Reducing Global Schedulers' Complexity Through Runtime System Decoupling

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

Reducing Global Schedulers' Complexity Through Runtime System Decoupling

Alexandre de Limas Santana[*], Vinicius de Freitas[*],
Márcio Castro[*], Laércio Lima Pilla[†], Jean-François Méhaut[†]

Abstract

Global schedulers are components used in parallel solutions, specially in dynamic applications, to optimize resource usage. Nonetheless, their development is a cumbersome process due to necessary adaptations to cope with the programming interfaces and abstractions of runtime systems. This paper proposes a model to dissociate schedulers from runtime systems to lower software complexity. Our model is based on the scheduler breakdown into modular and reusable concepts that better express the scheduler requirements. Through the use of meta-programming and design patterns, we were able to achieve fully reusable workload-aware scheduling strategies with up to 63% fewer lines of code with negligible run time overhead.

1 Introduction

The efforts to provide advances in parallel components, programming models and architectural design by the high performance computing community has led to solutions able to reach unprecedented computational landmarks. Unavoidably, future parallel components will be required to seamlessly benefit from improvements in multiple scientific fronts, preferably with low re-implementation efforts. Of special interest, within the context of dynamic applications, are global schedulers. They are specialized resource management components required to guarantee an adequate allocation of resources in a parallel solution. For that, they must be aware of parallelism intricacies in order to distribute the application workload among available processing elements (PEs).

Scheduling strategies may consider different information, like topology data [Hoefler et al. 2014], power consumption [Langer et al. 2015], or communication affinity [Jeannot et al. 2014, Cruz et al. 2015] to achieve their goals. As applications and scheduling strategies became more complex, runtime systems (RTS) such as *Charm++* [Kale et al. 2007], *OpenMP* [Chapman et al. 2008] and *OpenACC* [Wienke et al. 2012], have been applied as containers and frameworks to simplify the development of applications' parallel behavior and their relationship with global schedulers. As a result, these systems provide reusability to their components and provide a beneficial disconnection of application and scheduler code.

[*]Federal University of Santa Catarina (UFSC), INE, PPGCC, Florianópolis, Brazil

[†]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG. Grenoble, France

A RTS depends on software hooks to assemble components into a parallel solution and common approaches are strict APIs and code annotations. As consequence, RTS components are required to be adapted into the system's workflow and parallelism abstractions (*e.g.,* threads, tasks, chares). However, scheduler components are composed by algorithms that each have their own functional requirements but does not depend on parallelism nor data abstractions. As such, by enforcing such traits through strict software hooks, global schedulers' software become bloated with adaptations, becoming more complex and less resilient to system modifications.

As novel parallel platforms are proposed, larger portions of RTSs are dedicated to exploit their particularities to maximize applications' performance. The exploitation of individual traits in parallel solutions leads to an increase in software complexity and component specialization, ultimately limiting their reusability [Dongarra et al. 2005]. Classic techniques such as aspect-oriented programming [Kiczales et al. 1997] and component-based software engineering [Heineman and Councill 2001] have been used to compose very large systems[1] based on reusable components. However, due to possible resource competition among parallel segments of code, these techniques can not be directly applied [Grossman et al. 2017]. We believe that the lack of studies in how to properly compose global schedulers with other components and RTSs will eventually result in bloated systems that are too complex to manage and challenging to port into future parallel programming models and tools.

To counteract this problem, we propose to exploit the sequential execution flow within RTSs to extract the scheduler component into a self-contained module. Isolated from its context, we are able to create a system-independent global scheduler model based on reusable and specialized concepts. As a result, this model can be used to implement scheduling policies that are less complex due to their isolation from specific technologies, RTSs and external scheduling-unrelated libraries. To achieve this results without high overheads, our proposal is based solely on modern language meta-programming facets and the *Adapter Design Pattern* [Vlissides et al. 1994] to link smaller segments of code into a global scheduler.

