UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Alexandre de Limas Santana

# A design method for supporting the development and integration of *ARTful* global schedulers into multiple programming models

Florianópolis
2019

Alexandre de Limas Santana

# A design method for supporting the development and integration of *ARTful* global schedulers into multiple programming models

Dissertação submetido ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do título de Mestre em Ciência da Computação.
Orientador: Prof. Dr. Márcio Bastos Castro
Coorientador: Prof. Dr. Laércio Lima Pilla

Florianópolis
2019

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Santana, Alexandre
A design method for supporting the development and integration of ARTful global schedulers into multiple programming models / Alexandre Santana ; orientador, Márcio Castro, coorientador, Laércio Pilla, 2019.
76 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2019.

Inclui referências.

1. Ciência da Computação. 2. Escalonamento global. 3. Modelos de programação paralela. 4. Biblioteca. 5. Portabilidade. I. Castro, Márcio. II. Pilla, Laércio. III. Universidade Federal de Santa Catarina. Programa de Pós Graduação em Ciência da Computação. IV. Título.

Alexandre de Limas Santana

**A design method for supporting the development and integration of *ARTful* global schedulers into multiple programming models**

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Douglas D. J. de Macedo, Dr.
Universidade Federal de Santa Catarina

Prof. Gerson Cavalheiro, Dr.
Universidade Federal de Pelotas

Prof. Luis Fabricio Wanderley Goés, Dr.
Pontífica Universidade Católica de Minas Gerais

Certificamos que esta é a **versão original e final** do trabalho de conclução que foi julgado adequado para obtenção do título de Mestre em Ciência da Computação.

Digitally signed by Jose Luis Almada Guntzel:53378067004
Date: 2019.09.13 17:43:25 -03'00'

Prof. José Luís A. Güntzel, Dr.
Coordenador do Programa

Digitally signed by Marcio Bastos Castro:00370772016
Date: 2019.09.13 17:36:00 -03'00'

Prof. Dr. Márcio Bastos Castro
Orientador

Florianópolis, 7 de agosto de 2019.

Eu dedico este trabalho à minha família, meus amigos, e especialmente à minha esposa Danielly. Vocês foram, e são, meu suporte nos momentos mais escuros, assim como também companheiros dos mais radiantes acontecimentos desta jornada. Jamais esquecerei meus orientadores, agradeço sua paciência e entusiasmo, as verdadeiras características daqueles que carregam, na linha da frente, a tocha da curiosidade nas cavernas da dúvida.

**ACKNOWLEDGEMENTS**

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

## RESUMO

Plataformas de execução em ambientes de alto desempenho estão tornando-se cada vez mais diversas com o desenvolvimento de novas arquiteturas e ferramentas para se beneficiar do paralelismo inerente das aplicações. Estas novas opções de ferramentas oferecem possibilidades de melhoria no desempenho de aplicações científicas e de engenharia. A crescente gama de plataformas torna mais complexa a distribuição das tarefas da aplicação no ambiente, passo que deve ser gerido pelos sistemas de execução e seus balanceadores de carga, de modo a não prejudicar a portabilidade da aplicação. No entanto, o desenvolvimento e implantação de novos escalonadores de tarefas em sistemas amplamente utilizados na indústria como *OpenMP* e *MPI* sofrem com a falta de suporte de *frameworks*. Este trabalho propõe uma biblioteca, *MOGSLib*, para auxiliar o desenvolvimento e implantação de escalonadores globais em diferentes sistemas de execução de alto desempenho. *MOGSLib* aplica abstrações reutilizáveis, independentes e testáveis na forma de classes template em C++ para representar as políticas de escalonamento, sua relação com o sistema de execução e a taxonomia da solução de escalonamento. Nós avaliamos o sobrecusto de nossas estratégias portáveis em comparação com suas versões nativas nos sistemas de execução em dois ambientes distintos, os sistemas *Charm++* e *OpenMP*. Em nosso experimentos, conseguimos dar suporte à definição de estratégias de escalonamento do usuário e sua implantação como escalonadores de laço na GNU library for OpenMP (libGOMP) e balanceadores de carga no *Charm++* através da seleção de implementação das abstrações que os compõem. Nós verificamos que nossas versões dos escalonadores oferecem tempos de execução de aplicação equivalentes aos balanceadores nativos para a classe de escalonadores centralizados e cientes de carga aplicados em *kernels* de dinâmica molecular. Por fim, a flexibilidade de incorporar funcionalidades e políticas de escalonamento do usuário em sistemas de execução com modificações limitadas nos códigos de sistemas de execução mostra que é possível construir balanceadores de carga flexíveis com pouco sobrecusto até para ambientes de alto desempenho.



**Palavras-chave:** Escalonamento global. Sistemas de execução. Portabilidade de implementação. Qualidade de software. Reusabilidade.

## RESUMO ESTENDIDO

### Introdução

Aplicações científicas e de engenharia destinadas a ambientes de alta performance estão constantemente evoluindo. Os avanços em múltiplas fontes de pesquisa nesta área possibilitaram que o comportamento destas aplicações seja expressado de maneira produtiva por via de modelos de programação paralela. Dentre outras características, estes modelos auxiliam desenvolvedores a atingir aplicações com portabilidade de performance em diferentes ambientes de execução através da configuração de parâmetros direcionados para cada cenário de execução (ASTORGA et al., 2018; PENNYCOOK; JARVIS, 2012; EDWARDS; TROTT; SUNDERLAND, 2014; HORNUNG; KEASLER, 2014).

Uma visão genérica da plataforma de execução e das estratégias usadas para distribuir as tarefas da aplicação são maneiras eficientes de alcançar aplicações com portabilidade de performance. De fato, a distribuição dinâmica da carga de uma aplicação é uma característica com suporte em muitos dos modelos de programação paralela. Este serviço é prestado pelos escalonadores globais do sistema de execução, componentes que atacam o problema do desbalanceamento de carga por via da redistribuição das cargas da aplicação durante a execução da mesma. No entanto, a lógica executada por estes escalonadores é diretamente relacionada com o modelo de execução empregado pelo sistema de execução. Por exemplo, no *OpenMP*, onde um modelo de *fork-join* é empregado, os escalonadores de laço são responsáveis por decidir quais *threads* executarão cada iteração do laço (DAGUM; MENON, 1998). No sistema de *work-pool* distribuído do *Charm++*, balanceadores de carga devem decidir em qual Processing Unit (PU) os *chares* (abstração do sistema para a unidade mínima de paralelismo) serão alocados para serem computados (ACUN; KALE, 2016). Outros sistemas como o *StarPU* aplicam sistemas de execução baseado em tarefas em sistemas heterogêneos onde os escalonadores devem alocar as tarefas em unidades de processamento com base em grafos de dependência (AUGONNET et al., 2011).

Sistemas de execução possuem estratégias de escalonamento capazes de cobrir as demandas da maioria das aplicações regulares usando escalonadores estáticos. Adicionalmente, o suporte para aplicações irregulares é geralmente feito por via de estratégias de roubo de tarefas ou filas centralizadas (LOPES; MENASCÉ, 2016). Quando estas técnicas genéricas não são suficientes, novas demandas por politicas de escalonamento impulsionam o desenvolvimento de novas técnicas a serem integradas em sistemas de execução (CIORBA; IWAINSKY; BUDER, 2018). Entretanto, apenas um pequeno grupo de sistemas possui um *framework* integrado para a construção de escalonadores definidos pelo usuário (ACUN et al., 2016; AUGONNET et al., 2011) e, mesmo assim, as implementações são limitadas a um único sistema de execução. Consequentemente, escalonadores personalizados são integrados por métodos intrusivos, modificações dos sistemas de execução (KALE; GROPP, 2017) ou mesmo diretamente ligados à aplicação (MEI et al., 2011). Todavia, estes processos geram efeitos colaterais indesejados desde a criação de versões não oficiais destas bibliotecas até a degradação da qualidade de *software* das soluções.

### Objetivos

Este trabalho tem como objetivo promover ferramentas para dar suporte ao desenvolvimento de escalonadores globais que possam aderir a múltiplos sistemas de execução. Para isto, é necessário analisar o comportamento dos escalonadores globais em diferentes sistemas de execução afim de sintetizá-lo em abstrações genéricas. Nossa abordagem envolve

a criação de um conjunto de abstrações e especificações, intitulado de *ARTful scheduling*, para representar escalonadores globais de maneira genérica, porém com a capacidade de se adequar a diferentes contextos de uso em sistemas de execução. Tendo em vista que nosso foco são ambientes de alta performance, devemos, adicionalmente, garantir que a proposta não entra em conflito com a principal métrica do escopo, o sobrecusto de performance na tomada de decisão do mapeamento de tarefas. Para avaliar nossa abordagem, as especificações serão implementadas em uma biblioteca e *framework* de desenvolvimento, intitulada de Meta-Programming Oriented Global Scheduler Library (MOGSLib), e incorporado em dois sistemas de execução utilizados em ambientes de alto desempenho: libGOMP e *Charm++*. Sendo assim, nossas contribuições e objetivos são delineados da seguinte forma:

- Definição das abstrações e especificações *ARTful scheduling* para escalonadores globais portáveis;
- Implementação das abstrações na forma de uma biblioteca portável de escalonadores globais intitulada de MOGSLib;
- Implantação da biblioteca MOGSLib nos sistemas de execução libGOMP e *Charm++*;
- Análise comparativa dos escalonadores globais da MOGSLib em relação a estratégias idênticas porém implementadas de maneira nativa nos sistemas de execução.

**Método**

Escalonamento e portabilidade de implementação são temas presentes em diversas áreas da computação. Mesmo dentro da área de alta performance, estes temas são recorrentes e podem aparecer com diferentes conotações e usos. Sendo assim, como parte de nossa metodologia, nós nos propusemos a estudar as diferentes formas ao qual escalonadores são empregados em aplicações para alto desempenho assim como as técnicas existentes para portabilidade de componentes dentro desta área. A partir de uma análise sistemática da literatura, os seguintes passos serão seguidos para atingir os objetivos:

1. Levantamento do estado da arte nos temas de portabilidade de implementação, sistemas de execução e bibliotecas de escalonadores globais;
2. Criação de um modelo conceitual para a expressão de escalonadores globais que se adéque ao *ARTful scheduling*;
3. Seleção de escalonadores globais implementados diretamente nos sistemas de execução *Charm++* e libGOMP;
4. Re-implementação dos escalonadores do passo 3 através das abstrações encontradas no passo 2 na nova biblioteca proposta MOGSLib;
5. Desenvolvimento de ferramentas na biblioteca MOGSLib para auxiliar o processo de teste, composição, especialização e implantação dos escalonadores genéricos nos sistemas *Charm++* e libGOMP;
6. Análise comparativa do sobrecusto de decisão de mapeamento de ambas versões dos escalonadores em *benchmarks* sintéticos e *kernels* de aplicações em ambos os sistemas.

**Resultados e Discussão**

Este trabalho atingiu uma definição genérica de escalonadores globais por meio da decomposição do domínio de escalonadores em abstrações menores e agrupáveis. Esta abordagem mostrou-se condizente com os trabalhos relacionados encontrados e capaz de prover

descrições genéricas porém passíveis de especialização em diferentes sistemas. As especificações do *ARTful scheduling* pode ser implementadas e aplicadas de diferentes formas. No decorrer deste trabalho, a abordagem selecionada foi usar o suporte da linguagem *C++* à programação genérica para construção da biblioteca MOGSLib. A implementação das abstrações do *ARTful scheduling* podem ser integradas aos sistemas de execução *Charm++* e libGOMP. A adequação dos escalonadores ao sistema de execução dá-se pela abstração da relação entre escalonador e sistema, permitindo a expressão de suas interações através do uso de uma camada de indireção usada para esconder componentes especializados atrás de uma interface comum e genérica.
Os escalonadores globais escolhidos para os experimentos são da classe de escalonadores cientes da carga de trabalho da aplicação. Esta escolha está relacionada com o uso desta classe de escalonadores em diversos modelos de programação paralela e das diferentes formas que estes tomam em sistemas distintos. No sistema libGOMP, o escalonador escolhido foi o *BinLPT* (PENNA et al., 2017) enquanto no *Charm++* foi selecionado o balanceador de carga padrão do sistema intitulado de *GreedyLB*. Em ambos os sistemas, as versões implementadas a partir das especificações de *ARTful scheduling* obtiveram performance melhor na tomada de decisão do mapeamento, chegando a um ganho de até 48% no sistema de execução *Charm++* ao executar *benchmarks* sintéticos. No entanto, estes ganhos tem pouca significativa em relação aos tempos de execução das aplicações, que apresentam-se em maiores magnitudes. Em relação a esta métrica, ambas as versões dos escalonadores obtiveram impactos estatisticamente similares, o que indica a equivalência entre as duas implementações em quesito ao seu impacto.

**Considerações Finais**

Quantificar a complexidade e o esforço do desenvolvimento de componentes de gerência de recursos ainda é um desafio na área de computação de alto desempenho. Alguns fatores como o auxílio de bibliotecas para desenvolvimento e a adequação do problema a modelos de programação são tidos como fatores de alto impacto para redução do esforço do desenvolvimento (WIENKE et al., 2016). Nossa implementação de escalonadores globais independentes de plataforma atingiu o objetivo de prover aos desenvolvedores um conjunto de abstrações e um processo de desenvolvimento para a criação de escalonadores globais em sistemas de alto desempenho. Embora as métricas de produtividade e portabilidade não possam ser avaliadas de maneira quantitativa atualmente, os impactos de nossa abordagem mostram-se adequados para o cenário de aplicações de alto desempenho. A biblioteca MOGSLib serve como prova de que é possível criar um *design* orientado ao re-uso usando diretivas de linguagens modernas de propósito geral mesmo em sistemas de alto desempenho. Atualmente a biblioteca oferece suporte ao desenvolvimento de escalonadores globais ciente de carga que podem ser acoplados aos sistemas libGOMP e *Charm++*. O *design* da biblioteca é pensado para ser extensível e módulos desenvolvidos pelo usuário são a principal característica da biblioteca, permitindo que novas estruturas sejam adicionadas a sistemas de execução sem que haja a necessidade de alterações no código destas ferramentas. Como trabalhos futuros, pretendemos estender as abstrações do *ARTful scheduling* para acomodar diferentes taxonomias de escalonadores de carga como: (i) escalonadores distribuídos, (ii) escalonadores baseados em roubo de tarefas e (ii) políticas orientadas a grafos de dependências. Outras possibilidades de extensão é a adequação da biblioteca MOGSLib a outros sistemas de execução orientados a tarefas como *StarPU* e as versões mais recentes do *OpenMP*. Por fim, a crescente ênfase na área de alto desempenho em relação a *manycores* indica a possibilidade de que bibliotecas como a MOGSLib possam auxiliar o desenvolvimento de novas políticas de escalonamento

direcionadas a diferentes arquiteturas de aceleradores.

**Palavras-chave:** Escalonamento global. Sistemas de execução. Portabilidade de implementação. Qualidade de software. Reusabilidade.

