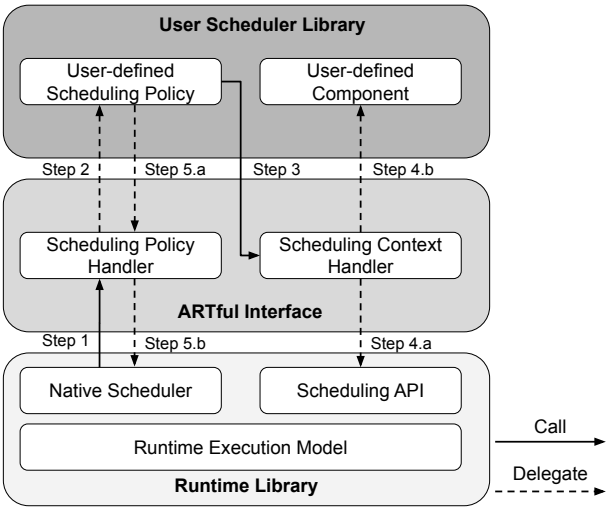


Figure 2: The program flow of an ARTful scheduler library attached to a runtime library to provide portable scheduling solutions

Figure 2 summarizes the program flow of an *ARTful scheduler library*. In Step 1, the native runtime scheduling module calls the *scheduling policy handler* in the ARTful library interface. It passes a policy identifier to the handler to invoke a policy to compute the resource mapping. We assume that the components, policies, and requirement concepts have already been registered to the interface in a previous step. The policy handler finds and calls the scheduling policy in Step 2, passing a reference to the *scheduling context handler*. Then, in Step 3, the policy interacts with the context handler to obtain components that fulfill its requirements. Step 4 is characterized by the fulfillment of the policy requirements, either by components registered by the runtime library (Step 4.a) or by user-defined components that represent the same concept (Step 4.b). In Step 5, the mapping of tasks to resources is computed by the policy component (Step 5.a) and it then returns control to the runtime library (Step 5.b). Although different ARTful libraries share the same behavior, they may not contain interchangeable concepts, components, or policies. In order to support compatible library implementations, the ARTful model must define the same sets of scheduling policy identifiers and requirement concepts. The model isolates development and integration activities, allowing teams to discuss concepts in scheduling solutions as contracts between policy and the features in a runtime library. Ultimately, ARTful is a framework to define standards and implement portable scheduling solutions based on concepts leveraged by research/development teams.

4 MOGSLib: An ARTful scheduler library

We developed the Metaprogramming-oriented Global Scheduling Library (MOGSLib) in a previous work [25]. MOGSLib accomplishes a unified development process for workload-aware scheduling policies that operate on both libGOMP (the popular OpenMP implementation provided by GNU compilers) and Charm++ runtime libraries. MOGSLib is implemented using object-oriented and template metaprogramming directives of the *C++14* language. The library interface is created during compilation through template specialization and compile-time parameters, allowing the *ARTful* abstractions to adjust to the scheduling context at this time. The library defined the concept of workload prediction data, with implementations to the Charm++ library and a user-defined implementation to present libGOMP schedulers with static workload models. In the remainder of this section, we briefly present how *ARTful* components are implemented in MOGSLib and how they are connected to the Charm++ and libGOMP runtime libraries.

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). 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]. These policies, implemented as native scheduling solutions, handle different abstractions for tasks and resources. `BinLPT` is an unofficial libGOMP extension targeting applications with static workload imbalance, and it schedules loop iterations to OpenMP threads. `GreedyLB` is the default dynamic workload scheduler in the Charm++ system and maps chares to operating system processes or threads. Both policies are greedy workload-aware strategies, made different by how they pre-process their workload input data. With respect to integration, `BinLPT` makes part of an unofficial and enhanced version of LibGOMP [23]. `GreedyLB` is developed using Charm++ load balancer framework and makes part of the runtime official codebase.