We evaluated our proposed model by comparing two re-implemented versions of scheduling policies from *Charm++* and *OpenMP* against the original versions native to these systems. Our global scheduler implementations are independent of RTS which requires them to be packed in external containers. Therefore, a global scheduler library called *Meta-programmed-Oriented Global Scheduler Library* (MOGSLib) was developed to portray a collection of reusable scheduling concepts that can be assembled to form system-specific global schedulers. The main focus of our experimental analysis is the comparison between identical scheduling strategies to evaluate discrepancies in performance, complexity and reusability. The proposed model was able to achieve the original behavior of the strategies in regards to application execution and strategy decision times, and schedule quality, while also lowering the number of lines of code (LoC) needed to express schedulers.

The remainder of this paper is structured as follows: Section 2 presents our global scheduler model. Section 3 describes our experiments. Section 4 discusses our results and analysis. Sec-

[1]Systems featuring millions of source lines of code.

tion 5 presents related work. Finally, Section 6 concludes this paper.

2 System-Independent Scheduler Model

The implementation of a global scheduling algorithm depends on a multitude of factors and design choices aside from the scheduling policy such as: (i) data structure selection, (ii) third-party library usage, (iii) memory placement (*e.g.,* Data- or Object-Oriented Design) and (iv) target RTS. These decisions are important as they provide optimizations for a scheduler in regards to its target environment. However, each design decision further specializes the global scheduler implementation and limit its reusability as a whole.

Regardless of the different designs an implementation can portray, each runtime system and library also offer its own set of capabilities for schedulers (*e.g.,* load balancing database in *Charm++* [Kale et al. 2007]). The divergent interfaces among tools results in different implementations even when accessing a common functionality in distinct systems. As a consequence, modifications in scheduling policies are required when experimenting with different designs choices (RTS, data structures, libraries, etc.).

The contemporary relationship between runtime system and global schedulers is depicted in Figure 1. This figure expresses the dependencies (directed arrows) from software abstractions (inner boxes) to components within their context (outer boxes). As such, the problematic relationships are characterized by dependencies that spans out of the component's source context. Those relationships require unrelated code to be injected into a component, further binding its implementation and increasing its complexity as its code increases.

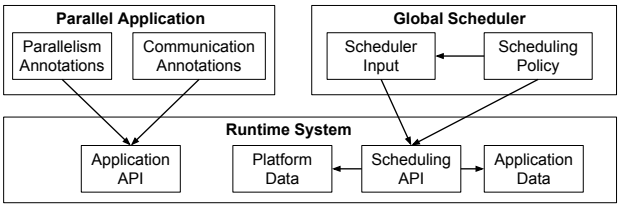


Figure 1: Simplification of parallel components' dependencies.

As a practical example of the aforementioned problem, we propose a scenario where a developer would implement a workload-aware scheduling policy. In this scenario, the scheduler requires the application data regarding its tasks' workload. An implementation of this policy in the *OpenMP* loop-scheduling interface would rely on user provided data as *OpenMP* has neither a method in its scheduling API to query the application workload, nor a method to inform it on the application API. On the other hand, *Charm++* presents data structures on its scheduling API that contain these and other data dynamically collected by the system. Regardless of RTS, the scheduling policy must query its required data from some source. As frameworks for developing global schedulers on these systems must be as flexible as possible, the same scheduling API and data structures are displayed for all policies to obtain their own set of data. This design forces scheduling policies to contain scheduling-unrelated code responsible for manipulating RTS structures to fulfill their functional requirements. Nonetheless, we envision that the exposure of a global scheduler's requirements through scheduling concepts is a solution that not only simplifies the development of schedulers but isolates the policy code from external functionalities.

2.1 Scheduling Concepts

Scheduling concepts are code segments which provide scheduling-unrelated functionalities that may be specialized for different RTSs or contexts. Different specializations of concepts must express its functionalities through functions with equal names/syntax. However, specializations must be sensible to its target environment in order to call the correct procedures needed to fulfill its functionality in the target RTS, platform or library.