**ABSTRACT**

Execution platforms for high performance computing are becoming diverse as a result of new architectures and tools to benefit from the parallel behavior of applications. These new options showcase performance enhancing opportunities for scientific and engineering applications. The execution platform diversity and the mapping of an application's tasks must be implicitly handled by runtime systems and their global schedulers as to enable the application performance and implementation portability. However, the development and integration of novel global schedulers into industry standards systems like *OpenMP* and *MPI* lack framework support. This work proposes a library, *MOGSLib*, to support the development and integration of global schedulers into different high performance runtime systems. *MOGSLib* employs reusable, independent and testable abstractions represented as C++ template structures to express scheduling policies, their relationship to runtime systems and the scheduling solution taxonomy. This approach allows a bottom-up development process based on the incremental composition of abstractions through template specializations. We evaluate the overhead of employing our portable policies in comparison to their system-native counterparts in two environments, the *Charm++* and *OpenMP* systems. Throughout our experiments we achieved development support for user-defined scheduling policies that can be implanted both as loop schedulers in libGOMP and load balancers in *Charm++* through the selection of abstraction implementations. We leveraged the overhead by employing workload-aware policies on molecular dynamics kernels which resulted in equivalent application makespan for both native and the *MOGSLib* scheduler versions. Ultimately, the flexibility to incorporate user-defined structures and scheduling policies into runtime systems with limited alterations into runtime system code bases hint that the definition of flexible global schedulers is available with neglible overheads even for high performance environments.

**Keywords:** Global scheduling. Runtime systems. Implementation portability. Code quality. Reusability

## LIST OF FIGURES

## LIST OF TABLES

## LIST OF ALGORITHMS

## LIST OF ABBREVIATIONS AND ACRONYMS

## CONTENTS

# 1 INTRODUCTION

Scientific and engineering applications designed for High Performance Computing (HPC) environments are constantly evolving. Such applications not only demand increasing levels of parallelism but also a way of reducing the complexity of expressing their parallel behavior in generic execution platforms. Moreover, a common problem in this class of applications is dynamic load imbalance originating from unpredictable interactions of an application's tasks, resulting in computations carried out in parallel to display irregular execution times. Modern solutions for these problems are reliant on parallel programming models and runtime systems to assist developers in creating parallel solutions and scheduling strategies. It is reasonable that the developer understanding and support from these tools (*programming models and parallel libraries*) are ranked as the most impacting features for reducing the development effort of parallel applications (WIENKE et al., 2016). However, performance portability of parallel components among different systems is still a problem and current solutions are mainly focused on the fine tuning of applications into different execution platforms (ASTORGA et al., 2018; PENNYCOOK; JARVIS, 2012; EDWARDS; TROTT; SUNDERLAND, 2014; HORNUNG; KEASLER, 2014).

A generic view of the underlying execution platform and strategies for distributing the application's workload on the platform are efficient ways to achieve application portability. The dynamic distribution of an application's tasks is a supported feature in most, if not all, parallel programming models. This service is performed by the runtime system's global schedulers, components that tackle the load imbalance problem by redistributing the workload dynamically following a policy. The logic performed by these schedulers is tightly coupled to the system's execution model. For instance, *OpenMP* applies a fork-join execution model and its loop schedulers are tasked to decide which threads are to execute each loop iteration (DAGUM; MENON, 1998). In the distributed work-pool execution model of *Charm++*, load balancers must decide in which PU the chares (*Charm++*'s work unit abstraction) will be allocated to perform their computation (ACUN; KALE, 2016). Other systems like *StarPU* apply a task-based and heterogeneous execution model where the schedulers allocate the tasks to the available processing units based on dependency graphs (AUGONNET et al., 2011).

## 1.1 PROBLEM DEFINITION

Runtime systems are packed with out-of-the-box scheduling strategies to cover the demands of most applications with regular workloads (static scheduling). The support for irregular applications is usually delivered through generic work-stealing or a central queue of tasks (LOPES; MENASCÉ, 2016). When generic techniques are not sufficient, new demands for scheduling policies drive the development of novel techniques as extensions

to the runtime system (CIORBA; IWAINSKY; BUDER, 2018). Moreover, only but a few systems have integrated development tools for user-defined schedulers (ACUN et al., 2016; AUGONNET et al., 2011) and, even then, the resulting implementations are limited to the Runtime System (RTS). Consequently, custom schedulers are integrated through runtime system modifications (KALE; GROPP, 2017) or alongside the application behavior (MEI et al., 2011). These processes carry out undesirable side effects from the creation of unofficial system branches to portability degradation.

HPC applications enjoy evolving and long lasting standards such as, but not limited to, *OpenMP* (DAGUM; MENON, 1998), *MPI* (WALKER; DONGARRA, 1996) and *BLAS* (BLACKFORD et al., 2002) to ease their development. These tools increase the applications code quality and allow flexible implementations that can be reused into future architectures with limited re-implementation efforts. Moreover, these standards are implemented by runtime systems responsible for delivering application performance portability by mapping the application workload to the execution platform. However, resource managing components for HPC systems lack similar developing support as applications. This degrades the RTS support for architecture features as they must be manually added into alternative libraries through intrusive methods (DURAND et al., 2013).

## 1.2 MOTIVATION

The absence of a standard scheduling framework or library for HPC applications contrasts with the abstraction of commonly used functionalities into their own composable domain-specific libraries (CHEVALIER; PELLEGRINI, 2008). This topic is hardly discussed in the literature and we attribute this scenario to the observed differences in the scheduling taxonomies between runtime systems (THOMAN et al., 2018). Consequently, a scheduling strategy optimizing a given metric will portray distinct implementations among runtime systems analogous to how performance portable applications express slight variations on kernel decomposition for each intended architecture and execution model. In fact, successful global scheduling strategies are specialized to their context, either by accommodating application- or system-specific features into their search algorithms (MEI et al., 2011). For that matter, it makes sense that state-of-the-art policies are developed alongside runtime systems as they become implicitly available to multiple applications and can benefit from internal data structures and introspection tools. However, this approach might become unsustainable as solutions rapidly become unsupported due to the lack of implementation portability to future versions or even other runtime systems (HARRELL et al., 2018).

We believe that the lack of a streamlined process for implementing user defined global schedulers in runtime systems is a problem that must be addressed for the design of future systems. This topic is currently foreshadowed by the performance-centric research of the HPC community and the small code size of scheduling policies when compared to

application's codes. Nonetheless, other computational research fields are tackling their version of this problem that manifests as generic scheduling for computer grids (DAIL; CASANOVA; BERMAN, 2002) and development support for user-defined policies in real-time operational system kernels (MOLLISON; ANDERSON, 2013). Moreover, application implementation portability is gaining momentum in HPC research with solutions being integrated into industry standards (PENNYCOOK; SEWALL; HAMMOND, 2018), runtime systems (AUMAGE et al., 2017) and as combination of compiler and high-level libraries (GROSSMAN et al., 2017). We envision that code quality and reuse is recently emphasized in HPC research due to systems becoming too complex and bloated with functionalities, optimizations and configuration options to squeeze performance out of modern architectures. Global schedulers are components that must be aware of the platform intricacies and it is likely that future runtime systems may benefit from readily available resource management components in the transition to novel architectures and execution models.

## 1.3 GOALS AND CONTRIBUTIONS

We draw inspiration from frameworks aiming to support the development of single-source applications in HPC environments to propose an implementation model for modular and system-independent global schedulers. We leverage the global schedulers' development and integration process within different runtime systems. Our proposal is the creation of a bottom-up approach for developing system-independent global schedulers based on the decomposition of its logic into specialized and composable structures. Therefore, this work delivers the following contributions to the state-of-the-art:

- *A novel set of abstractions and specifications to develop implementation portable global schedulers.* The *ARTful* specifications stands for Abstract, Re-usable and Testable scheduling components, which represent the core design pillars of our approach. We study the relationship between state-of-the-art runtime systems and their schedulers to configure this relationship as an abstraction we call *Scheduling Context*. This abstraction is used to decouple the global scheduler policy from features imposed by the runtime system, allowing for policies unbounded by external dependencies. This isolation enables the unitary tests of scheduling components to be performed prior to their integration on the system. Moreover, the construction of the global scheduler is obtained by the composition of its components adapted to the target context. Consequently, with a finer abstraction granularity, it is possible to reuse the scheduling elements into solutions targeting the same context or applying policies with similar taxonomy.
- *A novel framework for developing ARTful global schedulers through a bottom-up approach entitled MOGSLib.* The Metaprogrammed-Oriented Global Scheduler Li-

brary (MOGSLib) stands as our technical contribution to the state of the art. It provides necessary tools to develop, test, compose and later adapt global schedulers into runtime systems. This process is absent in state-of-the-art scheduling frameworks which mainly focus on the development of schedulers for a single runtime system based on top-down approaches. MOGSLib is a *C++* 14 header-only library of templated structures that support the expression of global schedulers components through generic meta-programming. This approach enables the composition of schedulers to be expressed as static rules, mitigating runtime overheads and allowing loose datatype definitions during development without sacrificing strong type check at compilation.

- *The integration of MOGSLib into the LibGOMP and Charm++ runtime systems.* We integrate MOGSLib into two distinct runtime systems with different scheduler taxonomies. Our goal is to evaluate the scheduling overheads of a portable global scheduler when compared to native scheduling solutions. We experiment our approach by re-implementing native workload-aware strategies through MOGSLib and evaluate their performance when balancing synthetic benchmarks and application kernels. The portability of single-source schedulers into different runtime systems is unprecedented and our work aims to study its benefits and drawbacks.

It is worth mentioning that our work refrains from introducing new scheduling techniques into the evaluated runtime systems. Our goal is rather to provide development support for new strategies in both systems. As such, our evaluation process quantitatively compares existing strategies already implemented on the runtime system against a re-implementation outside of the system's library scope in regards to schedule overhead. A brief discussion about portability, code quality is also present but, due to the lack of interesting metrics for HPC internal components (WIENKE et al., 2016), it is focused on a qualitative analysis. This work has been presented in the WSCAD-2018 Brazilian national conference (SANTANA et al., 2018) for introducing the MOGSLib library and new contributions are being developed to generate a new publication.

## 1.4 WORK ORGANIZATION

The remainder of this paper is organized as follows: Chapter 2 presents the necessary background for our problem. Chapter 3 discusses the related work. Next, in Chapter 4 we expose our approach. We present our implementation in the target systems in Chapter 5. In Chapter 6 we explain our experimental method and results. Finally, we conclude this work in Chapter 7.

# 2 BACKGROUND

In this chapter we discuss the background on which this work relies. First, we present a definition of the global scheduling problem. Next, we introduce the basic concepts of the two programming models this work encompasses, *OpenMP* and *Charm++*. The global schedulers used in our experiments are discussed afterwards. Then, we briefly discuss how global scheduling solutions are classified in parallel and distributed systems through their topology. Finally, some basic notion of *C++* template meta-programming is presented as to aid the understanding of the techniques applied into MOGSLib to implement the *ARTful* scheduling model in *C++*.

*Global scheduling* is described by Casavant and Kuhl as the problem of defining *where* to run a *task* (consumer), leaving the decision of *when* to run a task to local scheduling (CASAVANT; KUHL, 1988). The generic scheduling problem is known to be NP-Hard (GRAHAM et al., 1979) and solutions often rely in heuristics and approximations. Moreover, the practical definition of global schedulers differs from system to system due to differences on the parallel decomposition of applications. The following sections further detail the specific scheduling problems on the programming models evaluated in this work. Overall, this work targets the application level distribution of parallel units of work, often referred as tasks, into the execution platform generic resources (processing elements).

## 2.1 THE *OPENMP* PROGRAMMING MODEL

*OpenMP* is an industry standard Application Programming Interface (API) for parallel programming on single node shared-memory architectures. It is available for the Fortran and C family languages through annotations in the application code to declare parallel zones and loops. The standard employs a fork-join execution model with threads managed by the runtime system as the default form of processing parallel loops. The user can configure some runtime parameters through annotations and system variables such as the selection of the scheduling policy, through the **OMP_SCHEDULE** system variable. Recent versions of *OpenMP* allow the definition of independent tasks to be executed in parallel and also support directives to offload tasks and loop iterations to manycore and GPU devices (SUPINSKI et al., 2018). In this work, we focus on the parallel loop interface of *OpenMP* solely targeting the CPU cores and how the runtime system deals with the distribution of its work units (loop iterations) among the processing units (processor's cores).

Figure 1 displays a fragment of *C++* code with *OpenMP* annotations. This example showcases the multiplication of two arrays by their indices as executed in parallel by *OpenMP*. The code takes two array parameters, $b$ and $c$, with size $N$ and multiplies their values by their index storing the result in $a$. *OpenMP* eases the creation of parallel

```
1     int * vector_mul ( int *b, int * c, int N ) {
2       int *a = new int[N]();
3
4       #pragma omp parallel for schedule(static)
5       for(int i = 0; i < N; ++i)
6           a[i] = b[i] * c[i];
7       return a;
8     }
```

Figure 1 – Example of *OpenMP* parallel loop.

behavior, as seen on line 4, through the use of annotation. Specifically, the *parallel for* directive informs the runtime that the next for loop is to be executed in parallel and the iterations must be distributed by the *static* scheduling policy. Despite the lack of user-defined policy support, there are a few configurable native policies in *OpenMP* which are briefly described as follows:

- **Static Loop Scheduler**: this policy partitions the loop into evenly sized chunks (sets of continuous loop iterations) and assigns them in a round-robin fashion to each processor. This strategy incurs in little overhead but fails to provide irregular applications with a balanced distribution.
- **Dynamic Loop Scheduler**: this policy assigns chunks of iterations to each processor on demand until there are no more chunks left to be processed. This strategy incurs in high overhead as threads must constantly call the runtime to request work. However, the distribution is capable of considering the application irregularity as well as other sources of load imbalance implicitly.
- **Guided Loop Scheduler**: this policy behaves similarly to the dynamic policy. However, it dynamically adjusts the chunk size, starting with large chunks of iterations and slowly decreasing the size. This approach aims to find a balance between the scheduling overhead and load imbalance by assigning more chunks by request and performing a fine tune in the last loop iterations.

The loop schedulers in *OpenMP* are tasked with providing a mapping function $M : L \rightarrow T$ which maps the set of loop iterations $L$ to the set of *OpenMP* threads $T$. Developers aiming to expand *OpenMP*'s policy set must alter the inner structures of a runtime system providing the backend for *OpenMP* pragmas to the compiler. One example of such runtime is the GNU library for *OpenMP* (libGOMP) that provides the default *OpenMP* annotations for the gcc compiler[1]. In libGOMP, threads and loop iterations are indexed by increasing unsigned integers ranging from 0 to $|T| - 1$ and $|L| - 1$ respectively. Iterations can be assigned to *OpenMP* threads by associating their ids in an array structure that represents the thread team workload. The addition of new policies in such way requires developer knowledge of the library internal mechanisms. Additionally,

[1] Available in https://github.com/gcc-mirror/gcc/tree/master/libgomp

the resulting library is forked from the official version and becomes increasingly hard to maintain as new versions are pushed by the community.

## 2.2 THE *CHARM++* PROGRAMMING MODEL

*Charm++* is a distributed parallel programming model that creates a *C++* idiom through library calls to support an asynchronous message-driven execution model. Parallel applications in *Charm++* are decomposed as migrateable *C++* objects called **chares** which represent a basic unit of parallel computation. *Charm++* is best used when over-decomposing an application into much more chares than processing units, allowing the runtime system to detect load imbalance and attempt to correct it by migrating chares.