MOGSLib scheduling policies are developed to schedule the generic concepts of **tasks** and **processing units** (**PUs**). `BinLPT` and `GreedyLB` are implemented as scheduling policy components that depend on the workload prediction requirement concept. MOGSLib codebase encompasses all user-defined components, including scheduling policies and requirement concept implementations. This enables unitary testing and benchmarking, using testing component implementations, therefore allowing tests to be carried out before and after integration to the runtime, and without changes to the scheduling policy. The components are implemented as classes and the scheduling functionality can be accessed through a unified interface.

4.2 MOGSLib scheduling contexts

The development of ARTful libraries, like MOGSLib, must leverage runtime system features of their target libraries. As previously mentioned, runtime features are frequently required in practical scheduling solution development. Specialized component implementations contemplate this scenario, linking the runtime features to the required concepts. The design allows native and portable policies to be equivalent in practice, obtaining the same data, from the same sources, after the same pre-processing steps.

Sometimes libraries have unique characteristics unrelated to requirement concepts. For instance, Charm++ applications may contain unmigratable tasks. The system also allows PUs to become unavailable during runtime. Often, Charm++ scheduling policies add the load of unmigratable tasks to the background noise of the PU workload. In such cases, unavailable PUs and unmigratable tasks are removed from scheduling decisions to avoid producing illegal resource scheduling. These pre-processing steps are implemented by Charm++ load balancer developers, adding these checks when the target application may expose these features.

MOGSLib abstracts access to Charm++'s load balancing database (LBDB) using an internal class. This class contains Charm++ commonly used routines to filter unmigratable PUs and unavailable tasks from the scheduler input. The same class serves as an implementation basis for the components covering the concepts of PU and task workload input data. In this scenario, ARTful improves reusability, allowing any workload-aware policy to benefit from Charm++ data without duplicating LBDB manipulation routines. Code complexity is also reduced, freeing policy developers from considering Charm++ semantics during scheduling algorithm development. MOGSLib workload components for Charm++ expose template parameters that control whether or not these pre-processing routines should be called. Despite this feature being unrelated to the ARTful model, it showcases how the design allows for small quality improvements even on consolidated runtime systems with frameworks for developing their own scheduling solutions.

There will also be cases where a concept is not covered by the library features. For instance, OpenMP lacks support for user-defined schedulers. `BinLPT`'s authors extended libGOMP with their new scheduler by diving into the library's code and adding new control paths to invoke the policy [23]. Adding any new scheduling algorithm would incur more alterations, decreasing the maintainability of the unofficial library. However, supporting new schedulers in this scenario requires even more changes as new data types are included in the decision. They had to extend the library API to add `omp_set_workload`, a function to convey the workload input data from the user level to the runtime. In the *ARTful model*, these unsupported features are covered as user-defined components disassociated from the runtime library. Schedulers and concept implementation extensions can be incorporated into the runtime library, requiring no further changes to it. A single modification to call the *ARTful interface* and to interpret its output is enough to support multiple policies and user-defined features.

4.3 MOGSLib integration with runtime libraries

ARTful libraries containing scheduling policies must be integrated into the runtime system. This can be realized by some form of the adapter design pattern to exchange information. There are two features to cover when integrating an ARTful library into a runtime library. First, specialized components must be registered into the ARTful interface so they are available for the policies. Then, the runtime library must call the library, and enact the schedule based on the policy output. This process varies greatly depending on the characteristics of the runtime library.

In Charm++, user-defined centralized load balancers are developed by extending the C++ `BaseLB` class. The policy logic is called through the virtual `work` function, a universal interface for Charm++ load balancers. MOGSLib is made available in the system through a custom load balancer, `MOGSLibLB`. Algorithm 3 showcases the `work` function of `MOGSLibLB`. It starts by initializing a static reference to the system's load balancing database (line 3). The reference is passed to the abstraction that handles the LBDB manipulation and provides the workload input data concepts. In line 5, the library gets the selected scheduling policy identifier, which can be informed statically through MOGSLib compilation, an environment variable, or a user-defined function. In line 6, the *ARTful scheduling policy handler* is called to pass control to the policy identified by the token `strategy`, obtained in the previous line. Finally, `MOGSLibLB` collects the

library output and uses Charm++'s API (lines 11–13) to enact the schedule decision.

```
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 }
```

Algorithm 3: Charm++ `MOGSLibLB` `work` implementation, directing the schedule decision to the ARTful scheduling policies.