The adapter design pattern is a software modeling technique that fits the aforementioned characteristics of scheduling concepts. As an example, a Unified Modeling Language class diagram (UML) is displayed in Figure 2 depicting classes representing both an abstract concept and the specialized concepts for the functionality of querying the application's workload. Both *Static Workload* and *Dynamic Workload* represents specialized concepts for accessing the application workload with different semantics, the former gathers static data and the latter dynamic workload through RTS data structures. Finally, both classes are implementations of the *Application Workload* interface, which defines a layer of functions for accessing the specialization's methods.

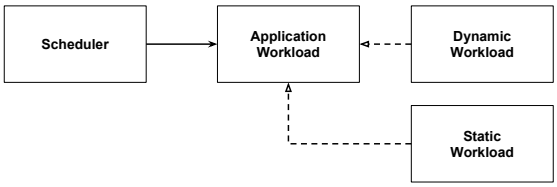


Figure 2: Adapter pattern example.

Although there is the possibility of implementing scheduling concepts solely through the adapter pattern, its usage incurs overheads as it relies on runtime type checks and virtual functions. We propose that scheduling policies can be better declared as partially defined template structures that depend on auxiliary data-types to implement functions that are sensitive to a given context. As template structures, scheduling concepts must be attached to a data-type that contains all the methods the concept requires during compilation. This way, both the scheduling concept and the specialized structure that provides its functionality are loosely linked until compilation, when data-types must be resolved. After the compilation, a direct association links both structures to construct a concrete scheduling concept that can provide its functionality without virtual calls nor dynamic type checks, avoiding their overhead.

To exemplify the proposed approach, we present a snippet in Figure 3, which showcases a concept that exposes the workload of an application. In the snippet, lines 1-6 portray the declaration of a concept that depends on a *Concrete* data-type (line 1) to properly provide its functionality through the *workloads* method (line 5). In lines 8-20, two classes that contain all the required methods to be a *Concrete* type for the *WorkloadConcept* are presented. The first class (lines 10-19) represents a class that packages the semantics to obtain the application's workload from the *Charm++* RTS. The *WorkloadCharm* is capable of querying the load balancing database contained in Charm++ (lines 13-15) and obtain the workload data from its parallelism abstractions, represented by structures named *chares*. On the other hand, the second class (lines 21-30) portrays an auxiliary data structure to the *OpenMP* RTS that registers the application's workload data informed by the user (lines 17-19). With those definitions, a scheduling policy can make use of two complete *WorkloadConcept*, one for the *Charm++* system and other for *OpenMP*

depending of the *Concrete* template parameter. The advantage of this approach is that the concept is entirely responsible for gathering, storing, and manipulating the data structures for exposing its functionality. Ultimately, this design allows for a less complex scheduling strategy that requires no changes if the semantics of acquiring the application workload is changed.

```
1 template<typename Concrete>
2 class WorkloadConcept {
3 public:
4     Concrete data;
5     Load* workloads(){
6       return data.workloads();
7     }
8 };
9
10 class WorkloadCharm {
11 public:
12     LDBD *charm_data;
13     inline Load* workloads(){
14       return charm_data->chares.loads();
15     }
16     inline void set_data(LBDB *data) {
17       charm_data = data;
18     }
19 };
20
21 class WorkloadOmp {
22 public:
23     Load *_loads;
24     inline Load* workloads(){
25       return _loads;
26     }
27     inline void set_workloads(Load *loads){
28       _loads = loads;
29     }
30 };
```

Figure 3: Meta-programmed scheduling concept.

Our approach of using modular and smaller scheduling concepts that compose a larger component displays advantages beyond alleviating the software complexity. Through the addition of dummy classes (like the one in the example) containing testing workloads to schedulers, it is possible to validate the scheduling policy independently from applications or RTSs. That way, we can find flaws in the implementation code on early stages of prototyping more precisely.

```
1 template<typename ... Concepts>
2 class Scheduler {
3 public:
4 TaskMap work(Tuple<Concepts> concepts);
5 };
6
7 template<typename Loads, typename PEs>
8 class Greedy :
9   public Scheduler<Loads, PEs> {
10 public:
11 TaskMap work(Tuple<Loads, PEs> concepts);
12 };
```

Figure 4: Global scheduler model abstraction.