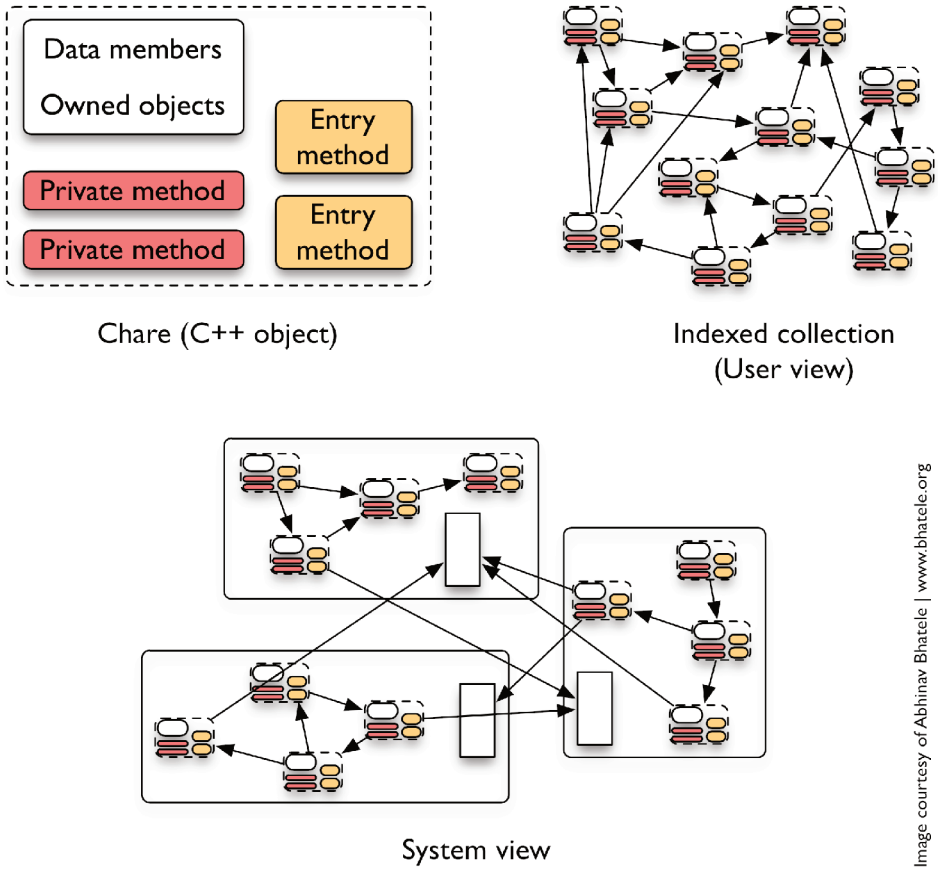


Figure 2 *Charm++* programming model overview.

The runtime system is able to abstract chare placement and their redistribution from the application programming interface. An overview of this programming model is depicted in Figure 2 with the different views it provides. In this figure, it is possible to observe the over-decomposed model through the declaration of a chare collection. The chare placement is defined by the system which allocates a subset of chares to each processing element (delimited by the box). Communication is handled by the system

which directs messages to the correct chare through a local scheduler present in each processing element (inner white boxes).

*Charm++* portrays a mature framework for the development and integration of load balancers comprised of distributed and centralized policies. One of its main characteristics is that *Charm++* automatically instruments the application in order to obtain relevant metrics for scheduling the chares. Moreover, this and other platform information can be used by the system's global schedulers to make an informed decision based on the recent past of chares.

Global scheduler developers in *Charm++* must understand the system's Load Balancing Database (LBDB) and the execution flow of both application and load balancing. The former is a *C++* language-level structure comprised of the aforementioned instrumented application and platform data. The structure doubles as a communication layer between the system and the load balancers, allowing one to be apart from the other's internal routines. The execution flow of load balancing starts when a chare calls the runtime API **AtSync** method and start a synchronization process among the chares. Next, the control is passed onto the global scheduler which analyzes the execution platform state through LBDB and decides the new application chare mapping. Finally, the system automatically migrates the chares based on the scheduler decision and resumes the application.

*Charm++* support global schedulers with different taxonomy features. These include but are not limited to workload-aware centralized policies and distributed policies that evaluate communication over the platform topology (FREITAS et al., 2018). Regardless of the strategy, load balancers in *Charm++* act as an implementation of a mapping function $M : C \rightarrow P$ where $C$ is the set of application chares and $P$ is the set of processing units within the distributed platform. *Charm++* implements this decision at language level through the association of every chare and processing unit to an integer id ranging from zero up to $|C| - 1$ and $|P| - 1$, respectively. The system then provides an *assign* function encapsulated in the LBDB structure that registers a chare migration to a processing unit. This function must be called once for every migrating chare $c \in C$ and its parameters are the chare and its new target processing element ids, respectively. Despite the generic approach, *Charm++* load balancers have proven to be successfully applied even on large class of applications such as NAMD (MEI et al., 2011) and ChaNGa (JETLEY et al., 2008).

## 2.3 THE EVALUATED GLOBAL SCHEDULING POLICIES

We set out to evaluate scheduling policies that can be employed on different parallel programming models as to tackle realistic use cases for implementation portability. With that in mind, we explore the workload-aware class of global schedulers as they can be applied to imbalanced scenarios such as molecular dynamics applications found

Algorithm 1 – *Longest Processing Time First* scheduling algorithm

**Require:** $T$, $P$ and $\omega$
**Ensure:** $S$

1: $S \Leftarrow \emptyset$
2: $T \Leftarrow \text{max\_heap}(T)$
3: $P \Leftarrow \text{min\_heap}(P)$

4: **while** $T$ is not $\emptyset$ **do**
5: $\quad t_i \leftarrow \text{pop}(T)$
6: $\quad p_j \leftarrow \text{top}(P)$
7: $\quad S \leftarrow S \cup \{t_i \rightarrow p_j\}$
8: $\quad w(p_j) \leftarrow w(p_j) + w(t_i)$
9: $\quad \text{update}(P)$

throughout multiple programming models (SEDOVA et al., 2018). This class of scheduler can use input data from several sources from static and dynamic performance models to user informed data. As such, it stands out as an initial study for leveraging portable implementations of global schedulers.

Workload-aware strategies for global scheduling rely on the explicit knowledge over the cost of processing each work unit. This means that there must be a function $\omega$ that takes a work unit $u_i$ from the set of application's work units $U$ as parameter and outputs a processing cost $c_i \in \mathbb{N}^+$ such that the total application processing cost $C$ is calculated as:

$$C = \sum_{u_i \in U} \omega(u_i)$$

The objective of this class of schedulers is to attempt an even distribution of workload among processing units. Note that the workload is associated to arbitrary metric units and their values may represent any metric that is meaningful to the problem (*e.g.* walltime, computations or application-specific abstractions). Moreover, there are no restrictions imposed regarding the origin of the workload data. This means that the cost function $\omega$ can be defined by the user's knowledge over the application or carried out by dynamic observations as in *Charm++*.

The default workload-aware policy in the *Charm++* system is similar to the Longest Processing Time First (LPTF) scheduling policy in CPU scheduling. Its behavior in *Charm++* is depicted in Algorithm 1. The policy takes as parameters: (i) a set of tasks $T$; (ii) a set of processing elements $P$ and (iii) a function $\omega$ to obtain the workload of the processing units and tasks. The policy organizes the set of tasks in a max-heap (highest value first) and the set of processing elements in a min-heap (lowest values first) in regards to their workloads. Then, the algorithm iteratively assigns the heaviest task to the least loaded processing element. The assigned processor's workload is incremented in

Algorithm 2 – *BinLPT* scheduling algorithm

**Require:** $A$, $k$ and $n$
**Ensure:** $P$

1: $C \leftarrow$ compute-chunks$(A, k)$
2: sort($C$, descending order)
3: $P \Leftarrow$ min_heap$(P)$

4: **for** $i \leftarrow 0$ to $N$ **do**
5: $\quad T_i \leftarrow 0$
6: $\quad P_{T_i} \leftarrow \emptyset$

7: **for** $i \leftarrow 0$ to $|C|$ **do**
8: $\quad T_j \leftarrow \min T$
9: $\quad P_{T_j} \leftarrow P_{T_j} \cup \{C_i\}$
10: $\quad T_j \leftarrow T_j + \omega\{C_i\}$

the same amount of the tasks' load and the heap is updated to re-organize its elements.

The LPTF scheduler in *Charm++* is known as the *GreedyLB* load balancer. It is the default load balancer in the system and it is specially useful due to its use of the dynamic observations made by the runtime system. The policy itself is not configurable and portrays a static behavior although the runtime system allows the definition of scheduling parameters like the load balancer invocation frequency.

*OpenMP* has no native workload-aware schedulers. The system relies on its guided and dynamic loop schedulers to tackle load imbalance. However, these strategies may incur in scheduling overheads and might be hard to fine-tune for portable applications. These characteristics led to independent approaches for tackling imbalanced applications with predictable workloads. In this work we study one of such approaches, the *BinLPT* loop scheduler evaluated in (PENNA et al., 2017). This policy is implanted on a publicly available custom version of libGOMP [2] through hacks and additions to the *OpenMP* standard functions.

Similarly to the LPTF scheduler, *BinLPT* employs a greedy approach to distribute the application's workload among the processing elements. The *BinLPT* behavior is summarized in Algorithm 2. It uses a function, *compute-chunks*, to calculate the average weight of the application when divided in up to $k$ chunks. This function also creates the chunks by packaging contiguous iterations until their combined workload is greater than the calculated average weight per chunk. The policy sorts the chunks in descending order (line 2) and iteratively assigns all iterations in the largest chunk to the least loaded thread (lines 7 to 10). *BinLPT* outputs a multiset that enumerates which chunks were attributed to which threads.

The policy requires access to the workload estimations for each loop iteration.

[2] www.github.com/lapesd/libgomp

This data must be forwarded to the policy which is implemented within the libGOMP library. To achieve this, new functions needed to be added into the library by the policy developers. Namely, a new procedure called *omp_set_workload* was added to the library to reference the workload data (an array of integers) from the user space from within the library space. As such, this enhanced library is capable of achieving workload-aware load balancing using user-informed workloads. Unfortunately, new policies capable of making use of these additions would require further alterations to the library, a process that would not scale well with novel strategies requirements.

## 2.4 THE GLOBAL SCHEDULER TAXONOMY

The categorization of scheduling solutions aids the understanding of how parallel systems deal with the task distribution problem. Moreover, the scheduler taxonomy analysis helps to understand the differences and similarities between schedulers in different programming models. In this work, we refer to the scheduling solution taxonomy proposed in (LOPES; MENASCÉ, 2016) originating from the analysis of schedulers for distributed systems.

The aforementioned work proposes two taxonomies, one for scheduling problems and another for scheduling solutions. The scheduling problem taxonomy is related to the characteristics of the parallel context environment. its features leverage the application, execution platform and other scheduling constraints in the definition of three categories: (i) *workload*; (ii) *resources* and (iii) *scheduling requirements*. The scheduling solution taxonomy categorizes schedulers through how the scheduling problem is addressed. Its characteristics are also organized in the definition of three categories: (i) *optimality*; (ii) *operation* and (iii) *topology distribution*.

Parallel programming models are designed to serve as specialized tools for solving a specific set of problems. As such, runtime systems implementing the model's API are designed around these problems which influence the execution model. Consequently, global schedulers are tied to the system's definition of *workload*, *resources* and *requirements*. On the other hand, scheduling solutions in a system can display a different set of configurations regarding its taxonomy (*optimality*, *operation* and *topology distribution*).

As an example to the aforementioned definitions, *Charm++* is designed for a distributed computation scenario where nodes are possibly irregular in regard to processing power. Consequently, its schedulers are also designed to solve a problem with these scheduling problem characteristics. Indeed, the system offers native support for load balancers to make both distributed and centralized decisions (*topology distribution*). Moreover, when creating centralized policies for clusters with identical machines, *Charm++* dynamic load balancers have similar taxonomies to *OpenMP* loop schedulers (online, non-optimal and centralized). These overlapping scheduling topologies create room for a generic global scheduler definition that functions in a subset of problems in both systems.

We attribute the scheduling solution taxonomy features to the capabilities of the global scheduler's policy (the schedule search routine). These policies frequently tolerate flexible definitions of problem taxonomies and may be specialized to perform on different concrete definitions (*e.g.* dynamic vs user-informed workload, heterogeneous vs homogeneous architectures). On the other hand, the problem taxonomy influences the runtime system scheduling API and consequently how the policy is implanted. We envision that it is possible to dissociate both taxonomies in a scheduler implementation, in practice, by abstracting the relationship between the runtime requirements and the scheduler. As such, our goal is to create scheduling policies that may be unaware of the system characteristics. Ultimately leading to a policy that is defined in generic terms and specialized when implanted in a runtime system that requires a slightly different behavior (*e.g.* testing for irregular machines instead of assuming that all nodes are equal).

## 2.5 *C++* GENERIC PROGRAMMING

The C language family is commonly used for developing high performance code due to its adherence in most parallel programming models. We chose to implement our global scheduler library, MOGSLib, in *C++* as both the *OpenMP* and *Charm++* programming models are based on the C language family. Moreover, *C++* has a robust and growing set of directives to address the generic programming style of development. Although the *ARTful* scheduling model is not limited to this style of programming, similar approaches for the generic expression of behavior in HPC components are observed in other works (AUMAGE et al., 2017; MOLLISON; ANDERSON, 2013).

Generic programming in *C++* is often referred as *meta-programming*. This programming style is based around the declaration of templates for functions and data structures that support multiple definitions for different data types. Templates are generic definitions of behavior represented as flexible-typed algorithms that are translated to actual code by the compiler when the latter identifies its usage in the source code. This means that even though *C++* is a statically typed language, it is possible to write generic code through a template definition of functions and structures.

Template functions and structures can be specialized to work with a set of data-types through the explicit definition of the template code when given a specific type parameter. This process is called template specialization and is widely used in the standard template library of the *C++* language (STL). Through these language directives it is possible to express a default generic behavior for functions addressing different data-types and specialized behavior when necessary. The actual code associated to the function call or structure usage is defined during compilation when the compiler analyzes the data types in the expression and chooses the most fitting implementation for the template.

Figure 3 displays a generic code that computes the distance between 2 points in a 2D plane. The template procedure with generic code to express the calculation is depicted

```
1     template<typename T>
2     int distance(T a, T b) {
3       return abs(b - a);
4     }
5
6     struct Point2D {
7       int x,y;
8     };
9
10    template<>
11    int distance<Point2d>(Point2d a, Point2d b) {
12      int dx = pow(b.x - a.x);
13      int dy = pow(b.y - a.y);
14      return sqrt(dx + dy);
15    }
16
17    int d = distance(param1, param2);
```

Figure 3 – Template specialization on *C++*.

in lines 1 through 4, indicated by the expression in line 1. The *distance* procedure may take any data type $T$ and the result of the distance is carried out by the absolute difference of its parameters $a$ and $b$ using the arithmetic operator for operands of type $T$. This is the expected behavior when calculating the distance between two numbers, *integers, 16/32/64 bit precision floating points* and so on. Lines 10 to 14 showcase how the *distance* procedure can be specialized to work under a data-type, *Point2D*, that represents a bi-dimensional point given its definition in lines 6 to 8. The function is defined with the same signature but this time without the generic notation on the template expression (line 10). This last routine calculates the Euclidian distance between the two points maintaining the same semantics and syntax of the original *distance* routine. As a consequence of template specialization, the expression portrayed in line 17 yields a computation that is dependent of the data-types of the parameters *param*1 and *param*2. If the compiler evaluates their types and detects that they are *Point2D* objects, the specialized function is translated into the code, otherwise the generic function is selected.

Template meta-programming in *C++* is specially useful when combined with constant expressions. These expressions use static operands that are evaluated during compilation time and thus can be used to employ boolean logic to the template type-resolution step of compilation. This allows extra semantics for the selection and specialization of template functions and structures.