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. Moreover, the alterations to support MOGSLib and the changes to add `BinLPT` are mostly the same [22], save for the endpoints to support workload input data.

LibGOMP uses two functions, `gomp_loop_runtime_start` and `gomp_loop_init`, to decide a schedule. The former function is called when the application reaches an OpenMP `parallel for` construct and it is responsible for deciding which policy to execute. A new option is added to this internal decision mechanism enabling a new policy that we called `mogslib`. The `gomp_loop_init` function happens before the parallel loop starts, and it contains the logic to decide the schedule. At this point, we added MOGSLib calls to obtain the library output in a similar fashion to lines 5-6 in Algorithm 3. Another function added to the LibGOMP implementation, the `gomp_iter_mogslib_next`, was added. This function is called when a thread finishes performing its work pool and it behaves by stealing chunks of iterations from other threads when idle, minimizing the dynamic overhead of static scheduling.

5 Experiments

A common drawback of portable software is the overhead introduced by the portability mechanisms. Overhead is a critical metric for scheduling solutions in HPC as some runtime libraries halt processing units while computing a schedule. In this section, we refer to the time a policy takes to compute a schedule as its *scheduling overhead*. Our experiments are set to compare native and portable policies and evaluate potential overheads induced by the ARTful model. We also analyze the sum of the scheduling overhead across the execution of benchmarks to depict local and global overhead values. Finally, we illustrate the use of scheduling algorithms developed for one runtime system in another runtime system to evaluate software portability.

MOGSLib encapsulates the ARTful scheduling policies while Charm++ and libGOMP systems expose their native solutions. On Charm++, the `GreedyLB` load balancer is compared with `MOGSLibLB`, both employing the same policy. We first compare the schedulers using a synthetic benchmark (*lb_test*) and then a molecular dynamics benchmark (*LeanMD*). On LibGOMP, we use a synthetic benchmark (*SchedCost*) and a molecular dynamics benchmark (*LavaMD*) to compare both versions of the `BinLPT` loop scheduler. Finally, we include experiments using `GreedyLB`

in the context of OpenMP, comparing it to the native `dynamic` scheduler, and using `BinLPT` in the context of Charm++, comparing it to the use of no scheduler and with `GreedyLB`.

All comparisons were 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 the experiments in Section 5.1 and 5.2 is the *Ecotype* cluster from the Grid5000 distributed environment[1]. All Charm++ tests are deployed on four nodes whereas the OpenMP tests run over a single shared memory node. Each node in *Ecotype* contains two Intel Xeon E5-2630L, v4@1.80GHz processors (10 cores per CPU with hyper-threading) and 128GB of DDR3 memory, 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. Meanwhile, the experiments in Section 5.3 were performed on a Dell Latitude 7390 notebook with an Intel Core i5 8250U processor, 16GB of DDR4 RAM (1»16GB, 2400MHz), and an Intel PCIe SSD with 512GB of capacity. It runs the Linux Mint operating system (kernel 5.4.0-59) and its compiler is g++ v7.5.0. We used Charm++ v6.9.0[2], and the custom version of LibGOMP used in this paper is publicly available online with both the original `BinLPT` and MOGSLib schedulers[3].

In the following sections, we discuss the details and results of the experiments with synthetic benchmarks as well as 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 and its implementation of the ARTful model.

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 run over the 80 available processing units (160 virtual cores over 4 compute nodes). The number of tasks 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 3 displays the observed scheduler overhead for the different scenarios as boxplots[4]. The horizontal axis showcases the different application sizes, each portraying the data for both

[1]Information about Grid5000 is available at `https://www.grid5000.fr/mediawiki/index.php/`.

[2]Information about Charm++'s software is available at `http://charm.cs.illinois.edu/software`.

[3]Code available on GitHub at `https://github.com/alexandrelimassantana/libgomp`.