2.2 Global Scheduler Model

Similar to the process of declaring a scheduling concept, a global scheduler can be declared as a template structure that depends on one or multiple scheduling concepts. A *C++* example of this approach is depicted in Figure 4. Lines 1-6 serve as a declaration of the scheduler template model. The first line states that the `Scheduler` template will require an arbitrary number of parameters. The collection of parameters forms the *Concepts* type which is used in line 4 to construct the `work()` function signature. Moreover, as seen in lines 7-12, every scheduler policy has a specialized `work()` function signature that depends on its requirements rather than being defined by an external API.

2.3 The Role of MOGSLib

As concepts are defined as incomplete template structures, there must be concrete classes capable of providing the necessary functionalities for the

concepts. These structures must be sensitive to the parallel solution's contexts (the target RTS, chosen libraries and execution environment) but they should remain decoupled from those. Our proposal is to encapsulate this software stack into a library that exposes a configuration interface that can be easily composed into different contexts. Our implementation of such library is the *Meta-programming-Oriented Global Scheduler Library* (*MOGSLib*), an extensible and open-source library developed in *C++14*[2].

A global scheduler in *MOGSLib* is represented by a tuple (P, F, S) where P is the scheduling policy, F is a set of concrete scheduling concepts and S is a target context. Through this representation, it is possible for a given scheduling policy P_i to generate different global schedulers by utilizing different concrete concepts or being targeted to a different context. As reusability is encouraged, previously developed functionalities can be used to compose new global schedulers, reducing the effort to develop them (coding, testing, etc.) and providing better reproducibility in scientific experiments.

A careful explanation of how the library operates is out of the scope of this work and interested readers are encouraged to check its public repository. Nonetheless, an overview of *MOGSLib* components and their interactions with external technologies is provided in Figure 5 where the components are denoted by labeled boxes and their dependencies by directed arrows. Objectively, *MOGSLib* is attached to the RTS and uses pre-compilation scripts to prompt the user for compilation and template parameters, ultimately generating a global scheduler that can interoperate with external RTSs and libraries (*e.g.,* Load balancers for *Charm++* and loop schedulers for *OpenMP*).

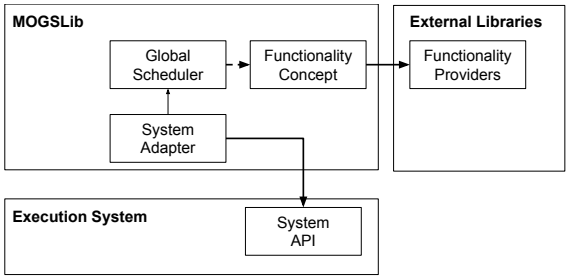


Figure 5: *MOGSLib*'s components overview.

3 Experiments

In order to obtain precise information about overhead, we chose to implement centralized and greedy scheduling policies due to their predictable quasi-linear execution time. This class of schedulers is capable of making fast and precise decisions in smaller scenarios while displaying little variations in policy execution time when receiving the same input data. As such, greedy schedulers are ideal to observe small overhead variations between different global scheduler models.

In this work, we have chosen two greedy policies to implement in our model to compare against the native versions found in runtime systems. In this work, we selected two policies implemented within runtime systems to re-implement in our model Those policies are: (i) *Charm++*'s native greedy scheduler (*GreedyLB*), and (ii) a workload-aware loop scheduler implemented in *libGOMP* (*BinLPT*) [Penna et al. 2016]. The *GreedyLB* strategy iteratively pulls tasks from a task load *max-heap* and assigns to the top element of a PE load *min-heap* until there are no more unassigned tasks. The loop scheduler *BinLPT* groups adja-

[2]Available at: `https://github.com/ECLScheduling/lb-framework`

cent iterations of a loop in up to k task packs (defined by the user) and iteratively assign the heaviest group to the least overloaded PE. Although different, these strategies share the same scheduling concepts requirements, which also serve as an example of code reuse. The required scheduling concepts are: (i) application workload data retrieval and (ii) PE workload data retrieval.