Figure 4 displays an example of template selection through constant expressions. This segment of code displays a structure to represent the workload *data* of a set of $n$ work units in lines 1 to 4. Runtime systems like *Charm++* can supplement global schedulers with workload data by instrumenting the application, however, systems like *OpenMP* have no native runtime support for evaluating the application workload. As such, the definition of the workload in *OpenMP* must be informed by the user and forwarded to the global scheduler as in the original *BinLPT* implementation. There is no generic definition on how to fetch the workload input of global schedulers in generic programming models.

```
1   struct Workload {
2       int n;
3       int *data;
4   };
5
6   template<bool U>
7   Workload workloads() {
8       return User::getWorkloads();
9   }
10
11  template<>
12  Workload workloads<false>() {
13      return RuntimeSystem::getWorkloads();
14  }
15
16  constexpr bool usr_defined = usr_workloads || rts_no_support;
17  workloads<usr_defined>();
```

Figure 4 – Template implementation selection on *C++*.

Instead, the function defined in lines 6 to 9 portrays a default approach of querying such data where the template parameter (line 6) is associated to a boolean value rather than a data-type. The default operation is to access the *User* namespace and call the *getWorkloads* function. However, a specialized function for when the template parameter is statically evaluated as **false** (user is not supplying the data) is available throughout lines 11 to 14, where the *RuntimeSystem* namespace is accessed instead. This behavior enables the definition of static rules for expressing which implementation should be picked by the compiler given static constants. As an example, in lines 16 and 17, the *workloads* function is called and the informed template parameter is a constant expression composed of two static values *usr_workloads* and *rts_no_support*. Ultimately, this can be used to select the workloads informed by the user whenever the user explicitly informs the scheduler to do so (*usr_workloads*) or when the runtime system has no support for this feature (*rts_no_support*). Although this is a simplification of the problem, template meta-programming is a powerful tool that allows the definition and selection of code fragments that will compose the final global scheduler implementation. Through this language mechanism, developers can express generic scheduler behavior that can be adapted to multiple runtime systems through constant rules evaluated during compilation which ultimately incurs in little runtime overhead.

# 3 RELATED WORK

Our work is focused on the development support and implementation portability of scheduling components for different contexts and runtime systems. As far as we know, through our research efforts, no other work for high performance environments share a similar goal. However, the search for an expressive way of developing modular and less complex HPC software components is manifested in similar work that target other components and characteristics. We consider as related work any research efforts, in HPC, that enables the user to define and aggregate their own software components through the use of abstraction over the application or runtime system domain. Despite the differences in goals and solutions, these techniques are subject to the same constraints of the high performance environments which often must abdicate from classic solutions in software engineering due to performance overheads.

Application portability in HPC environments is a research topic that study approaches for enabling application implementations to target multiple platforms. These studies do not address the development and/or portability of runtime system internal components and thus are not directly comparable to our work. Indeed, most portability efforts in HPC are based on performance portability which is achieved by the expression of generic application behavior that can be interpreted in other execution models or architectures. Examples of such work are programming models like *OpenACC* (COMMITTEE et al., 2015), *OpenCL* (STONE; GOHARA; SHI, 2010) and performance portability libraries such as *Raja* (HORNUNG; KEASLER, 2014) and *Kokkos* (EDWARDS; TROTT; SUNDERLAND, 2014). These tools target the application behavior portion of the HPC software and are mostly directed to provide a single algorithm implementation that can be adjusted to function in different runtime systems while preserving its performance. We refrain from further discussing these approaches as they deviate from our goal of providing users with the capability of developing components that expose implementation portability to different environments.

We draw inspiration from work that employ indirection layers and abstractions to configure solutions in the form of a set of attachable components that can be expanded by the user. These components are specialized and can be swapped to adapt the solution into a different set of constraints and tool as the swapped component performs the same procedure. Aumage et al. (2017) uses this concept to propose a component system called *COMET* that abstracts computational steps in parallel applications that can be mapped to task-based execution systems. The *COMET* system is used to express the application behavior as a graph of component interfaces that run atop the *StarPU* runtime system. The application segments are represented as meta-tasks and their implementations can be swapped during runtime by selecting a concrete component that will perform its computation as a task in the system. Grossman et al. (2017) proposes another work in this line in the form of *HiPER*, a description of a modular runtime system for exascale computers.

*HiPER* is a highly pluggable system with an unified scheduler capable of balancing the active modules activity. Modules can be added by registering lambda functions in the system API, allowing the expression of user modules and functionalities as anonymous functions in the system that are made available to the remainder of the system through callbacks. These techniques provide users with a higher degree of customization and control over the tools, allowing the development of experimental modules faster.

On meta-programming techniques for adapting parallel applications, (MERRILL et al., 2012) proposed the design methodology of rules (policies) for how to interpret GPU kernels based on the template library of *C++*. Their work is motivated by differences among GPU micro-architectures and fine tuning configurations that cause application performance variations in different environments. Their approach is directed to making kernels as template-dependent functions, so that optimizations related to memory or other sensitive performance traits can be unbounded. The resulting kernels are generic expressed and specialized by the policy parameter during compilation.

The demand for variety and user-defined scheduling is a research topic in the scope of real-time operating system (RTOS). Similarly to HPC runtime systems, in RTOS, the kernel must abstract basic functionalities to applications and scheduling is one of them. (MOLLISON; ANDERSON, 2013) proposed that user-defined scheduling policies could be implemented in user-space instead of kernel-space. They developed a library with a common higher level API the user can manipulate independently of the underlying kernel. Those higher level directives are translated by a driver they developed and forwarded to the kernel and C POSIX library function calls. Their solution enables schedulers to be developed out of the kernel-space with abstract implementations for base functions which involves thread locking, synchronization and other functionalities. The overhead of the technique was acceptable even on real-time constraints, which is one of the most critical metric for schedulers in real-time operating systems.

Table 1 correlates the aforementioned related work to this work. The differences in scope and objective do not allow a direct comparison among these efforts which makes a quantitative discussion impossible. As such, we analyse common characteristics of these works rather than their impact on their respective contexts. Indeed, each effort is based on reusability and complexity reduction and this is only achieved by the abstraction and automated management of a subset of a problem characteristics. Ultimately, we compare the related work to ours through the following three characteristics:

- **Abstraction Model**: how elements in the solution's domain are abstracted from the user.
- **Composition**: how the work aggregates the abstracted elements and the user-defined elements to form a complete software component. We identified two main approaches in the evaluated works. One approach is based around the use of a tool for assisting the user to explicit an assembly of the relevant abstractions using native

*language-level-directives* (LLD). Another approach is to create an abstraction layer and interface functionalities through a novel library that exposes an API available to be user segments.

- **Extensible**: if the work explicitly provides any *framework* or tool for the user to extend the proposed solution.

| **Work** | **Abstraction Model** | **Composition** | **Extensible** |
|---|---|---|---|
| Aumage et al. (2017) | Component System | API | Yes |
| Grossman et al. (2017) | Lambda Functions | API | Yes |
| Merrill et al. (2012) | Generic Programming | LLD | - |
| Mollison & Anderson (2013) | Indirection Layer | API | No |
| **This work** | Generic Components | LLD | Yes |

Table 1 – Comparison of related work.

In regards to the abstraction model, each work implemented its own technique. This is expected as each work is proposed for a different scenario containing its own challenges and constraints. Our work employs a mixture of generic programming with an expandable component system to decompose a global scheduler into smaller abstractions. In regards to the composition technique, most works employed a solution in the form of a library accessible through an API. In this category, our work falls in the same class as the (MERRILL et al., 2012) approach, using language level directives and tools to assist the compiler to assemble the solution. Indeed, application interfaces offer dynamic reconfiguration and flexibility but the use of language directives and static techniques are important to mitigate runtime overheads which are crucial in HPC environments. Finally, when leveraging the explicit extension support of the aforementioned works, most do offer some kind of technique or guideline on extending the solution. Both works that did not offer such support are (MERRILL et al., 2012) and (MOLLISON; ANDERSON, 2013). The former is a design methodology study and thus does not offer a concrete implementation for users to expand and the latter is a work in development which still has little support for the integration of user defined abstractions. In summary, our approach offers a blend of current techniques to experiment in a novel scenario, the global scheduling of applications in the scope of HPC runtime systems.

# 4 SOLUTION

In this chapter, we discuss the *ARTful* scheduler specifications and their relationship. Later, we showcase an implementation, MOGSLib, which employs *C++* metaprogramming and object-oriented directives to construct global schedulers that adhere to the *ARTful* specifications. Indeed, the *ARTful* specifications can be implemented with other programming techniques and our design decisions are discussed in this chapter.

## 4.1 THE *ARTFUL* SCHEDULING SPECIFICATIONS

The *ARTful* scheduling specifications are derived from the study over the anatomy of global scheduler in the *StarPU*, *OpenMP* and *Charm++* runtime systems. A representation of native scheduling solutions in these systems is depicted in Figure 5. This approach represents the state-of-the-art scheduling solutions, where the global schedulers are integrated components within the runtime system. Functionalities and data structures in the runtime system are made available for global schedulers so their policies can collect data according to the system's scheduling solution taxonomy. In robust runtime systems, these functionalities may offer a structured view of the execution platform, a communication graph of the application tasks and, in simpler systems, these may be limited to the number of active processing units and work units. Ultimately, we envision that the lack of an explicit definition of the relationship between runtime and scheduler, the *scheduling context*, hinders the scheduler's implementation portability.

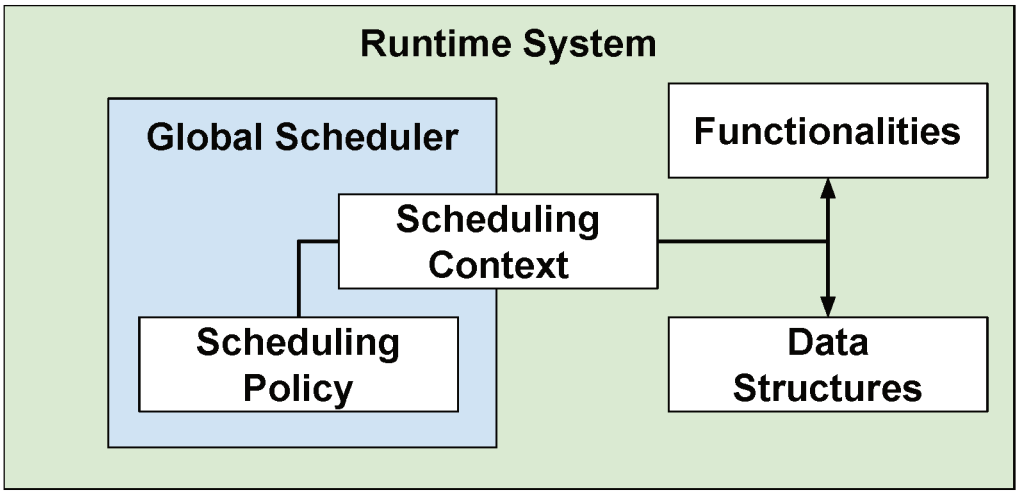


Figure 5 The runtime system native scheduling solution anatomy.

The *ARTful* scheduling specifications aims to specify a set of implementation rules to be followed with the intent of decoupling global schedulers from the runtime system. *ARTful* is an acronym for **Abstraction**, **Reusability** and **Testability** which are the main concepts behind the specifications. Our approach is to decompose global sched-

uler implementations into a set of abstractions that can be individually validated through tests and composed into different global scheduler assemblies. Indeed, the final goal of the specification is to support the development of scheduling solutions with the same set of taxonomy features than the existing system-native implementations. Moreover, the specifications enable the implementation portability of these abstractions allowing for different compositions which generate different scheduling traits. The *ARTful* specifications are described in the following sections and are constituted by the following abstractions: (i) the scheduling policy; (ii) the global scheduler; (iii) the scheduling context and (iv) the runtime system adapter.

### 4.1.1 Scheduling Policy Abstraction

At the core of any global scheduler is a set of algorithms to decide the processor mapping of the application work units. Those algorithms, namely scheduling policies, take an arbitrary set of input data to calculate a global schedule that fit into its optimization objectives (*e.g.* performance, power consumption, communication). A given policy might require input data to analyze the platform distributed topology (JEANNOT et al., 2013), power consumption data from each of its processors (FRASCA; MADDURI; RAGHAVAN, 2012), the workload of each work unit (PENNA et al., 2017), the memory architecture of the machine (DURAND et al., 2013) and others. Scheduling policies are distinct in objective and strategy, making it difficult to describe an unified interface for this entity without sacrificing simplicity or flexibility. Nonetheless, the scheduling policy can be described by its common output, the decision of a global schedule of an application's tasks.

The *ARTful* scheduling specifications for the definition of a scheduling policy takes in account the existing differences among strategies requirements and their common output. The following are the *ARTful* specifications for the implementation of a scheduling policy as a software component in the global scheduler assembly:

1. It must have only one procedure in its public interface;
2. It must not record state;
3. The output of its procedure must be a global schedule;
4. This procedure must not be limited in number nor ordering of input parameters;
5. The input data types must be flexible to account for different language definitions of what they represent.

The first, second and third specifications characterize the scheduling policy component as an entity with a single purpose in the global scheduler assembly. The third and fourth specification accounts for the well-defined objective of the component and the component flexibility to define multiple approaches respectively. Finally, the fifth specification is a pre-requisite for implementation portability. A concrete data type representation for

a scheduling input may differ from a system to another. As such, data types to represent concepts such as workload, communication, work unit identifiers, topology and others must be flexible as to account the discrepancies among contexts. With this component specification, it is possible to implement scheduling policies that operate on generic definitions of input data and can be used in global schedulers as long as the latter is able to provide the policy with its required set of data type definitions and input.

### 4.1.2 Global Scheduler Abstraction

The *ARTful* global scheduler specifications enables a clear distinction between the entities that represent the scheduling context and the global scheduler. In fact, our main difference from the native scheduling solution approach, previously depicted in Figure 5, is the abstraction of the scheduling context from the global scheduler logic. With contextual functionalities and scheduling policies abstracted, this entity takes the role of expressing which operations must be provided in any given context for supporting its scheduling strategy.

The global scheduler abstraction performs the control logic of fetching the input data through API calls to the scheduling context. If the policies define the algorithmic approach to solve the mapping problem, the *global scheduler* abstraction defines the shape of the scheduling solution. The *ARTful* specifications for the global scheduler entity is as follows:

1. It must have only one procedure in its public interface;
2. The procedure must output a global schedule;
3. The procedure must employ at least one scheduling policy;
4. The procedure signature must be syntactically equal for all global schedulers;
5. The global scheduler must depend on the scheduling context to provide any functionality that defines a taxonomy feature unrelated to the policy;

The first and second specifications define the global scheduler abstraction as a component that has only one responsibility. Moreover, its role is specified by the second and third specifications as an abstraction layer for interfacing with scheduling policies. The fourth specification is useful for creating a common syntax for calling global schedulers. Indeed, this feature is necessary for creating a low level collection of such components that can expose multiple approaches for a given scheduling problem. Finally, the fifth specification assures that global schedulers are both dependent and dissociated from the scheduling context. *ARTful* enables portability by allowing an implementation to cope with the context's scheduling taxonomy features without restricting the implementation to their semantics. Ultimately, these specifications allows global schedulers to be applied to different contexts as long as those can provide the scheduler with implementations for its required functionalities.

```
1: procedure LPT(wl, nt, npu)
2:     pu_load ← ARRAY(npu)
3:     map ← ARRAY(nt)
4:     for i ← 0; i < nt; i ← i + 1 do
5:         tid ← MAX(wl)                          ▷ Find the longest processing time task
6:         pid ← MIN(pu_load)                     ▷ Find the least loaded processing unit
7:         map[tid] ← pid                         ▷ Map the task to the processing unit
8:         pu_load[pid] ← pu_load[pid] + wl[tid]  ▷ Update the processing unit workload
9:         wl[tid] ← −1
10:    return map
```

Algorithm 3 – The Longest Processing Time (LPT) policy.