[4]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.

implementations of the `GreedyLB` load balancer, and the vertical axis represents the measured times in milliseconds. As we can see in Figure 3, 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 slight variations in implementation, namely, its smart use of STL algorithms and data structures.

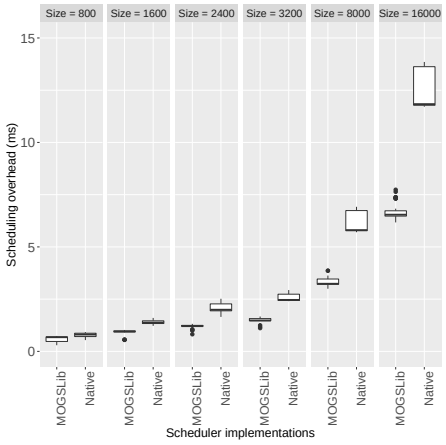


Figure 3: `GreedyLB` scheduler overhead on its native form and in MOGSLib for experiments with *lb_test*.

5.1.2 *SchedCost* on LibGOMP

SchedCost simulates an iterative application where each of its N iterations is composed of one parallel loop as depicted in Algorithm 4. 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).

```
1void 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}
```

Algorithm 4: Computation of the schedule decision cost in OpenMP by *SchedCost*.

We carefully configured *SchedCost* parameters to simulate the same volume of input data of small and medium use cases of the LULESH application [16]. 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. We considered 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 4 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, confirmed by our statistical analysis (both Mann-Whitney U tests returned p-values < 0.05).

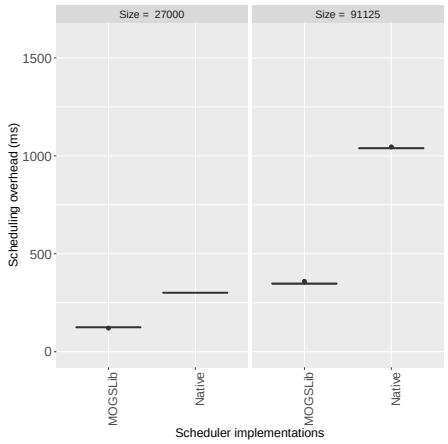


Figure 4: Boxplot of `BinLPT`'s scheduling overhead on its native form and in MOGSLib for experiments with *SchedCost*.

The average native `BinLPT` scheduling overhead is 283 and 1534 ms for the small and medium use cases, respectively, whereas MOGSLib achieves average scheduling overheads of 117 and 514 ms. This represents speedups of 2.42 and 2.98 for the small and medium cases in this section of the code. As such, we conclude that the ARTful model, implemented in MOGSLib, does not introduce notable overhead to workload-aware global schedulers on Charm++ and OpenMP.

5.1.3 Multiloop support

The custom libGOMP library, provided by `BinLPT`'s authors, contains an extra functionality to reduce the scheduling overhead in OpenMP applications. This functionality, called *multiloop support*, allows the user to re-use a previously calculated schedule in a future execution of the same loop. The feature assists iterative simulations that display small performance variations on each simulation step, with scheduling decisions being reused throughout multiple iterations. As expected, this feature required another set of alterations to the OpenMP runtime and new unofficial API extensions.

We implemented the multiloop support in MOGSLib as a library function when attached to OpenMP. Then, we employed the *SchedCost* benchmark once again, 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 in all remaining loop executions.

Figure 5 presents the schedule cost for different multiloop scenarios as boxplots. It differs from Figure 4 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. The origin for this overhead is hard to pinpoint as it is small, around 1 μs on average, when compared to the application execution time. We attempted to recreate all the aspects of the original implementation, except from the software organization where MOGSLib uses C++ classes to store data whereas libGOMP uses plain variables. We envision that this overhead penalty comes from the data access routine in the C++ object, being of static nature and small enough to be considered negligible.