3.1 Evaluated Metrics

Given the intent of analyzing a development model rather than a novel scheduling policy, our metrics have the intent of spotting differences between implementations and are enumerated as follows: (i) strategy decision time, (ii) application makespan, (iii) global scheduler lines of code (LoC) and (iv) number of reusable LoC. The time related metrics have the objective of measuring the overhead incurred by our model both in application makespan and in strategy decision time. The LoC metric serves as an indicator of the code complexity as fewer lines point to less complex segments of code [Nguyen et al. 2007].

3.2 Software and Hardware

In order to compare our model against native implementations, our evaluation contemplates the *Charm++* v6.7 and *OpenMP* v4.0 runtime systems. The *MOGSLib* library, *Charm++* runtime and benchmarks were compiled with `g++` v5.4.0 with the following compilation flags: `-O3 -std=c++14`. Finally, the *libGOMP* library was compiled with its own *makefile* found in its aforementioned repository with the `gcc` compiler without additional flags.

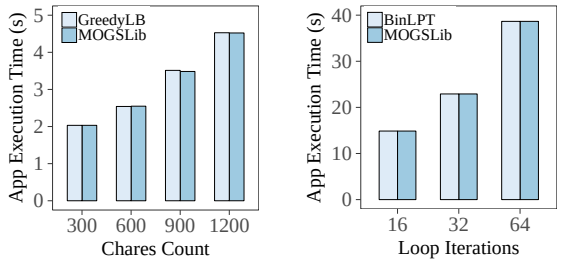
To test the greedy strategy in *Charm++*, we chose the synthetic benchmark contained within the default *Charm++* package, *LB Test*, an iterative application that issues busy wait operations to simulate the workload. The benchmark was executed with different configurations in order to discover a parameter set that displays enough load imbalance to benefit from a global scheduler. To create this scenario, the following *LB Test* configurations were applied: (i) **Iterations**: 150, (ii) **Load balancing calls**: every 40 iterations, (iii) **Minimal task load**: 10 microseconds, (iv) **Maximum task load**: 3000 microseconds. To analyze the schedulers' scalability under different numbers of tasks, we ran this experiment with 300, 600, 900 and 1,200 tasks.

The tests using the *OpenMP* runtime system were executed over a modified version of *libGOMP*, the library responsible for providing the *OpenMP* directives implementations for open-source compilers. The modified version of *libGOMP* contains the required hooks for both MOGSLib and *BinLPT* and can be found in GitHub[3].

We test the *BinLPT* scheduler with the *SimSched*[4] synthetic benchmark. This application simulates CPU intensive kernels utilizing statistical distributions to generate random classes of workload that are later assigned to loop iterations. The parameters for the *SimSched* benchmark used in this paper were selected in conformity to the *BinLPT* paper. Their objective is to create a use case that better fits the use case of this global scheduler and are configured as follows: (i) **Distribution**: exponential, (ii) **Number of workload classes**: 12, (iii) **Kernel complexity**: quadratic. The necessary modifications to support the *BinLPT* in *OpenMP* system, *SimSched* benchmark details and the parame-

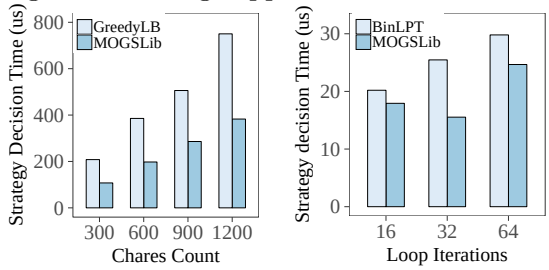
[3]Available at: `https://github.com/ECLScheduling/MOGSLib-libgomp-benchmark`

[4]Available at: `https://github.com/lapesd/libgomp-benchmarks`



(a) *Charm++*. (b) *OpenMP*.