As an example, consider the longest processing time (LPT) policy depicted in Algorithm 3. This is a workload-aware policy that requires, as input, the number of processing units in the system, `npu`, the number of tasks in the application, `nt`, and the workload of each task, `wl` (line 1). The policy iteratively finds the heaviest task (line 5) and the least loaded processing unit (line 6). Then, it associates their identifiers to create the task mapping (line 7), ignores the mapped task (line 8) and updates the workload of the processing unit (line 9). An *ARTful* compliant implementation of this policy should express the requirements (algorithm parameters) to be fulfilled by the global scheduler component. Moreover, the processing unit count and the workload of each task must be retrieved from the context (user, runtime or application) as the global scheduler entity can not implement these functionalities. Regardless of which data are needed, the global scheduler is only responsible for requesting this data from an abstract context which, in turn, will implement how the data is accessed.

### 4.1.3 Scheduling Context Abstraction

The scheduling context abstraction represents the environment where the scheduling solution will be applied and all its necessary scheduling related functionalities. In some cases, schedulers may rely on information provided by the user to aid its decision (BHATELE et al., 2011), extract data directly from the application (FATTEBERT; RICHARDS; GLOSLI, 2012; MEI et al., 2011) or even start communication with other scheduling decision entities (FREITAS et al., 2018). These functionalities are provided by runtime systems through a scheduling API. Indeed, these procedures are not contemplated in the other *ARTful* abstractions despite being important parts of the scheduling solution taxonomy. However, each runtime system portrays different semantics for these operations and thus it should be dissociated from the logic portrayed in the policy.

The *ARTful* scheduling context abstraction role is to encapsulate a set of function implementations required by a global scheduler to execute its policies. These implementations may portray a subset of the scheduling solution features of the system. An example

of such features can be observed in *Charm++*, where the runtime measures the application iterations execution for each task. The system provides the elapsed time of each task to its load balancers which are expected to use this data when employing workload aware policies. Only through the system's exposure of this functionality, and the scheduler's use of it, the system may enable adaptive scheduling solutions to its applications. Therefore, in order to account for these contextual features, the following specifications are designed for an *ARTful* context implementation:

1. Every context-sensitive code must be implemented on scheduling context components;
2. Each scheduling context must adhere to one or more requirements defined by a global scheduler;
3. Any context that implements a global scheduler requirements must be attachable to it;
4. Any context that does not implement a global scheduler requirements must generate errors if attached to it;
5. A scheduling context must logically represent a single set of compatible features in an environment;
6. The context implementation entities must not be private to the application code;

The first specification states that the scheduling context is the *ARTful* component responsible form encapsulating the global scheduler transformations to a given runtime system. The second specification reinforces the role of this abstraction as a functionality provider for global schedulers. Without following an interface defined by a global scheduler, a context implementation has no purpose. The third and fourth specifications are meant to guarantee the correct software relationship between scheduler and context implementations. Also, the fourth specification states that an error must be issued if an incorrect composition is made as to avoid implementations that fail silently. The fifth specification is a statement to avoid the generalization of contexts, which can increase the code complexity as different configurations of a runtime system are implemented in a single instance. Finally, the sixth specification guarantees that users can access the context instances in the application code. This is a mandatory characteristic to allow schedulers to access data from different sources without forcing runtime systems to expose an extensible user API to inform data that is not managed by the system.

### 4.1.4 Runtime System Adapter Abstraction

The *ARTful* runtime system adapter is an entity responsible for connecting the other abstractions to a runtime system. Regardless of the support for extending the set of policies in a system, there are no existing HPC runtimes without an integrated scheduling module. This abstraction is necessary in this scenario as portable global scheduler

implementations must be adaptable to different system APIs. An implementation of this concept is equivalent to a native global scheduler that, instead of making a schedule decision, forwards its responsibility to a decoupled *ARTful* global scheduler. Additionally, the adapter has the role of holding references to the runtime system data structures so that these are available outside of the runtime system scope (*ARTful* scheduling contexts).

An *ARTful* system adapter has no interface specifications as each runtime portrays an unique strategy to express its global schedulers. Nonetheless, runtime adapters must adhere to its system's routine while portraying the following behavior for an *ARTful* scheduling solution:

1. Expand the runtime system global scheduling options to add *ARTful* schedulers;
2. Initialize references, if necessary, in *ARTful* scheduling contexts to data structures within the runtime system;
3. Forward the schedule decision to an *ARTful* global scheduler when the system selects such a schedule;
4. Implement the global schedule calculated by the *ARTful* entities;

In a system where there is a *framework* for implementing user-defined schedulers, the *ARTful* system adapter can be developed as a user-defined scheduler. Other systems, like *OpenMP*, must rely on alterations in the runtime library functions for expanding the system's global scheduler pool. However, since the adapter yields control to *ARTful* global schedulers, a single adapter can be used to extend the runtime system with multiple scheduling strategies. This is an advantage to the current development process where, for each new scheduler addition, more code must be added to the runtime system library.

## 4.2 *ARTFUL* ABSTRACTIONS OVERVIEW

The abstractions proposed in the *ARTful* specifications create a different relationship between a global scheduler and the runtime system. Invariably, this allows the creation of decoupled modules for global schedulers that, due to compatibility reasons, must communicate to the current native global schedulers.

The anatomy of an *ARTful* scheduling solution and its relationship to the runtime system is presented in Figure 6. The runtime system remains largely unchanged by our approach and only requires the addition of a new **native global scheduler** to forward the schedule decision to the *ARTful runtime adapter*. The adapter then performs two operations, it initializes the scheduling contexts in the library for that runtime system and then calls a global scheduler implementation. The scheduler employs a scheduling policy indirectly using the functionalities provided by the runtime system through the scheduling context. The adapter outputs the schedule to the native global scheduler which then implements the decision in the runtime system.

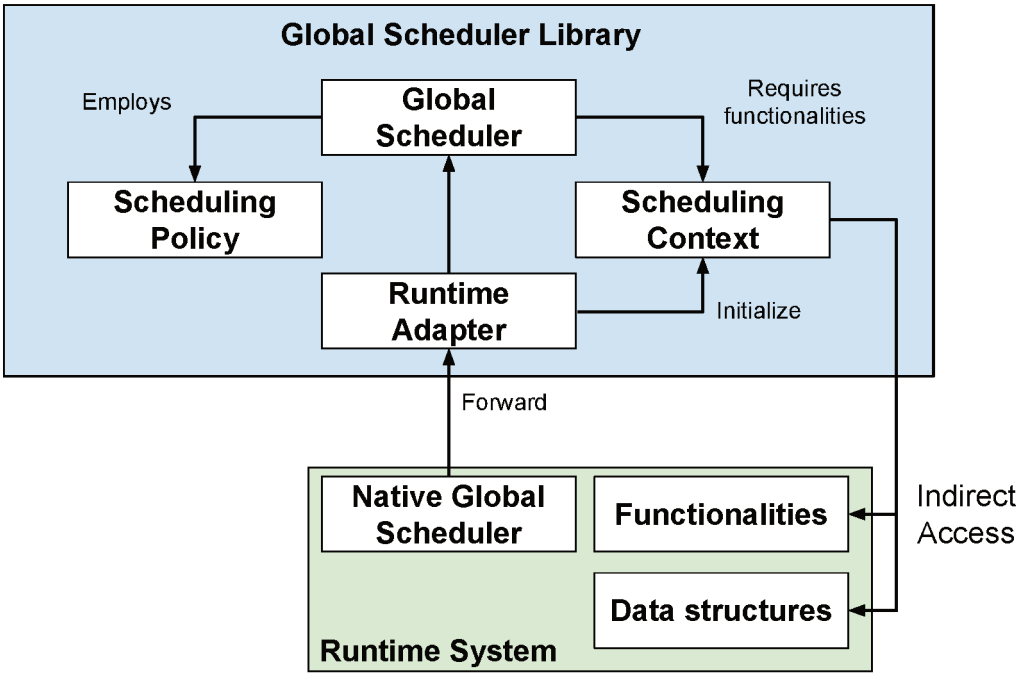


Figure 6 The *ARTful* scheduling solution anatomy.

Different techniques for fulfilling these specifications are possible, enabling different design techniques for composing the library components. Moreover, the *ARTful* specifications and abstractions enable the definition of independent entities that can be assembled in multiple compositions. The composition process is dependant on how the *ARTful* abstractions are implemented. Moreover, the abstractions do not configure a scheduling solution without being attached. We discuss an approach on how to develop and present such abstractions to runtime systems in Chapter 5 by showcasing an implementation of the aforementioned concepts.

# 5 IMPLEMENTATION

The *ARTful* specifications allow developers to create scheduling solutions through a bottom-up process. This implies that global schedulers can be developed starting from the scheduling policy and leaving the adaptation necessary to cope with the runtime system for later, through the development of *scheduling contexts* and *runtime adapters*. In this chapter, we explain our implementation process by presenting the Meta-programmed Oriented Global Scheduling Library (MOGSLib), a collection of *ARTful scheduling abstraction* for the *Charm++* and *OpenMP* systems.

MOGSLib is a library implemented using the object oriented and template meta-programming directives of the *C++* 14 language. While the object oriented aspects of the language allow for the encapsulation of behavior in logical units, the templates are used to generically describe the *ARTful* abstractions connections. This approach is selected as it allows the use of recent algorithm implementations from the standard template library (STL). Additionally, as template classes are generated on demand by the compiler during the name resolution phase, the *ARTful* components can be generically defined during development. The concrete specialized classes are created during compilation and, if incorrect data types are used, errors do not carry on to the execution. The details about how each *ARTful* abstraction is developed in MOGSLib is presented in the remainder of this section.

## 5.1 SCHEDULING POLICIES

We chose the class of centralized and workload-aware policies to experiment in this work. This class of scheduler employs a centralized decision based on the totality of available application and platform data. These policies achieve quasi-linear mapping decision times based on the number of the application's parallel work units. These characteristics and the lack of communication among scheduling entities, present in distributed strategies, reduce the schedule decision time variance which aids our analysis.

In this work, we analyse two policies: (i) the *BinLPT* (PENNA et al., 2017) in *OpenMP* and (ii) the *GreedyLB* in *Charm++*. Both of these policies are implementations of the generic LPT policy discussed in Algorithm 3. *BinLPT* partitions the work units in chunks through a greedy bin packing heuristic. The policy bundles the maximum number of contiguous iterations whose overall load does not exceed the total average load into a single chunk. Later, it schedules the chunks into processing units using the LPT rule. The *BinLPT* was conceived as a loop scheduling policy for *OpenMP* and implemented in an expanded version of libGOMP customized to portray this policy and functions so that users can register the workload of each iteration. Our version of this policy is developed in *C++* 14 within MOGSLib and the workload can be informed through the context data structure rather than requiring additional API extensions to libGOMP.

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

In MOGSLib, every policy is implemented as *C++* static functions encapsulated within template structures. The template parameters are used to abstract data-types during development, allowing the policy to be flexible in regards to the definitions of the data-types that represent the *workload* and the task/work unit *identifiers*. In *Charm++*, when using the observed data from runtime, the *workload* represents the elapsed processing time of the task in the previous iteration in seconds. As such, in *Charm++*, workload is defined by a system-specific type, *LBRealType*, that defaults to *double* in *C++* but can be tweaked by the user during *Charm++* compilation. In libGOMP, the *workload* data represents an arbitrary unit as the data is informed by the user due to the lack of runtime instrumentation. As workloads can not be negative, we configured the default workload data-type in libGOMP to *unsigned integer*. By defining the *workload* as a generic numeric data-type, the policy can be implemented once and used for both systems and can even work on user custom numeric data types.

```
1   template<typename Id, typename WorkLoad>
2   struct LPT {
3   public:
4       using Out = vector<Id>;
5       static void map(Out &map, vector<Workload> tasks, Id npus) {
6           // calculate the global schedule
7           // Output schedule in the map variable
8       }
9   };
```

Algorithm 4 – The LPT template structure interface in MOGSLib.

Algorithm 4 displays the template structure implementation of the generic LPT policy in MOGSLib. In MOGSLib, every policy structure must declare its only public procedure as a static function called `map` (lines 5–8). The first parameter for this procedure is actually the return value, the calculated *global schedule*. This design decision is taken so that policies do not allocate memory for their output and, instead, receive an allocated memory to write their output. The remainder arguments are requirements related to the *LPT* policy: (i) a collection of workload values for the tasks and (ii) the processing units count. This implementation design allows for unitary tests to be performed in the policy at early stages of development while abstracting details in the scheduling context and the global scheduler overall behavior.

## 5.2 GLOBAL SCHEDULERS

The next abstraction to be developed following the *ARTful* scheduling model is the *global scheduler*. In MOGSLib, *global scheduler components* are implemented as *C++* template classes. A class representation for *global schedulers* is employed in both *Charm++* and *StarPU* load balancers due to its versatility to encapsulate data. As such, *global schedulers* instances in the *ARTful* are objects that may register and make decisions based on its internal state. Moreover, the template parameter is exclusive to the MOGSLib implementation and is used to both abstract and bind the class to a data-type that represents the *scheduling context*. In our scope, the objective of the global scheduler class is to situate the generically defined scheduling policies into *OpenMP loop schedulers* and *Charm++ load balancers*. Therefore, the **workload** of each task and the **processing unit count** of the platform must be informed by the *scheduling context*.

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

Algorithm 5 – The implementation of a MOGSLib scheduler that employs the *LPT* policy.

Algorithm 5 showcases the implementation of an *ARTful* global scheduler abstraction that employs the previously described LPT policy. In lines 1 to 2, the template class is declared with a dependency on a data-type to represent the *scheduling context*, `Ctx`. This generic context is used in lines 3 and 4 to derive the concrete data-types for the work/processing units identifiers, `Id`, and the task's workload data-type, `Load`. In MOGSLib, the public procedure to decide a schedule is the `map` function (lines 9–17) which always take one reference to an instance of its *scheduling context* data-type. Despite the data-type being generically defined, the syntax to call the `map` function is the same for any scheduler class. The `map` function may manipulate the context instance to query for functionalities that must be present in its implementation. This is observed in lines 10 and 11, where the *scheduling context* is queried to perform one function to obtain the task's workload and the processing unit count respectively. Moreover, the global schedule

is forwarded to the *ARTful* scheduling policy (lines 13–15) and the output is passed on to the *scheduling context* in line 17 (the practical output of the function).

In a broad sense, any data-type can be passed as a template parameter for a global scheduler implementation in MOGSLib. However, the compilation will only succeed as long as the data-type has a definition for all names required within the global scheduler class. This creates a binding between the two classes that is expressed through a generic interface which exposes the scheduler's requirements to the context. More about the design of the *scheduling context* implementation is discussed in Section 5.3.

## 5.3 SCHEDULING CONTEXTS

In this work, we developed context structures for the *Charm++* and libGOMP systems in order to implant the aforementioned policies into both systems. The *GreedyLB* policy requires the following input data: (i) amount of processing units in the platform; (ii) workload of each work unit in the application. These are the bare minimum requirements of workload-aware strategies and a context structure interface, developed in MOGSLib, can be expressed as in lines 1 to 7 in Algorithm 6. The *BinLPT* also depends on the same aforementioned input data with the additional information of the maximum number of chunks to be created. A compliant interface for the *BinLPT* context can be defined by extending the default *WorkloadCtx* as depicted in lines 9 to 12 in Algorithm 6.

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

Algorithm 6 – The context structure interfaces required by *GreedyLB* and *BinLPT* respectively.