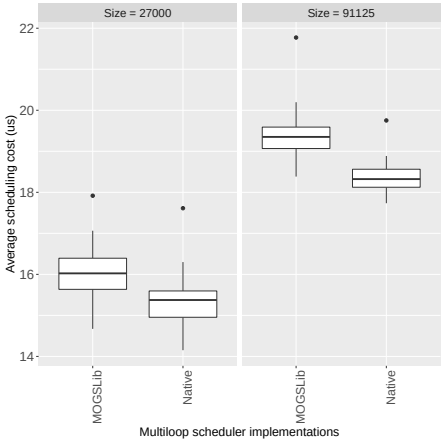


Figure 5: Boxplot of `BinLPT`'s schedule cost on its native form and in MOGSLib for multiloop experiments with *SchedCost*. The vertical axis starts at 14 μs to emphasize the difference between scheduler versions.

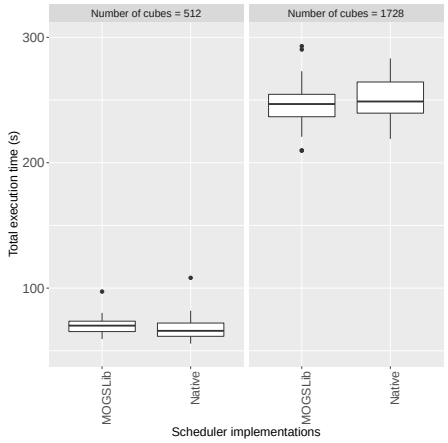
5.2 Molecular dynamics benchmarks

The synthetic benchmark experiments suggest that MOGSLib performs decisions similarly or faster than native schedulers. However, this assertion does not mean that MOGSLib can reduce the application makespan with identical policies. Both versions of a scheduler compute the 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, if any, of the schedulers on the *total execution time* of applications.

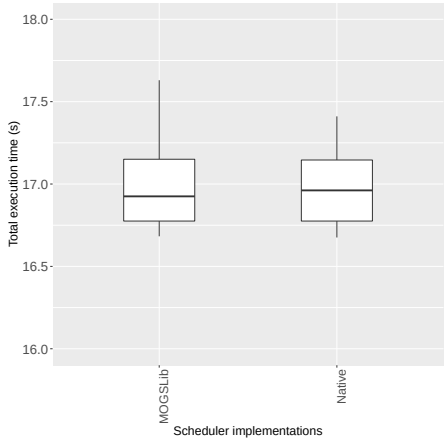
5.2.1 LeanMD on Charm++

LeanMD [19] is a Charm++ application that calculates the force interactions among particles using the Lennard-Jones potential computation[5]. We fixed the benchmark to run for 300 iterations, and to call a load balancer after 20 iterations, repeating the call every 100 thereafter. We simulated three-dimensional spaces with small (8^3 cubes) and medium (12^3 cubes) input sizes. LeanMD was executed 30 times for each configuration of input size and scheduler version.

Figure 6(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.



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



(b) Times for LavaMD using `BinLPT` on LibGOMP.

Figure 6: Total execution time when using native and MOGSLib schedulers. The vertical axes start at different points to emphasize the difference between scheduler versions.

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

[5]LeanMD is available at `http://charmplusplus.org/miniApps/`.

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 when 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 [8] is a benchmark with an OpenMP version that simulates the pressure-induced solidification of molten tantalum and uranium atoms[6]. For these experiments, we recreated a subset of the experiments done by Penna et al. [23] where the `BinLPT` 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 6(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 6(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.

5.3 Experiments with reused schedulers in other runtime systems

Our previous results have focused on the effects of moving from native schedulers to ARTful ones using MOGSLib. We now move our attention to the possibility of using schedulers in a context that is different from where they were originally implemented. These experiments illustrate simple cases using `BinLPT` (originally from OpenMP) in Charm++, and `GreedyLB` (originally from Charm++) in OpenMP.

5.3.1 `BinLPT` in Charm++

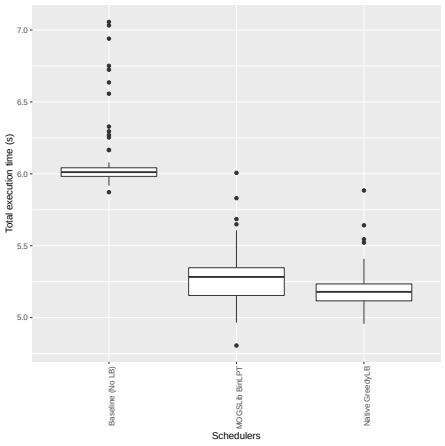
Although `BinLPT` was originally thought for use in OpenMP loops, where data locality can be improved by keeping chunks of sequential iterations with the same thread, we can employ it on Charm++ to keep tasks that communicate on a ring pattern close together. We tested this idea by running *lb_test* (Section 5.1.1) for 20 iterations with calls for a load balancer every 9 iterations. The 64 tasks were set with fixed workloads that vary from $10\mu s$ to $100\mu s$ per iteration, and they use the 8 available virtual cores for execution. In this situation, *lb_test* was executed 100 times using no load balancer, MOGSLib's `BinLPT` (32 chunks), and the native `GreedyLB` algorithm.

Figure 7(a) displays the total execution times measured for the different schedulers as boxplots. We can see that *lb_test* has a better performance using `BinLPT` than the case where no scheduler is employed, demonstrating that `BinLPT` can still be useful in other contexts. We can also see that `GreedyLB` performs slightly better than `BinLPT` and the three schedulers perform differently (all Mann-Whitney U tests returned p-values < 0.05). Nonetheless, `BinLPT` also has the objective of preserving locality.

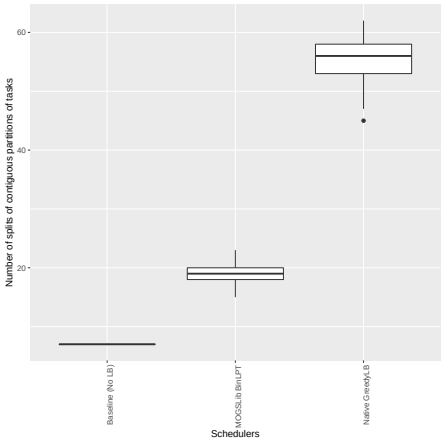
We can check how contiguous the tasks remain after `BinLPT`'s scheduling decisions by counting the number of times tasks i and $i + 1$ are placed on different resources. This is illustrated in Figure 7(b). As we see in this case, the absence of a scheduler leads to a compact mapping of tasks, so splits in their contiguous mapping happen only a number of times equal to the

[6]LavaMD is available at `https://rodinia.cs.virginia.edu/`.

number of resources minus one (7 in our case). On the other extreme, `GreedyLB` cares not for locality, leading to an average of 55 splits. Finally, `BinLPT` can balance the benchmark's load while limiting the number of splits to an average of 19, preserving some locality for the tasks communicating in a ring pattern.



(a) Times for *lb_test* using different schedulers.



(b) Splits for *lb_test* using different schedulers.

Figure 7: Total execution time and number of splits of contiguous task partitions when using different schedulers on Charm++.

5.3.2 `GreedyLB` in OpenMP

As `GreedyLB` implements a list-scheduling algorithm following the well-known LPT rule, it is applicable in several contexts. We were able to evaluate its effect in OpenMP by comparing it to the basic list-scheduling behavior implemented by OpenMP's `dynamic` scheduler. We tested this idea by running *SchedCost* (Section 5.1.2) with exponential costs for its 64 iterations with loads measured in millions of operations. Tests were run 100 times using each of the two aforementioned schedulers.

Figure 8 presents the total execution times of the loops measured for the different schedulers

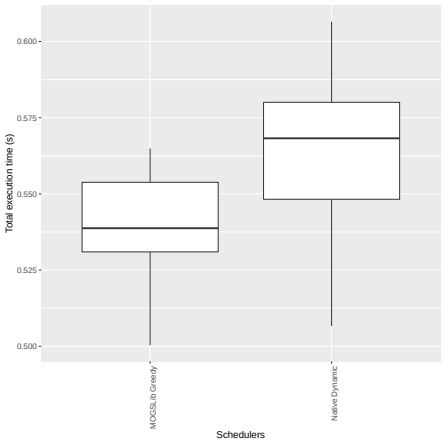


Figure 8: Boxplot of loop execution times (in seconds) using `GreedyLB` and `dynamic` on OpenMP. The vertical axis starts at 500 ms to emphasize the difference between schedulers.

as boxplots. We can see that the native scheduler achieves a median time of 0.568 seconds, while `GreedyLB` outperforms it with a median time of 0.539 seconds (speedup of 1.05). Although small, their difference is statistically significant (the Mann-Whitney U test returned a p-value < 0.05). This result illustrates that new load-aware schedulers can be beneficial to OpenMP even in situations where the overhead of using `dynamic` would be small.

6 Conclusions

In this work, we discussed the current state of scheduling software in the context of HPC applications, which is performed mostly by runtime libraries. Due to the large variety of HPC platforms, portability is given increased value alongside performance (HPC's most critical metric). In this context, we observed similarities across scheduling solutions in runtime libraries and proposed a design to enhance support for user-defined schedulers and implementation portability. The ARTful scheduler model proposed in this work is based on the idea that scheduling solutions contain two kinds of components: generic components, that originate from the development of scheduling policies; and specialized components, that come from the integration with runtime libraries. ARTful abstracts policies and their requirements into software components that interact indirectly through an interface realized by the blackboard design pattern. Our work defines how to model ARTful components and how these interact to support the development of a single user-defined scheduling policy that can be adapted to different runtime systems.

We showcased our implementation of an ARTful library (MOGSLib) to evaluate potential overheads induced by the portable scheduler design. Our experiments revolve around implementing native scheduling policies from the Charm++ and OpenMP systems using the ARTful model. These policies are made available in both runtime libraries indirectly, through the ARTful interface, where we can compare them with their native counterparts. Synthetic benchmark experiments showcased that ARTful schedulers can be faster than their native implementations. However, these performance variations were not relevant enough to improve the total execution times of molecular dynamics applications.

We conclude that the development of portable scheduling solutions is possible through classical software engineering techniques. Nonetheless, our experiments still cover a small fraction

of the design space of scheduling solutions. Our future work aims towards the exploration of this rich space, evaluating new runtime libraries, new scheduling policies, and other execution models (e.g., task-based systems). The component-based design of the ARTful model allows researchers to build their scheduler libraries, allowing custom scheduling features to be inserted into runtime libraries without altering their codebases. We envision that the ARTful model can assist practical scheduler development while many libraries still lack the necessary features for portability or even user-defined policies.

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).

This work was partially developed while Laércio L. Pilla was a member of the *Laboratoire de Recherche en Informatique*.

References

[1] oneapi deep neural network library. `https://github.com/oneapi-src/oneDNN`. Accessed: 2021-01-03.

[2] Bilge Acun, Akhil Langer, Esteban Meneses, Harshitha Menon, Osman Sarood, Ehsan Totoni, and Laxmikant V Kalé. Power, reliability, and performance: One system to rule them all. *IEEE Computer*, 49(10):30–37, 2016.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23:187–198, February 2011.

[4] Olivier Aumage, Julien Bigot, Hélène Coullon, Christian Pérez, and Jérôme Richard. Combining both a component model and a task-based model for hpc applications: a feasibility study on gysela. In *Cluster, Cloud and Grid Computing (CCGRID), 2017 17th IEEE/ACM International Symposium on*, pages 635–644, Madrid, Spain, 2017. Proceedings of the International Symposium on Cluster, Cloud and Grid Computing (CCGrid), IEEE/ACM.

[5] Seonmyeong Bak, Yanfei Guo, Pavan Balaji, and Vivek Sarkar. Optimized execution of parallel loops via user-defined scheduling policies. volume 48, pages 38:1–38:10, Kyoto, Japan, 2019. Proceedings of the International Conference on Parallel Processing, ACM.

[6] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[7] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.*, 14:141–154, February 1988.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. pages 44–54. Proceedings of the International Symposium on Workload Characterization (IISWC), IEEE, 2009.

[9] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[10] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

[11] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

[12] Max Grossman, Vivek Kumar, Nick Vrvilo, Zoran Budimlic, and Vivek Sarkar. A pluggable framework for composable hpc scheduling libraries. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 723–732, Orlando, FL, US, 2017. Proceedings of the International Parallel and Distributed Processing Symposium Workshops, IEEE.

[13] Emmanuel Jeannot, Esteban Meneses, Guillaume Mercier, François Tessier, and Gengbin Zheng. Communication and Topology-aware Load Balancing in Charm++ with TreeMatch. Indianapolis, United States, September 2013. Proceedings of the International Conference on Cluster Computing (CLUSTER), IEEE.

[14] V. Kale, C. Iwainsky, M. Klemm, J. H. Müller Kordörfer, and F. M. Ciorba. Towards A Standard Interface for User-Defined Scheduling in OpenMP. In *Proceedings of the 2019 International Workshop on OpenMP (iWomp 2019)*, Auckland, September 2019. Proceedings of the International Workshop on OpenMP (iWomp).

[15] Vivek Kale and William D Gropp. A user-defined schedule for openmp. volume 11, page 2017, Stonybrook, New York, USA, 2017. Proceedings of the 2017 Conference on OpenMP.

[16] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles Still. Exploring traditional and emerging parallel programming models using a proxy application. Boston, USA, May 2013. Proceedings of the International Parallel & Distributed Processing Symposium (IPDPS), IEEE.

[17] Raquel V Lopes and Daniel Menascé. A taxonomy of job scheduling on distributed computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3412–3428, 2016.

[18] M Douglas McIlroy, J Buxton, Peter Naur, and Brian Randell. Mass-produced software components. pages 88–98. Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany, 1968.

[19] Vikas Mehta. *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.

[20] Malcolm S Mollison and James H Anderson. Bringing theory into practice: A userspace library for multicore real-time scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 283–292, Philadelphia, PA, US, 2013. Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE.

[21] E. L. Padoin, M. Diener, P. O. A. Navaux, and J. Méhaut. Managing power demand and load imbalance to save energy on systems with heterogeneous cpu speeds. pages 72–79. Proceedings of the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), IEEE, Oct 2019.

[22] Pedro Penna, Marcio Castro, Patrícia Plentz, Henrique Freitas, Francois Broquedis, and Jean-François Méhaut. Binlpt: A novel workload-aware loop scheduler for irregular parallel loops. Campinas, Brazil, 2017. Proceedings of the Brazilian Symposium on High Performance Computing Systems, IEEE.

[23] Pedro Henrique Penna, Antônio Tadeu A. Gomes, Márcio Castro, Patricia Della Méa Plentz, Henrique Cota de Freitas, François Broquedis, and Jean-François Méhaut. A comprehensive performance evaluation of the binlpt workload-aware loop scheduler. *Concurrency and Computation: Practice and Experience*, page e5170, feb 2019.

[24] Laércio L Pilla, Christiane P Ribeiro, Pierre Coucheney, François Broquedis, Bruno Gaujal, Philippe OA Navaux, and Jean-François Méhaut. A topology-aware load balancing algorithm for clustered hierarchical multi-core machines. *Future Generation Computer Systems*, 30:191–201, 2014.

[25] Alexandre Santana, Vinicius Freitas, Laércio Lima Pilla, Márcio Castro, and Jean-François Méhaut. Reducing global schedulers complexity through runtime system decoupling. pages 38–44, São Paulo, Brazil, 2018. Proceedings of the Brazilian Symposium on High Performance Computing Systems (WSCAD), IEEE.

[26] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos. A taxonomy of task-based parallel programming technologies for high-performance computing. *Springer Journal of Supercomputing*, 74(4):1422–1434, 2018.

[27] Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L Chamberlain, Romain Cledat, H Carter Edwards, Hal Finkel, et al. Trends in data locality abstractions for hpc systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):3007–3020, 2017.

[28] David W Walker and Jack J Dongarra. Mpi: A standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[29] Gengbin Zheng, Abhinav Bhatelé, Esteban Meneses, and Laxmikant V. Kalé. Periodic hierarchical load balancing for large supercomputers. *The International Journal of High Performance Computing Applications*, 25(4):371–385, 2011.