Figure 6: Average application execution time.



(a) *Charm++*. (b) *OpenMP*.

Figure 7: Average schedule decision time.

ters to test workload-aware in it are explained in [Penna et al. 2016].

Our experiments were executed on the *Genepi*[5] cluster within the *Grid'5000* distributed environment. Furthermore, our *Charm++* tests were executed over 4 nodes whereas the *OpenMP* tests used only one.

4 Results

The results of our experiments in regards to total application execution time are provided in Figure 6(a) for the *Charm++* system and Figure 6(b) for *OpenMP*. Each bar in those figures represents the arithmetic mean of 50 application runs. In order to further analyze the application execution time, we executed two-tailed *t-student* tests to check if both scheduler versions (native and *MOGSLib*) were originated from a distribution with the same parameters. The confidence interval was set to 5% and p-values are displayed in Table 1.

Table 1: Application execution time parametric tests results.

Charm++ environment	
Task Count	p-value
300	0.96
600	0.55
900	0.18
1200	0.43

OpenMP environment	
Loop Iterations	p-value
16	0.45
32	0.30
64	0.38

As the experiments generated p-values that surpass the confidence interval of 0.05, we cannot reject the null hypothesis that both distributions are equal. This conclusion implies that both native and the *MOGSLib* schedulers are able to perform equally on the different tested applications, runtime systems and application size.

4.1 Schedule Decision Time Analysis

In order to analyze both implementations in detail, we analyzed the time taken in order to decide the task mapping on each of the aforementioned scenarios through Figures 7(a) and 7(b). The bars depicted in these figures represent the arithmetic mean of the time each strategy took to decide a schedule. Therefore, in *Charm++* experiments, each bar represents 150 data points (3 schedules computed for each of the 50 runs). Moreover, in *OpenMP* experiments, each bar is composed by 50 data points, as the scheduler is called before the loop and there is only one loop per application kernel.

[5] *Genepi* complete system specification at: `https://www.grid5000.fr/mediawiki/index.php/Grenoble:Hardware#genepi`

The scheduler decision time data depicted in Figures 7(a) and 7(b) presented standard deviations smaller than 1%. Furthermore, our model portrayed decision times that were 45% and 18% faster on *Charm++* and *OpenMP* systems, respectively. However, for these tests, the impact of the scheduler decision time is negligible due to its scale (microseconds) in comparison to the application makespan (in seconds). This overhead would be more important in scenarios with more tasks or higher rescheduling frequency. With a small time scale, differences between implementations were bound to happen and their origin is related to different parameters between schedulers. In *Charm++*, generic data structures were used in our model in contrast to the ones used by the *Charm++* scheduler. Moreover, the only difference between the schedulers in *OpenMP* was the compiler used to generate the *MOGSLib* global scheduler. While *libGOMP* is compiled through `gcc`, *MOGSLib* used `g++` to compile and link its scheduler into *OpenMP*.

4.2 Complexity Analysis

To better analyze our model in contrast to native implementations, we break our approach in different components that form the global scheduler. Each component's lines of code count is displayed in Table 2, with the last column designated to portray where the component can be reused within other parallel solutions.

Table 2: MOGSLib components' LoC.

Component	*BinLPT*	*GreedyLB*	**Reusable on**
Scheduling Policy	37	30	Runtime Systems
Runtime System Adapter	60	22	Scheduling Policies
Concepts	30	40	Scheduling Policies

The native versions of *BinLPT* and *GreedyLB* are composed, respectively, by 84 and 81 LoC. Our version of those same schedulers are composed by 127 and 97 LoC respectively when accounting for the sum of components that assemble the scheduler. However, every component can be reused in at least one scenario as stated in the last column in Table 2. In the scenario where a new scheduler is proposed and the concepts and RTS adapter have been previously developed, the only required implementation is the scheduling policy. This scenario is not uncommon as the concepts can be reused and novel policies are encouraged to use existing system adapters and scheduling concepts. As such, when analyzing solely the size of the scheduling policy code, our model attained up to 63% size reduction in comparison to the native versions.