The scheduling policy characteristics do not account for all the features present in the scheduling solution taxonomy. *Scheduling context* implementations must provide implementations for the *global scheduler* requirements that not only comply with the policy's semantics but also to the runtime system's scheduling objectives. *Charm++* applications may contain unmigratable tasks (rigid jobs) and the platform may have unavailable processing units (cannot be marked as a destination for a migration). The system's load balancers must filter the related input data to ignore those elements in the platform to avoid an illegal distribution of work units. Additionally, the workload of processors in the last application iteration is calculated dynamically by registering the time it took to process a task in the processor. This measurements are the reason behind

*Charm++* adaptive scheduling solutions, meaning that workload-aware strategies in this system should employ those metrics as to conform to *Charm++*'s objectives.

A context structure for centralized workload-aware *Charm++* global schedulers, *CharmCentralizedWL*, is depicted in Algorithm 7. The context uses the system Load Balancer Database (LBDB) to query the work/processing units workloads. The database in *Charm++* is a reference to a system-specific data structure that can be accessed in MOGSLib through an adapter to the runtime system with the syntax depicted in line 2. Moreover, the context is responsible for filtering the unavailable processors (lines 8–13), unmigratable chares (lines 15–19) and calculate the workload of the tasks (lines 21–31) using the *Charm++* semantics (manipulation of the LBDB structure). These functionalities are implicitly inserted within the implementation of the workload-aware policy requirements expressed as the `npus` (lines 34–37) and `workloads` (lines 39–43) functions. This context in *Charm++* improves the software quality of the workload-aware global schedulers as common segments of code are no longer replicated in different policies and is explicitly manifested as a scheduling taxonomy feature of *Charm++*.

In the original libGOMP, the support for defining the workload of each iteration of the parallel loop is not present. The authors of *BinLPT* extended the *OpenMP* specifications to portray a new API call, `omp_set_workload` so that the workload could be informed by the user to the loop schedulers in the runtime system. We recreated this feature in MOGSLib within the *OpenMP* context structure depicted in Algorithm 8. In the `LibGOMPWorkload` context structure, the amount of processing units in the environment is determined by the *OpenMP* API function `omp_max_threads` (lines 4–6). The amount of chunks must be gathered from the runtime system's internal structure and must be initialized by the system's adapter (lines 12–14) defined by exposing a public workload array variable which the user can manipulate in the application code. Finally, the workload of each iteration can be informed by the user through the context by manipulating the *workloads* variable (line 2). This approach achieves the same goal of enabling workload-aware scheduling policies in *OpenMP* but does not require additional changes nor extensions to the libGOMP interface. Indeed, other functionalities not present in the original *OpenMP* specifications can fit into the context structure, allowing for the use of different input data in *OpenMP* loop schedulers such as the architecture memory layout in NUMA-aware strategies or the speed of every processor for architectures with uneven processing power.

In a later work of the *BinLPT* authors (PENNA et al., 2019), their custom version of the libGOMP library was once again extended. The new functionality, called *Multi-loop support*, enabled *BinLPT* to reuse a previously calculated iteration mapping by associating its output to a loop id. This feature required a new addition to the *OpenMP* API in the form of the *omp_loop_register* function to inform the runtime which loop will have its schedule calculated next. In order to support this feature, we accommodated the multi-loop logic in a specialized *OpenMP* context that inherited the capabilities of our

```
1   struct CharmCentralizedWL : public WorkloadCtx<unsigned, LBRealType> {
2       LDStats *lbdb = RTS::Charm::lbdb_ref;
3
4   private:
5       vector<Id> PUs, chares;
6       vector<Load> workloads;
7
8       void filterPUs() {
9           PUs.clear();
10          for(Id i = 0; i < lbdb->nprocs(); ++i)
11              if(lbdb->procs[i].available)
12                  PUs.push_back(i);
13      }
14
15      void filterChares() {
16          for(auto i = 0; i < lbdb->n_objs; ++i)
17              if(lbdb->objData[i].migratable)
18                  chares.push_back(i);
19      }
20
21      void calculateWorkloads() {
22          workloads.resize(chares.size());
23          auto i = 0;
24          for(auto chare : chares) {
25              auto &host_pu = lbdb->from_proc[chare];
26              auto &pe_speed = lbdb->procs[host_pu].pe_speed;
27              auto &wall_time = lbdb->objData[chare].wallTime;
28
29              workloads[i++] = wall_time * pe_speed;
30          }
31      }
32
33  public:
34      Id npus() {
35          filterPUs();
36          return PUs.size();
37      }
38
39      vector<Load> workloads() {
40          filterChares();
41          calculateWorkloads();
42          return workloads;
43      }
44  };
```

Algorithm 7 – The context structure implementation for centralized and workload-aware load balancers in *Charm++*.

aforementioned structure. As a result, any scheduler in MOGSLib may benefit from this feature in libGOMP without the cost of additional changes to the runtime system library. Indeed, global schedulers aware of this functionality must be tweaked to issue function calls to the context data structure from the *work* function in their global scheduler class. These adaptations situates multi-loop as an user-defined feature in the scheduling solution accessible to any policy rather than a runtime trait applied to a single policy.

## 5.4 *MOGSLIB* ASSEMBLING TOOLS

Before we discuss the *ARTful runtime adapters* implementation, it is important to showcase how the individual components are grouped together to build the MOGSLib

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

Algorithm 8 The context structure implementation for workload-aware loop schedulers in *OpenMP*.

library API. The *ARTful* specifications guides the development of individual components that can be composed to form a concrete global scheduler. Indeed, a global scheduler library that follows the *ARTful* specifications is comprised of multiple separate components as depicted in Figure 7. *ARTful* global scheduler implementations are configured by a valid combination of *global scheduler class*, *scheduling context* and *runtime adapter*. In MOGSLib, a combination is valid if a *global scheduler class* is able to compile when receiving a *scheduling context class* as template parameter. Additionally, the *scheduling context class* must be correctly initialized by the *runtime adapter*, which will be discussed in Section 5.5. Therefore, it is possible to create tools that aid the composition of global scheduler implementation through user declarative directives to inform which classes and structures should be aggregated together.

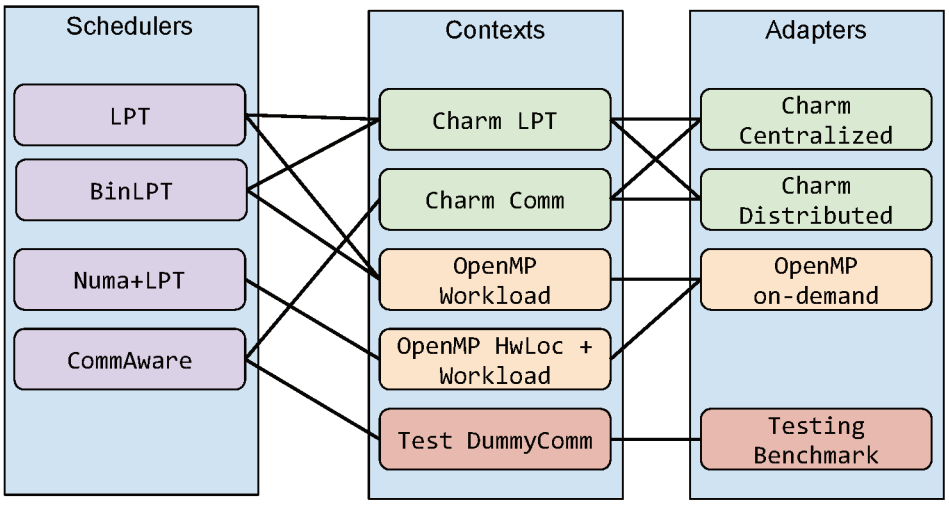


Figure 7 MOGSLib as a collection of *ARTful* compliant implementations.

MOGSLib uses a pre-compilation script to query the user about the structures that must be composed to form a set of context-sensitive global schedulers. The user must inform the path of at least three *C++* header files (template declarations) within

the MOGSLib include directory. The first header file must define a *ARTful global scheduler* class. The second header file must define a *ARTful scheduling context* class to be attached to the scheduler. The last header file must define the *ARTful runtime adapter* class that will be attached to the remainder of the abstractions. This information enable MOGSLib to create the necessary links between the data structures and compose the library API into an automated header file. Finally, MOGSLib can be attached to any *C++* or C source file, consequently allowing MOGSLib to be attached into runtime systems and applications through its inclusion into their source code.

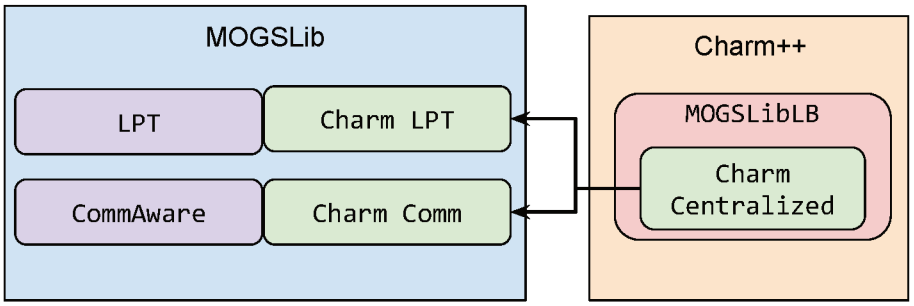


Figure 8 MOGSLib overview after the pre-compilation phase.

MOGSLib is compiled alongside the runtime system exposing only the set of schedulers and abstractions informed in the pre-compilation script. If the compilation succeeds, the library exposes an interface for selecting a scheduler from the set of global schedulers. An overview of the library is depicted in Figure 8. In this image, MOGSLib exposes two figurative strategies, the **LPT** and **CommAware** *scheduling policies*. Those policies are supported by the **CharmLPT** and **CharmComm** contexts respectively. The first context exposing a set of functionalities for querying the workload of tasks in *Charm++* and the second for querying the communication data. Both of these *scheduling solutions* are integrated into *Charm++* through a runtime adapter class **CharmCentralized**. Ultimately, the runtime adapter must be implemented within a native global scheduler, *MOGSLibLB* in the figure, which will forward the decision to the MOGSLib *ARTful* schedulers.

## 5.5 RUNTIME SYSTEM ADAPTERS

In order to implant the *ARTful* global schedulers in the runtime system, it is necessary to create a native global scheduler in the system. This process is different from runtime system to system and MOGSLib enables the reuse of these structures as to make it easier for developers to build upon previous adapters to systems. This effectively reduces the complexity of developing additional scheduling solutions into a system that have been already adapted to work alongside MOGSLib.

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

Algorithm 9 – The *Charm++* centralized adapter *work* function.

The development of centralized load balancers in *Charm++* is supported by the system's load balance *framework*. The development of user-defined load balancers can be achieved by extending the library's *BaseLB* class and implementing the required methods to interface with the system, namely the *work* function. Algorithm 9 showcases the *work* function of the *MOGSLibLB*, the native load balancer that acts as the **charm centralized runtime adapter**. The adapter registers a reference to the system's load balancing database in line 2, so its functionalities can be accessed in the *Charm++ ARTful scheduling contexts*. Later, it calls a function pointer in the library's API, `selected_scheduler`, which the user can define. This function outputs the name of a scheduler within the API that will be called and defaults to the first scheduler passed as parameter in the pre-compilation tool. In line 5, the control is forwarded to the selected *ARTful global scheduler* and its associated *ARTful scheduling context* which can already access the *Charm++* functionalities (line 2). After calculating the global schedule, the adapter implements the decision using the *Charm++* API (lines 10–12) to assign the work units to the processing units.

In libGOMP, the development of loop schedulers rely on modifications within the library's internal functions that handle scheduling. These functions are responsible for exposing the policies and managing the control cycle of the runtime when the scheduling must be performed. The MOGSLib integration to libGOMP is achieved through additions of two functions, `gomp_loop_runtime_start` and `gomp_loop_init`. The first function is executed when the application reaches an *OpenMP* parallel for construct and it detects which policy will be executed. In order to add MOGSLib as a loop scheduler, a new option is added to the `gomp_loop_runtime_start` internal decision mechanism, the **mogslib** policy. The other function, `gomp_loop_init`, is called before the parallel loop starts and it is designed to calculate a static schedule. This function is altered to call the MOGSLib library API to perform the global schedule decision and apply it statically to the threads. Another function is also added to the libGOMP implementation, the `gomp_iter_mogslib_next`. The *gomp_iter_next* functions are called when a thread finishes performing its work pool. The added function in the adapter serves the purpose

of dynamically stealing chunks from other threads as to minimize the dynamic overhead of the static scheduling.

The MOGSLib adapter we developed for libGOMP portrays the same logic as the original *BinLPT* scheduler (PENNA et al., 2017). It has an initial static schedule decision and a dynamic phase for stealing iteration chunks from other overloaded threads. Moreover, our modifications to libGOMP are designed to pass control to MOGSLib schedulers in contrast to calling a policy. The results is an alteration to the runtime library that can support multiple policies with the same static and dynamic steps but different decision algorithms. Finally, MOGSLib is compiled alongside libGOMP and requires an alteration to the build process to include the MOGSLib include folder. Selecting the MOGSLib schedulers in libGOMP is feasible by passing the `runtime` parameter to the `schedule` *OpenMP* directive and setting the *OMP_SCHEDULE* environment variable to **mogslib**.

# 6 EXPERIMENTAL ANALYSIS

We design our experiments to uncover possible performance discrepancies between global schedulers implemented in MOGSLib and their RTS-native counterpart. Global schedulers, regardless of their policy, must display limited mapping decision costs as to not mitigate its performance gains. Our proposed scheduler model must cope with this constraint while, at the same time, decouple the implementation from the RTS and showcase portable implementations. We limit our scope to re-implement the existing policies discussed in chapter 5 and refrain from proposing novel strategies. Ultimately, the expected result of our experiments is to achieve equivalent performance between schedulers. This observation might suggest that global scheduler can be developed outside of runtime systems context and later adjusted into the systems through libraries similar to MOGSLib.

We validate our policy implementations in MOGSLib through unitary tests developed in the googletests (SEN, 2010) API. Therefore, both global schedulers implementations, native and MOGSLib, perform the same logic and the global schedule decisions are identical when the inputs are the same. Whenever applicable, we employed the *C++* Standard Template Library (STL) algorithms and data structures in our implementation, which configures the sole difference in the scheduling policy design between implementations. The experiments and their results are detailed in this chapter in the following sections.

## 6.1 EXECUTION PLATFORM

The execution platform for all experiments is the *Ecotype* cluster from the *Grid'5000* distributed environment. All *Charm++* tests are deployed on four nodes whereas the *OpenMP* tests run over a single node. The compiler used throughout all experiments is g++ version 7.3.0. The *Charm++* RTS version is 6.9.0[1] and the custom libGOMP version is publicly available on GitHub with both the original *BinLPT* and MOGSLib schedulers[2]. Each computing node, in *Ecotype*, has the following specifications[3]:

- **Processor**: Intel Xeon E5-2630L v4@1.80GHz (2 CPU per node);
- **Cores**: $2 \times 10$ (10 cores per CPU);
- **RAM memory**: 128GB DDR3;
- **Network**: Gigabit Ethernet interconnection @10Gbps;
- **Operating System**: *Debian 9*.

[1] available at: http://charm.cs.illinois.edu/software
[2] Available at: https://github.com/alexandrelimassantana/libgomp
[3] Complete Specifications: https://www.grid5000.fr/mediawiki/index.php/

## 6.2 BENCHMARK EXPERIMENTS

We evaluate the scheduler implementations in synthetic benchmarks to uncover performance discrepancies between both implementation approaches. This experimental step aims to measure the **schedule decision time** metric of our approach. In other words, we aim to analyze if MOGSLib incurs in overheads in the decision procedure over the native approaches in the *Charm++* and *OpenMP* systems.

The libGOMP experiments are performed on a custom synthetic benchmark, *SchedCost*, depicted in Algorithm 10. This benchmark performs $N$ parallel loops to simulate an iterative application with a single loop (starting at line 6). Each parallel loop execution performs a dummy application behavior, the addition of two arrays and the result recording in a third (line 8). The *walltime* prior to the loop execution, $PI$, (line 3) and the first instruction within the loop for each thread, $FI_n$, are recorded for each parallel loop (lines 6 and 7). This data is used to calculate the schedule decision cost, $COST$, through the difference between the minimum value among $FI$ and $LI$ (line 10). Finally, the total schedule overhead is achieved through the sum of $COST$ for all $N$ parallel loop repetitions (line 11).

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

Algorithm 10 – Application benchmark to calculate schedule decision cost in *OpenMP*.

*Charm++* is packaged alongside *lb_test*, a synthetic simulation used to test the behavior of load balancers in different application configurations. The simulation performs dummy floating point operations for a randomized amount of time on each of the *Charm++* application chares (tasks). *lb_test* simulates an iterative application where iterations are interleaved with load balancing steps and every chare can be migrated. Among other metrics, *lb_test* outputs the scheduling decision time for each load balancing call which can be used to calculate the total scheduling overhead in the application execution.

We selected the parameters to the *SchedCost* benchmark in order to simulate the characteristics of the LULESH parallel loops (KARLIN; KEASLER; NEELY, 2013). LULESH is an iterative proxy hydrodynamics shock application where each iteration contains multiple parallel loops to calculate various metrics originating from particle in-

teractions. The application is implemented in *OpenMP* and MPI but, in its single-node implementation, it is a balanced application which does not benefit from workload-aware strategies. LULESH has parallel loops with large iteration counts and the simulation spans over multiple steps, which causes workload-aware strategies to cause overheads in the application execution time. As such, we parametrized our benchmark to simulate two LULESH use cases derived from the application's tech as worst case scenarios: (i) 937 parallel loop calls with $30^3$ iterations and (ii) 1477 parallel loop calls with $45^3$ iterations. Each experiment is composed of 30 executions for each configuration carried out in random order.

| **Experiment** | **Min** | **Mean** | **Max** |
|---|---|---|---|
| Small use case | | | |
| Native | 282.91 | 293.22 | 1437.87 |
| MOGSLib | 207.67 | 226.58 | 1404.35 |
| Medium use case | | | |
| Native | 904.35 | 989.59 | 2197.92 |
| MOGSLib | 650.49 | 779.98 | 2312.36 |

Table 2 – *BinLPT* schedule decision cost in *LibGOMP* (microseconds).

The results of the *SchedCost* benchmark is depicted in the table 2. The arithmetic mean, min and max values correspond to the time, in microseconds (us), spent by the scheduler to map the loop iterations to the *OpenMP* threads in the execution platform. Each entry in the table accounts for 28110 and 66465 data points for the *small* and *medium* scenarios, respectively.

Figure 9 showcases the arithmetic mean of the accumulated decision time, in milliseconds (ms), for all loop execution in the simulation. The experiments suggests that MOGSLib performs the mapping faster, showcasing decision overhead reductions by up to 22% when compared to the native implementation. Further testing with different design decision when implementing the scheduler in MOGSLib points out that the use of the *C++* STL algorithms are the reason behind the performance improvements. It is worth noting that these differences are in the scale of microseconds while the LULESH application in this environment and same input take dozens of seconds to execute. Further analysis on these observations are discussed in Chapter 7.

The experiments in *Charm++* are carried out in a similar fashion through the *lb_test* simulation. Different parameter configurations for the application were evaluated and it was observed that most parameters could be ignored when solely analyzing the schedule decision time. The following *lb_test* parameters were fixed for the experiments to showcase a scenario where the *GreedyLB* workload-aware load balancer could be applied with performance gains for different numbers of chares (work units):

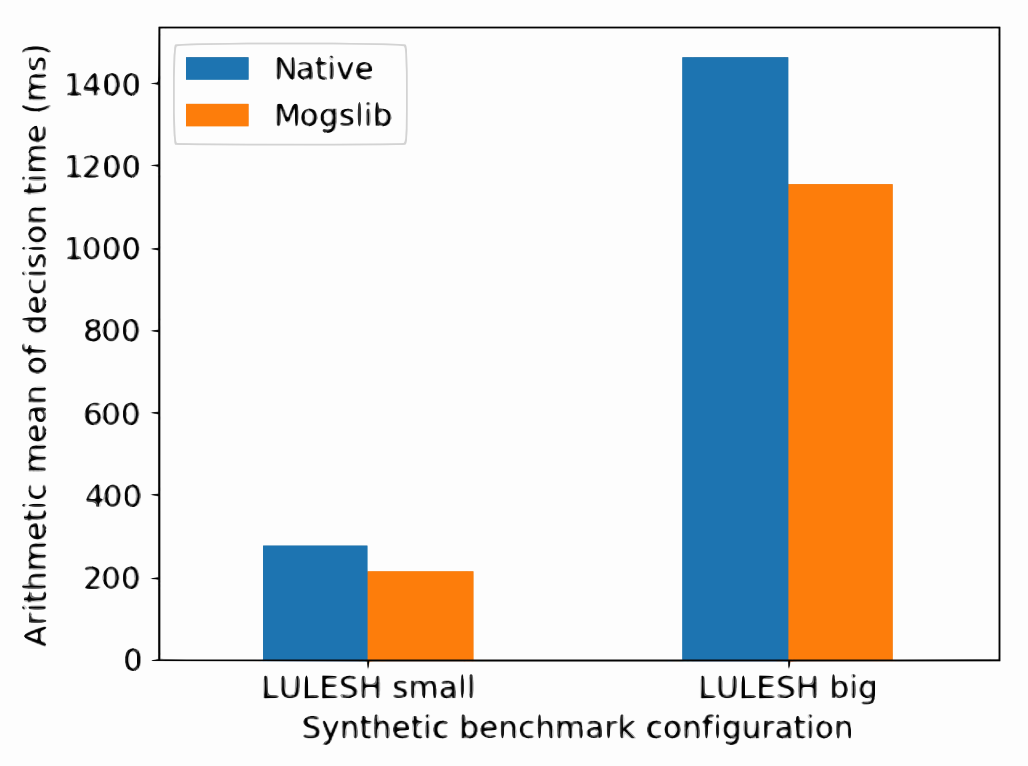


Figure 9 – Total *BinLPT* scheduler overhead in *LibGOMP*.

- min workload: $10\mu s$
- max workload: $3000\mu s$
- topology: Mesh2d
- PU count: 80
- iterations: 151
- *lb period*: 40

The experiments within the *Charm++* context portray various task counts starting at 800 tasks and going up to 3200 in steps of 800 with the additional scenarios of 8000 and 16000 tasks. The resulting scheduling decision cost for this experiment is displayed, in milliseconds, along the y axis in Figure 11 for the different task count configurations along the horizontal axis. Additionally, we showcase the linear regression of these scenarions in Figure 11 to extrapolate the curve in our analyzed range.

Both implementations of the *GreedyLB* scheduler showcased quasi-linear decision time as the number of work units increased. The MOGSLib implementation performed the decision up to 48% faster in the larger experiment, calculating the schedule of 16000 chares in 6.56 milliseconds. Once again, we tested the schedulers to verify the discrepancies in the decision time and observed that, when equally implemented, both versions portray similar decision times. This indicates that the observed discrepancies were also related to the employment of the STL library algorithms.

In both runtime systems, MOGSLib was able to obtain a better performance

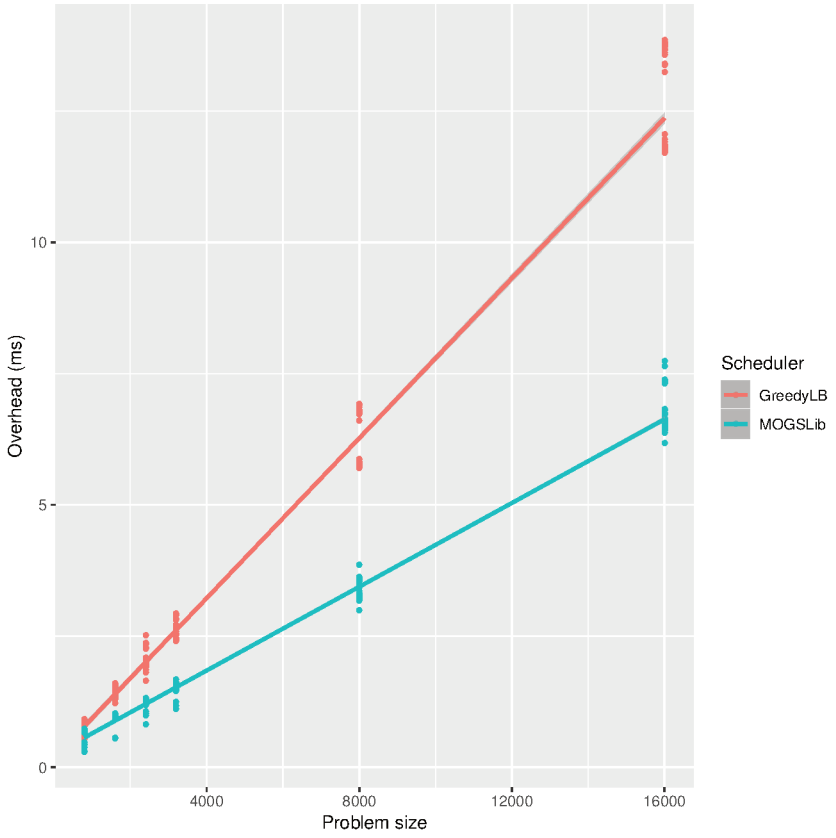


Figure 10 *GreedyLB* schedule decision cost linear regression in *Charm++*.

on schedule decision times. Moreover, the magnitude of these gains represent less than 1% of the total application execution time for these synthetic benchmarks. The gains on *OpenMP* are in the scale of microseconds and in *Charm++* in the magnitude of milliseconds. Both benchmarks, with these parameters take up to 10 seconds in the same testing environment, which suggests that these differences in performance are negligible.

## 6.3 APPLICATION KERNEL EXPERIMENTS

As the next step of our experimental analysis, we apply both global scheduler versions to application kernels to leverage the impact on the application execution time. We chose the class of molecular dynamics (MD) simulations to experiment as this class of problem is present in both *Charm++* and *OpenMP* programming models. The application kernels used in libGOMP and *Charm++* experiments are respectively *LavaMD* (CHE et al., 2009) and *LeanMD* (MEHTA, 2004). The core behavior of both application kernels is to simulate interactions between particles within a cutoff radius. Both simulations are decomposed as equally sized 3D domains and iteratively calculate metrics over all particles following discrete timesteps. *LavaMD*[4] simulates the pressure-induced solidification

[4] *Available at: https://rodinia.cs.virginia.edu/*

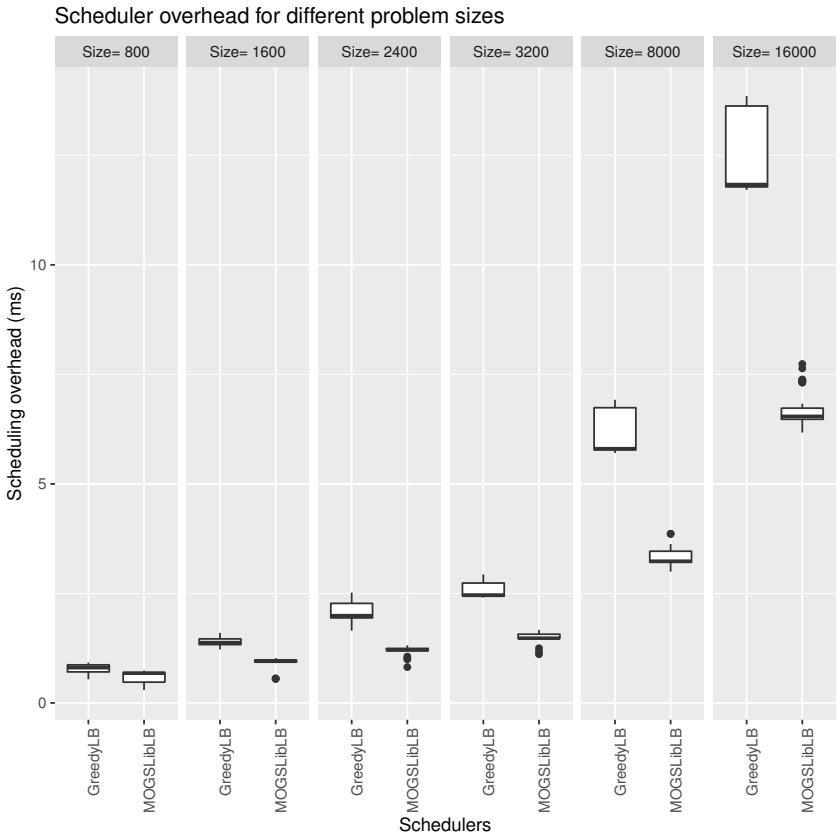


Figure 11 – *GreedyLB* schedule decision cost linear in *Charm++*.

of molten tantalum and uranium atoms. *LeanMD*[5] calculates the force interactions among particles using the Lennard-Jones potential computation. The source of load imbalance within these applications originate from neighbor count differences among particles, resulting in variable calculations for each particle.

The application kernels experiments test if the observations found in the synthetic benchmarks hold true in realistic use cases. We design these experiments to display use cases where the evaluated schedulers can be successfully applied. For the *BinLPT* tests, we recreate a subset of the experiments in (PENNA et al., 2019), a study that evaluates the *BinLPT* policy. As for *GreedyLB*, we configure a small scale *LeanMD* execution in terms of platform and input size as larger scale experiments would detriment the centralized scheduler performance.

The *LavaMD* experiments are configured to decompose the 3D domain into $11^3$ equally sized boxes. Each box contains a random number of particles generated through an exponential distribution with $\gamma = 0.2$. The *BinLPT* scheduler is configured to generate up to 80 task packs in this experiment in order to be aligned with the study we based our cases on. The results of this experiment are portrayed in Figure 12. We analyzed the results using parametric statistical tests and could not reject, with an interval of confidence of 5%, that both observations originate from a normal distributions with the

[5] *Available at: http://charmplusplus.org/miniApps/*

same parameters. This suggests that the scheduler implementations does not affect the overall performance of the application.

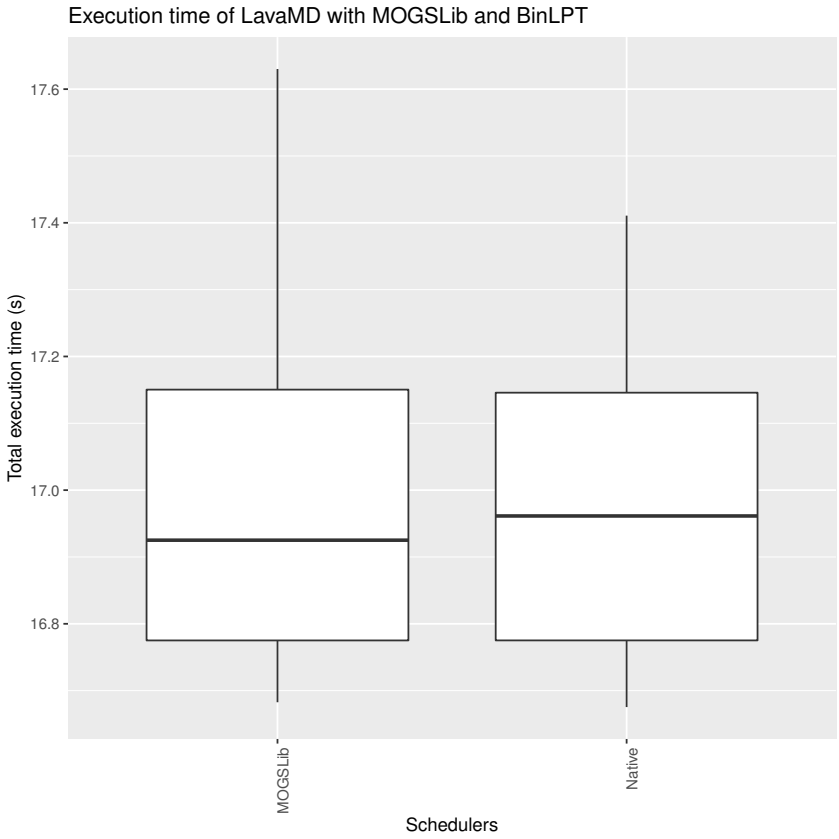


Figure 12 – *LavaMD* execution time when balanced by the *BinLPT* scheduler.

The *LeanMD* application comes packed with default parameter values for testing its behavior. Apart from the dimension of each box, the cells generated from the decomposition of the 3D simulated space, most of the parameters were adjusted. We simulate 300 discrete steps with load balancing starting at the 20th iteration and then performing each 100 steps. Two variations for the boxes count (total simulated space) are used to portray a small and a medium use cases with $8^3$ and $12^3$ boxes, respectively. The comparison between the *GreedyLB* load balancer versions is portrayed in Table 3. The table contains the arithmetic mean of the total application time (app time), the observed standard deviation (Std) and the scheduler overhead (sched time) in seconds.

## 6.4 *OPENMP* MULTI-LOOP SUPPORT

The *LibGOMP* version we used in this work has the multi-loop feature enabled for the *BinLPT* scheduler. The objective of this feature is to reduce the scheduler overhead by conditionally avoiding new schedule decisions when a previous schedule can be used. We recreate this feature outside of the runtime scope in a MOGSLib context structure derived from the *LibGOMP* default context. We examine the performance costs of implementing

| Experiment | App time(s) | Std | Sched time(s) |
|---|---|---|---|
| | $8^3$ boxes | | |
| Native | 32.58 | 0.954 | 0.054 |
| MOGSLib | 31.93 | 1.015 | 0.025 |
| | $12^3$ boxes | | |
| Native | 191.63 | 3.761 | 0.198 |
| MOGSLib | 190.73 | 3.394 | 0.088 |

Table 3 – *LeanMD* execution time.

this feature outside of the library source code through a direct comparison using the *SchedCost* application. We compare our version of the *BinLPT* scheduler implicitly using this feature through the context structure with the *LibGOMP* version. The medium use case parameters discussed in the schedule cost experiments in the *SchedCost* benchmark are re-applied to simulate an iterative application execution. The expected behavior of both schedulers is to calculate the schedule once and reuse it during the remaining parallel loop calls.

| Experiment | Min | Mean | Max |
|---|---|---|---|
| Native | 13.94 | 18.88 | 2216.76 |
| MOGSLib | 14.24 | 19.36 | 1992.87 |

Table 4 – Multi-loop support evaluation (microseconds).

The results for the multi-loop support experiments are showcased in Table 4. The table displays the minimum, maximum and arithmetic mean time cost in microseconds related to the loop scheduler decision making. The maximum value displayed is the first parallel loop call where no previous schedule was available. It can be noticed that the native version is consistently faster than the MOGSLib version. However, the time scale presented is in microseconds and such fluctuations make negligent impact over the total application time which often tends to be several orders of magnitude higher.

# 7 CONCLUSIONS

This work discusses the current techniques for developing scheduling components integrated to parallel solutions. In HPC environments, portability among systems is limited to the selection of application code segments and the fine-tuning of runtime parameter configurations. We propose a discussion about implementation portability on resource management components, specifically global schedulers integrated to runtime systems. We leverage the viability of portability by proposing a set of specifications, *ARTful*, for building and implanting schedulers in different systems. We implement a library, MOGSLib, following the specifications using generic and object oriented programming. As of now, the library serves both as a global scheduler library and development framework focused on a bottom up development process, unitary testing and reusability through composition.

Our experimental analysis focuses on finding performance discrepancies between system-native schedulers and our portable version. For that matter, we select the schedulers that apply workload-aware scheduling policies due to their predictable performance, scalability and use on realistic scenarios. We employ both synthetic benchmarks and application kernels in our analysis to simulate the scheduler use cases and leverage the impact of our implementation. Throughout our benchmark tests, we noticed that our schedulers can obtain up to 48% faster schedule decisions due to the use of standard algorithm implementations from the *C++* STL. Despite the gains being attributed to a source other than our implementation, the *ARTful* models enables the use of the aforementioned library in runtime systems without *C++* support (libGOMP). However, those gains are negligible as the application execution time is several orders of magnitude higher, as observed in the analysis of application kernels.

The experiments suggests that it is possible to design and implement global schedulers focused on portability and the expression of their requirements to the system. We believe that the definition of abstractions and guidelines might aid developers to implement global schedulers more easily on existing systems. The possibility of reusing previously developed abstractions may also reduce the development efforts for these components. However, we could not find metrics to measure the productivity, complexity and portability gains, which limits our ability to discuss the benefits of our approach in these parameters.

We also discuss how MOGSLib can incorporate experimental capabilities into a runtime system. The design focus on mitigating the amount of alterations to the runtime system code as to avoid the creation of multiple unofficial versions of these libraries. Moreover, MOGSLib has proven that certain functionalities can be implemented in user-space by allowing the use of the *C++* language, workload-aware policies and multi-loop support in libGOMP without modifications to the system other than the necessary for supporting MOGSLib.

We envision that in a scenario of programming model transformation and expansion, new methods to enhance the reusability of control and scheduling components must be developed. Our technique was able to support the implementation, testing and integration of scheduling components into runtime systems while also enabling the experimental addition of features into existing industry standards. Overall, MOGSLib is a work in progress that represents an alternative take on developing global schedulers for HPC systems without significant overhead penalties. The library is highly modular and can be expanded in multiple paths, specially in systems where global scheduler development tools are lacking.

As future work, we intend to experiment with other classes of global schedulers. Distributed strategies are particular challenging as they may employ different communication protocols and tools to decide a schedule. These features must be carefully designed into the *ARTful scheduling context* and may be evaluated in multiple systems with different machine topologies. Additionally, there is the class of dynamic schedulers, present in *OpenMP* and *StarPU* does not directly output a global schedule and rather pushes work units to processing elements. These schedulers may require adjustments to the *ARTful specifications* as the *scheduling policy* components can no longer be generalized by their output.

MOGSLib can be extended to expose more adapters to task-based programming models like *StarPU* and support the creation of decentralized global schedulers. More abstractions for incorporating popular libraries and interfaces like HWLoc (BROQUEDIS et al., 2010) may also be studied for the construction of topology-aware policies. We envision that our solution is best suited to scenarios where scheduling solutions are not yet consolidated and portray little to none development support. As such, we also evaluate the possibility of extending MOGSLib and the ARTful model to incorporate the abstractions found in the problem of distributing the workload in heterogeneous environments.

# BIBLIOGRAPHY

ACUN, B.; KALE, L. V. Mitigating processor variation through dynamic load balancing. In: **2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)**. [S.l.]: IEEE, 2016. p. 1073–1076.

ACUN, B. et al. Power, reliability, and performance: One system to rule them all. **Computer**, IEEE, v. 49, n. 10, p. 30–37, 2016.

ASTORGA, D. del R. et al. An adaptive offline implementation selector for heterogeneous parallel platforms. **The International Journal of High Performance Computing Applications**, SAGE Publications Sage UK: London, England, v. 32, n. 6, p. 854–863, 2018.

AUGONNET, C. et al. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. **Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009**, John Wiley & Sons, Ltd., v. 23, p. 187–198, fev. 2011. Disponível em: http://hal.inria.fr/inria-00550877.

AUMAGE, O. et al. Combining both a component model and a task-based model for hpc applications: a feasibility study on gysela. In: IEEE. **Cluster, Cloud and Grid Computing (CCGRID), 2017 17th IEEE/ACM International Symposium on**. [S.l.], 2017. p. 635–644.

BHATELE, A. et al. **Applying graph partitioning methods in measurement-based dynamic load balancing**. [S.l.], 2011.

BLACKFORD, L. S. et al. An updated set of basic linear algebra subprograms (blas). **ACM Transactions on Mathematical Software**, v. 28, n. 2, p. 135–151, 2002.

BROQUEDIS, F. et al. hwloc: A generic framework for managing hardware affinities in hpc applications. In: IEEE. **Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on**. [S.l.], 2010. p. 180–186.

CASAVANT, T. L.; KUHL, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 14, p. 141–154, fev. 1988.

CHE, S. et al. Rodinia: A benchmark suite for heterogeneous computing. In: IEEE. **2009 IEEE international symposium on workload characterization (IISWC)**. [S.l.], 2009. p. 44–54.

CHEVALIER, C.; PELLEGRINI, F. PT-Scotch: A tool for efficient parallel graph ordering. **Parallel computing**, Elsevier, v. 34, n. 6, p. 318–331, 2008.

CIORBA, F. M.; IWAINSKY, C.; BUDER, P. Openmp loop scheduling revisited: Making a case for more schedules. In: SPRINGER. **International Workshop on OpenMP**. [S.l.], 2018. p. 21–36.

COMMITTEE, O. S. et al. **The OpenACC application programming interface version 2.5**. 2015.

DAGUM, L.; MENON, R. Openmp: an industry standard api for shared-memory programming. **IEEE computational science and engineering**, IEEE, v. 5, n. 1, p. 46–55, 1998.

DAIL, H.; CASANOVA, H.; BERMAN, F. A decoupled scheduling approach for the grads program development environment. In: IEEE COMPUTER SOCIETY PRESS. **Proceedings of the 2002 ACM/IEEE conference on Supercomputing**. [S.l.], 2002. p. 1–14.

DURAND, M. et al. An efficient openmp loop scheduler for irregular applications on large-scale numa machines. In: SPRINGER. **International Workshop on OpenMP**. [S.l.], 2013. p. 141–155.

EDWARDS, H. C.; TROTT, C. R.; SUNDERLAND, D. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. **Journal of Parallel and Distributed Computing**, Elsevier, v. 74, n. 12, p. 3202–3216, 2014.

FATTEBERT, J.-L.; RICHARDS, D.; GLOSLI, J. Dynamic load balancing algorithm for molecular dynamics based on voronoi cells domain decompositions. **Computer Physics Communications**, v. 183, n. 12, p. 2608–2615, 2012.

FRASCA, M.; MADDURI, K.; RAGHAVAN, P. NUMA-aware graph mining techniques for performance and energy efficiency. In: **Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis**. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012. (SC '12).

FREITAS, V. et al. A Batch Task Migration Approach for Decentralized Global Rescheduling. In: **SBAC-PAD 2018 - International Symposium on Computer Architecture and High Performance Computing**. Lyon, France: [s.n.], 2018. p. 1–12. Disponível em: https://hal.inria.fr/hal-01860626.

GRAHAM, R. L. et al. Optimization and approximation in deterministic sequencing and scheduling: a survey. **Annals of discrete mathematics**, Elsevier, v. 5, p. 287–326, 1979.

GROSSMAN, M. et al. A pluggable framework for composable hpc scheduling libraries. In: IEEE. **Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International**. [S.l.], 2017. p. 723–732.

HARRELL, S. L. et al. Effective performance portability. In: IEEE. **2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)**. [S.l.], 2018. p. 24–36.

HORNUNG, R. D.; KEASLER, J. A. **The RAJA portability layer: overview and status**. [S.l.], 2014.

JEANNOT, E. et al. Communication and Topology-aware Load Balancing in Charm++ with TreeMatch. In: **IEEE Cluster 2013**. Indianapolis, United States: IEEE, 2013.

JETLEY, P. et al. Massively parallel cosmological simulations with changa. In: IEEE. **2008 IEEE International Symposium on Parallel and Distributed Processing**. [S.l.], 2008. p. 1–12.

KALE, V.; GROPP, W. D. A user-defined schedule for openmp. In: **Proceedings of the 2017 Conference on OpenMP, Stonybrook, New York, USA**. [S.l.: s.n.], 2017. v. 11, p. 2017.

KARLIN, I.; KEASLER, J.; NEELY, R. **LULESH 2.0 Updates and Changes**. [S.l.], 2013. 1-9 p.

LOPES, R. V.; MENASCÉ, D. A taxonomy of job scheduling on distributed computing systems. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 27, n. 12, p. 3412–3428, 2016.

MEHTA, V. **LeanMD: A Charm++ framework for high performance molecular dynamics simulation on large parallel machines**. Tese (Doutorado) — University of Illinois at Urbana-Champaign, 2004.

MEI, C. et al. Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)**. Seattle, USA: IEEE/ACM, 2011. p. 61:1–61:11.

MERRILL, D. et al. Policy-based tuning for performance portability and library co-optimization. In: IEEE. **Innovative Parallel Computing (InPar), 2012**. [S.l.], 2012. p. 1–10.

MOLLISON, M. S.; ANDERSON, J. H. Bringing theory into practice: A userspace library for multicore real-time scheduling. In: IEEE. **Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th**. [S.l.], 2013. p. 283–292.

PENNA, P. et al. Binlpt: A novel workload-aware loop scheduler for irregular parallel loops. In: **XVIII Simpósio em Sistemas Computacionais de Alto Desempenho**. [S.l.: s.n.], 2017.

PENNA, P. H. et al. A comprehensive performance evaluation of the binlpt workload-aware loop scheduler. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, p. e5170, 2019.

PENNYCOOK, S. J.; JARVIS, S. A. Developing performance-portable molecular dynamics kernels in opencl. In: IEEE. **2012 SC Companion: High Performance Computing, Networking Storage and Analysis**. [S.l.], 2012. p. 386–395.

PENNYCOOK, S. J.; SEWALL, J. D.; HAMMOND, J. R. Evaluating the impact of proposed openmp 5.0 features on performance, portability and productivity. In: IEEE. **2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)**. [S.l.], 2018. p. 37–46.

SANTANA, A. et al. Reducing global schedulers complexity through runtime system decoupling. In: IEEE. **2018 Symposium on High Performance Computing Systems (WSCAD)**. [S.l.], 2018. p. 38–44.

SEDOVA, A. et al. High-performance molecular dynamics simulation for biological and materials sciences: Challenges of performance portability. In: IEEE. **2018 IEEE/ACM**

**International Workshop on Performance, Portability and Productivity in HPC (P3HPC)**. [S.l.], 2018. p. 1–13.

SEN, A. A quick introduction to the google c++ testing framework. **IBM DeveloperWorks**, v. 20, p. 1–10, 2010.

STONE, J. E.; GOHARA, D.; SHI, G. Opencl: A parallel programming standard for heterogeneous computing systems. **Computing in science & engineering**, IEEE Computer Society, v. 12, n. 3, p. 66, 2010.

SUPINSKI, B. R. de et al. The ongoing evolution of openmp. **Proceedings of the IEEE**, IEEE, v. 106, n. 11, p. 2004–2019, 2018.

THOMAN, P. et al. A taxonomy of task-based parallel programming technologies for high-performance computing. **Springer Journal of Supercomputing**, v. 74, n. 4, p. 1422–1434, 2018.

WALKER, D. W.; DONGARRA, J. J. Mpi: A standard message passing interface. **Supercomputer**, ASFRA BV, v. 12, p. 56–68, 1996.

WIENKE, S. et al. Development effort estimation in hpc. In: IEEE. **High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for**. [S.l.], 2016. p. 107–118.