The segmentation into concepts is advantageous as each concept portrays a single role within the scheduler in contrast to current implementations found in RTSs. Despite resulting in a larger sum of LoC, this approach enables the composition of functionalities implemented by developers of different expertise. That way, developers can rely on reusing concrete concepts rather than reimplementation, leaving the burden of assembling the functionalities to be taken care by libraries such as *MOGSLib*.

5 Related Work

Schedulers are a relevant topic in real-time systems due to application diversity which, ultimately, demands specialized policies. Classically, schedulers are kernel components and, as such, developed policies are tied to a specific patch and OS. Furthermore, to enable higher customization and enhance the range of scheduling policies, Asberg [Åsberg et al. 2012] and Mollison [Mollison and Anderson 2013] moved the

policies implementations from kernel space to the user space. Through the use of an abstraction layer inside the kernel space, their work showcased policies developed over higher level abstractions with acceptable overhead in hard real-time systems. Ultimately, both proposals used different techniques to decouple the schedulers from the kernel primitives. However, in concordance with our proposal, they also proposed the extraction of the scheduler component into its own module.

In respect to providing modularity to scheduler components, HPC runtime systems like *Charm++*, *OpenMP* and *OpenACC* provide a simple way for decoupling application and scheduler code. Either through annotations, abstractions or language modifications, these systems allow resource management hooks in the applications life-cycle, thus enabling reuse and portability of scheduling policies among applications within the same system. Nonetheless, scheduler implementations are yet limited to a specific abstraction set, contain system-specific code and their algorithms are often scattered through different segments in the RTS. Ultimately, our approach intends to benefit from these RTSs while alleviating the scheduling implementation problems associated with their usage.

Grossman et al. [Grossman et al. 2017] proposed that, through a better description of a component's connections, the composability of parallel libraries can be achieved through modern language facets such as lambda functions and asynchronous calls. Their work is oriented towards the development of a novel runtime system that showcases component connections using lambda functions. Nonetheless, our work shares the use of modern language traits and better description of components' connections. However, we apply meta-programming and ultimately target system-independence rather than proposing a novel system architecture.

Bigot [Bigot et al. 2012] defined parallel solutions as an assembly of components that can be interconnected and swapped to provide performance portability. The work is based on performance portability through modularity and applies driver components to resolve the intricacies of specific systems and technologies. Despite the similarities, our work diverges in the technique applied and context within the parallel solution. Ultimately, their approach presents a component-based model for applications that utilizes a small runtime to link components together, whereas our work presents an attachable scheduler model that can be linked to runtime systems through a library in compilation time.

6 Conclusions

In order to ease the development of global schedulers and enable simpler scheduling policy implementations, this paper contributes to the topic by exposing a global scheduler model capable of describing its requirements through meta-programmed scheduler concepts. We discuss the impacts of reusability, modularity and software complexity on parallel components while we present a novel library, *MOGSLib*, as a technical contribution for global scheduler developers. The evaluation of the proposed model is made through synthetic benchmarks executed on *Charm++* and *OpenMP* systems analyzing two distinct workload-aware scheduling policies, *GreedyLB* and *BinLPT*.

Our results (presented in Section 4) displayed that the model incurs in negligible variations in scheduler quality and application makespan. Additionally, at the best case scenario, our approach can reduce the number of LoC needed to develop

a new global scheduler by up to 63% when reusing previously implemented scheduling concepts. The possibility of component reusability is beneficial as it enables code replayability without development efforts and less complex software segments. Even in the worst case scenario, where concepts must be implemented from scratch, this approach allows for a better prototyping phase that can adhere to test-oriented development due to each module being responsible for a single role in the system.

In regards to future works, we intend to further study the proposed model, experimenting its adoption into different scheduling policies. Of special interest are strategies that take into account informations about the platform topology, task affinity and memory hierarchy. As more functionalities are required for different policies, we aim to enhance *MOGSLib* with more concrete scheduling concepts and system adapters to provide developers with more tools and options to compose simple and reusable global schedulers.

ACKNOWLEDGMENT

This work was partially supported by the Brazilian Federal Agency for the Support and Evaluation of Graduate Education (CAPES) and by the Brazilian Council of Technological and Scientific Development (CNPq), project grant 401266/2016-8.

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`).

References

[Åsberg et al. 2012] Åsberg, M., Nolte, T., Kato, S., and Rajkumar, R. (2012). Exsched: An external CPU scheduler framework for real-time systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on*, pages 240–249. IEEE.

[Bigot et al. 2012] Bigot, J., Hou, Z., Pérez, C., and Pichon, V. (2012). A low level component model enabling performance portability of HPC applications. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 701–710. IEEE.

[Chapman et al. 2008] Chapman, B., Jost, G., and Van Der Pas, R. (2008). *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press.

[Cruz et al. 2015] Cruz, E. H., Diener, M., Pilla, L. L., and Navaux, P. O. (2015). An efficient algorithm for communication-based task mapping. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 207–214.

[Dongarra et al. 2005] Dongarra, J., Sterling, T., Simon, H., and Strohmaier, E. (2005). High-Performance Computing: Clusters, Constellations, MPPs, and Future Directions. *Computing in Science and Engineering*, 7:51–59.

[Grossman et al. 2017] Grossman, M., Kumar, V., Vrvilo, N., Budimlic, Z., and Sarkar, V. (2017). A pluggable framework for composable HPC scheduling libraries. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 723–732. IEEE.

[Heineman and Councill 2001] Heineman, G. T. and Councill, W. T. (2001). Component-based software engineering. *Putting the pieces together, addison-westley*, page 5.

[Hoefler et al. 2014] Hoefler, T., Jeannot, E., and Mercier, G. (2014). *An Overview of Topology Mapping Algorithms and Techniques in High-Performance Computing*, pages 73–94. John Wiley & Sons, Inc.

[Jeannot et al. 2014] Jeannot, E., Mercier, G., and Tessier, F. (2014). Process placement in multicore clusters: Algorithmic issues and practical techniques. *IEEE Transactions on Parallel and Distributed Systems*, 25(4):993–1002.

[Kale et al. 2007] Kale, L. V., Bohm, E., Mendes, C. L., Wilmarth, T., and Zheng, G. (2007). Programming petascale applications with Charm++ and AMPI. *Petascale Computing: Algorithms and Applications*, 1:421–441.

[Kiczales et al. 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer.

[Langer et al. 2015] Langer, A., Totoni, E., Palekar, U. S., and Kalé, L. V. (2015). Energy-efficient computing for HPC workloads on heterogeneous manycore chips. In *Proceedings of Programming Models and Applications on Multicores and Manycores*. ACM.

[Mollison and Anderson 2013] Mollison, M. S. and Anderson, J. H. (2013). 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. IEEE.

[Nguyen et al. 2007] Nguyen, V., Deeds-Rubin, S., Tan, T., and Boehm, B. (2007). A SLOC counting standard. In *Cocomo ii forum*, volume 2007, pages 1–16. Citeseer.

[Penna et al. 2016] Penna, P. H., Castro, M., Freitas, H. C., Broquedis, F., and Méhaut, J.-F. (2016). Design methodology for workload-aware loop scheduling strategies based on genetic algorithm and simulation. *Concurrency and Computation: Practice and Experience*.

[Vlissides et al. 1994] Vlissides, J., Helm, R., Johnson, R., and Gamma, E. (1994). Design patterns: elements of reusable object-oriented software.

[Wienke et al. 2012] Wienke, S., Springer, P., Terboven, C., and an Mey, D. (2012). OpenACC—first experiences with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer.