Multilevel Scheduling of Computations on Parallel Large-scale Systems

Inauguraldissertation

zur

Erlangung der Würde eines Doktors der Philosophie

vorgelegt der

Philosophisch-Naturwissenschaftlichen Fakultät

der Universität Basel

von Ahmed Hamdy Mohamed Eleliemy

Basel, 2021

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät
auf Antrag von

Prof. Dr. Florina M. Ciorba, First Supervisor
Prof. Dr. Heiko Schuldt, Second Supervisor
Prof. Dr. Wolfgang E. Nagel, External Expert

Basel, den 02.03.2021

Prof. Dr. Marcel Mayor, Dekan

to the soul of my parents

Abstract

Computational scientists are eager to utilize computing resources to execute their applications to advance their understanding of various complex phenomena. This eagerness drives the rapid technological development in high performance computing (HPC). Modern HPC systems exhibit rapid growth in the number of cores per computing node and the number of computing nodes per system. As such, modern HPC systems offer additional levels of hardware parallelism at the *core*, *node*, and *system* levels. Each level requires and employs techniques for appropriate scheduling of the computational work at the respective level. These scheduling techniques work *separately without coordination*, and each technique is designed to achieve specific performance targets. Currently, the *absence of coordination between schedulers at different levels* is an open research problem. In many cases, independent scheduling decisions degrade applications' performance and signify inefficient resources' usage of contemporary HPC systems. To solve this problem, we formulate the following research question: *How can the multilevel parallelism of a modern HPC system be exploited through scheduling to improve the performance of computationally-intensive applications and to enhance the utilization of HPC resources?*

Understanding the relation between the different scheduling levels is crucial for solving the aforementioned research question. However, it is challenged by (1) the absence of methods, models, and tools that allow examining and analyzing the interaction and the mutual impact of these scheduling levels, and (2) the different nature and performance targets of each of these scheduling levels. This doctoral dissertation addresses these challenges in the context of two specific scheduling classes: queuing-based job scheduling at the *batch-level* and dynamic loop self-scheduling (DLS) at the *application-level*. We propose and evaluate a *multilevel scheduling (MLS) prototype* that solves the problem by bridging the schedulers at these scheduling levels. The MLS prototype aims to *decrease applications' execution time* and *increase system utilization*. It employs two novel scheduling approaches that have been introduced by this doctoral dissertation: (1) the *distributed chunk-calculation approach* (DCA) and (2) the *resourceful coordination approach* (RCA) to achieve performance targets.

At the application-level, DCA addresses the scalability challenge associated with existing DLS implementation approaches while maintaining a global scheduling overview that is important to achieve global optimal scheduling decisions.

We apply DCA to several DLS techniques, and we show how it benefits applications' execution time (the first goal of the MLS prototype).

At the batch-level, RCA enables application schedulers to share their allocated but idle computing resources with other applications through a batch system. The significance of RCA is that it leverages and combines the advantages of node sharing and dynamic resource and job management. It offers an efficient resource sharing (of idle resources only) and avoids shrink and expansion operations on the application side. RCA allows batch systems to reassign computing resources once they become free (the second goal of the MLS prototype). By employing DCA and RCA, the MLS prototype answers the research question and shows a creative and useful way of exploiting the multilevel parallelism of modern HPC systems through scheduling.

This doctoral dissertation advances the state-of-the-art by demonstrating the usefulness and the performance potential of coordinated scheduling decisions at different levels. We also designed and implement a set of methods and tools, which we make available for the community to analyze the mutual impact of decision at different levels of scheduling.

Acknowledgements

I see my work as a result of the unconditional support and love of many people, and I am so grateful to them. I appreciate the continuous support of my research advisors: Prof. Dr. Florina M. Ciorba and Prof. Dr. Heiko Schuldt. Prof. Ciorba dedicated time and valuable resources for me to complete this work. She also guided me with her fruitful discussions and comments that shaped my research in its best form. I am also so grateful to Prof. Schuldt, who supported me in many ways more what he thinks.

Many thanks go to my friends: Antonio Maffia, Danilo Guerrera, Ali Mohammed, Jonas Korndörfer, Aurélien Cavelan, and Michal Grabarczyk The morning coffees and the joyful discussions we had together are priceless for me and will never be forgotten. Having such a good company helped me in avoiding stress and depression when things were not going as expected.

Special thanks go to my brother and sister, who supported me from the early days of my childhood and till now. My lovely wife, Omnia, thanks. You encouraged me and believed in me when no one else believed. Finally, my son, Noureldin, my daughter, Laila, since you came to my world and till I leave it, will remain the motivation behind any success I achieve.

This work was partly supported by the Swiss National Science Foundation, which is also thankfully acknowledged.

Contents

List of Figures

List of Tables

1

Introduction

Several domains of scientific research rely on powerful machines, known as high performance computing (HPC) systems. HPC systems refer to those computing platforms that offer more performance than the mainstream computing systems [KT11]. HPC systems enable advanced research in Chemistry [GAB+96], Biology [ST07], Medicine [SVP+10], Engineering [BLP95], and Finance [BLR+12]. Scientists utilize these systems to model, study, and simulate complex phenomena that are cost-prohibitive or not possible experimentally.

For HPC systems, performance is often defined as the number of double-precision floating-point operations per time unit[1] (FLOP/s) that a given HPC system delivers [Don04]. Performance is proportional to the processing frequency and the number of processing units.

Between the 1960s and the beginning of the 2000s, the transistor technology followed Moore's law [Moo+65] closely. Gordon Moore expected that the number of transistors on a chip doubles every year, and later found to be every 18 months. Adding more transistors and scaling up their operating frequency significantly increased system performance and allowed applications to gain performance for free. However, the current fabrication technology of transistors posed limited physical and thermal properties to support higher operating frequencies [Sch97; Kis02]. This fact made increasing parallelism per system the only sustainable way to increase systems' performance. Figure 1.1 shows the number of cores in the top-ranked HPC system in the world since 1996. One can clearly notice that the number of cores increased significantly. Hence, for the top-ranked HPC systems, the total number of cores is in the order of millions[2].

Modern HPC systems are in the form of large-scale *parallel* computing clus-

[1] This metric is used to rank the top 500 HPC systems since 1993 (https://www.top500.org/)
[2] https://www.top500.org/lists/top500/2020/06/

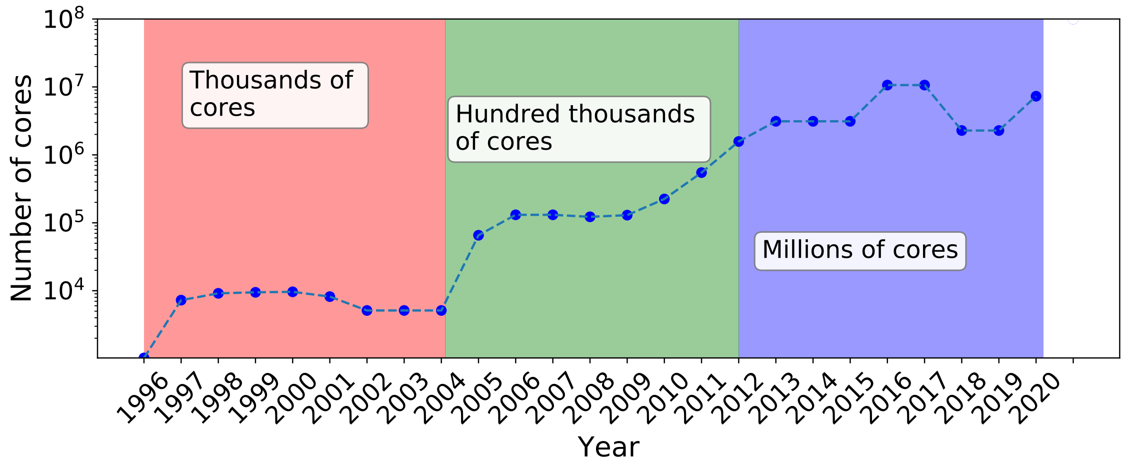


Figure 1.1 Total number of cores in the top-ranked HPC system between 1996 and 2020. The total number of cores per system exponentially increased since 1996.

ters. These parallel clusters aggregate hundreds or thousands of high-end multi-cores and many-core computing nodes [CW10], which are connected with high-speed interconnection networks, such as Infiniband [Pfi01] and Intel Omni-path [BDH+15]. Thus, modern HPC systems offer a high level of hardware parallelism at multiple (core, node, and system) levels. For instance, Figure 1.2 shows the different levels of hardware parallelism in the top-ranked HPC system in June 2020 (Fugaku supercomputer).

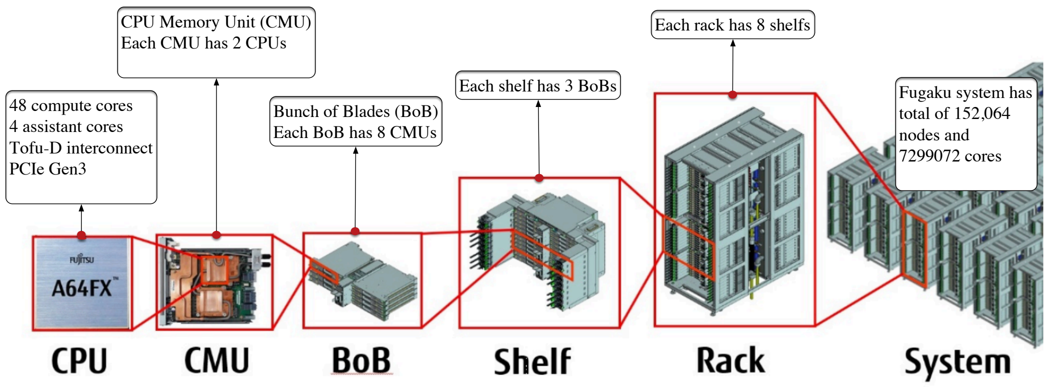


Figure 1.2 Multiple levels of hardware parallelism of the Fugaku supercomputer (adapted and modified from [Don20]). Fugaku is the top supercomputer in the top500 supercomputers June 2020 list with a peak performance of 513.8 PetaFLOP/s.

1.1 Motivation

The efficient utilization of hardware parallelism becomes more critical and challenging than ever. For instance, when a modern (large-scale) HPC system wastes only 1% to 10% of its computing cycles, it wastes energy that could support a small city [SLG+14]. In practice, HPC users aim to improve their applications' execution time without particular regard for increasing system utilization. On the contrary, HPC operators favor increasing the number of executed applications per time unit and increasing system utilization. This difference in the preferences promotes the following operational model. Applications execute on *exclusively-allocated* computing resources for a *specific time*, and applications are assumed to *utilize the allocated resources efficiently*. In many cases, this operational model is inefficient, i.e., applications may not fully utilize their allocated resources. This inefficiency results in increasing application execution time and decreasing system utilization. The work in this doctoral dissertation is motivated by the importance of overcoming such an operational inefficiency.

1.2 Problem Statement and Research Question

Scheduling is the cornerstone of the efficient usage of HPC resources. In general, scheduling refers to computations' assignment to computing resources over a certain period of time [BW91; Ull75]. For HPC systems, scheduling exists in various forms at different levels of hardware and software granularity [BBHB+07], such as scheduling operating system (OS) threads, scheduling application's threads and processes, and scheduling batches of jobs (see Figure 1.3).

Each scheduling technique at a specific level has a different scheduling problem and certain performance targets to achieve. For instance, various jobs *compete* to execute on the available computing resources of a given HPC system [HKK+03]. Batch level scheduling (BLS) techniques manage such competition by prioritizing applications and achieving fairness among HPC users. BLS techniques aim to increase the utilization of system resources and increase the total number of executed applications. BLS techniques do not target minimizing application execution time. Tasks (the finest granularity of work units) within a given application *coordinate* to execute on the allocated resources. Application level scheduling (ALS) techniques support such coordination by assigning ready tasks to free computing resources to minimize the application execution time [BBHB+07]. ALS techniques aim to decrease application execution time. ALS techniques do not target increasing system utilization. Batch and application scheduling techniques work *separately without coordination*.

In 1993, the absence of coordination between job, task, and thread schedulers at the operating system (OS) and application levels was identified and solved for systems of that time (multiprocessor computers with shared memory) [Nag93]. However, for modern HPC systems, non-coordinated scheduling decisions of batch and application schedulers is still relevant and remains an open research problem [BBHB+07; DGGL+18].

Multilevel scheduling (MLS) refers to exchanging scheduling information between scheduling levels, such as batch, application, and OS level. MLS helps in refining scheduling decisions at a certain level based on the available information about the current scheduling workload at other levels. We formulate the following research question to address the problem of coordination absence between schedulers at different scheduling levels: *How can MLS exploit the multiple levels of hardware parallelism of a modern HPC system to enhance scientific applications' performance and increase utilization of HPC resources?*

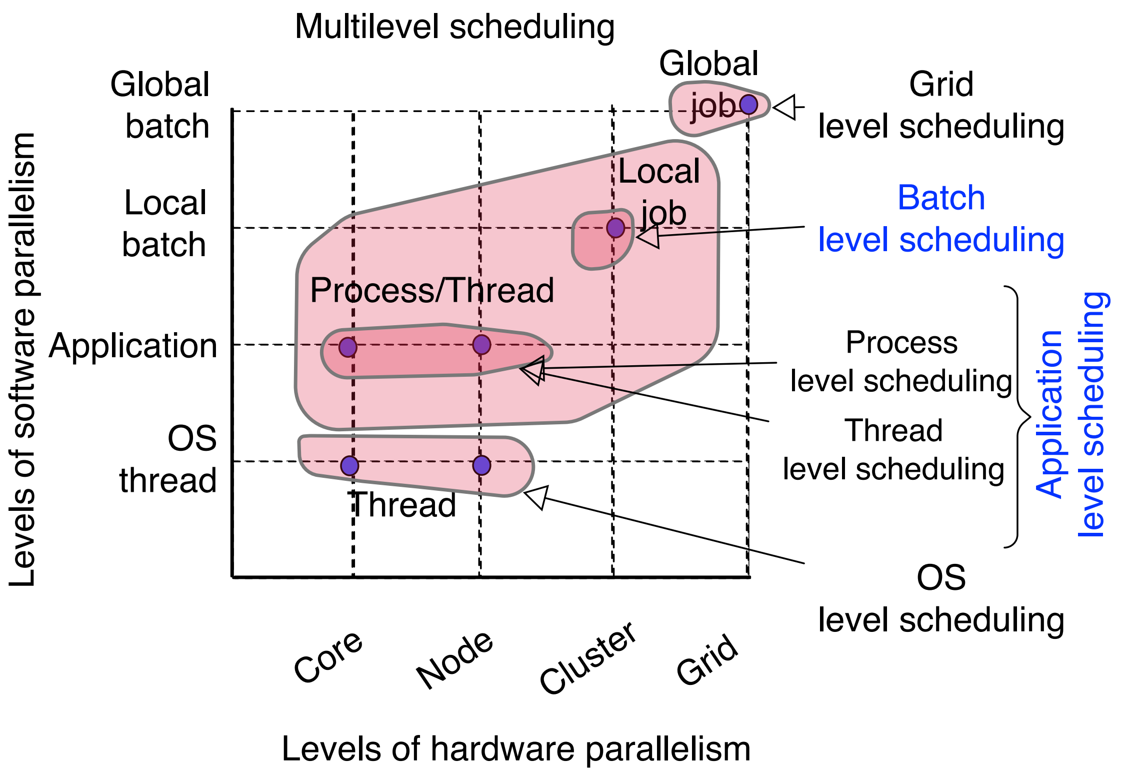


Figure 1.3 Clustering of multilevel scheduling (MLS) into batch level scheduling (BLS) and application level scheduling (ALS)

1.3 Scope of the Dissertation

Two dimensions define the scope in which one can answer the research question above. The first dimension is the applications. HPC applications have different characteristics [VM02] and can be classified into tightly-coupled and loosely-coupled parallel applications [SV09]. In tightly-coupled parallel applications, processes often synchronize with each other. Applications containing routines for solving linear systems are typical examples of tightly-coupled parallel applications [SV09; BCC+97]. On the contrary, in loosely-coupled parallel applications (also known as embarrassingly parallel applications), the synchronization between the processes is negligible or may not exist. Monte-Carlo simulations, image processing, and video rendering are typical examples of loosely-coupled parallel applications.

The second dimension is the systems. HPC systems evolve rapidly, and many HPC architectures existed since the end of the 1980s, such as vector processors, symmetric multiprocessors (SMP), massive parallel processors (MPP), and clusters [Don04; Don03; BG01]. In 2020, computing clusters represent 90% of the top 500 HPC systems[3]. Computing clusters comprise a collection of independent compute nodes. Each node can conduct operations independently, and all nodes are developed and marketed for standalone purposes [DSS+05]. Figure 1.4 shows the typical components of modern HPC clusters.

In this doctoral dissertation, delineating the scope of the studied scheduling techniques depends on the first and second dimensions above. This doctoral dissertation focuses on loosely-coupled applications executing on HPC clusters. Thus, two scheduling categories are relevant: batch level scheduling (BLS) and application level scheduling (ALS), as shown in Figure 1.3. BLS refers to mapping users' applications (jobs) to the available HPC resources. ALS refers to mapping tasks of a particular application to a set of computing resources assigned to execute that application. The answer to the aforementioned research question (see Section 1.2) is found in the context of two specific scheduling classes: queuing-based job scheduling at the *batch level* and dynamic loop self-scheduling (DLS) at the *application level*.

[3] https://www.top500.org/statistics/overtime/

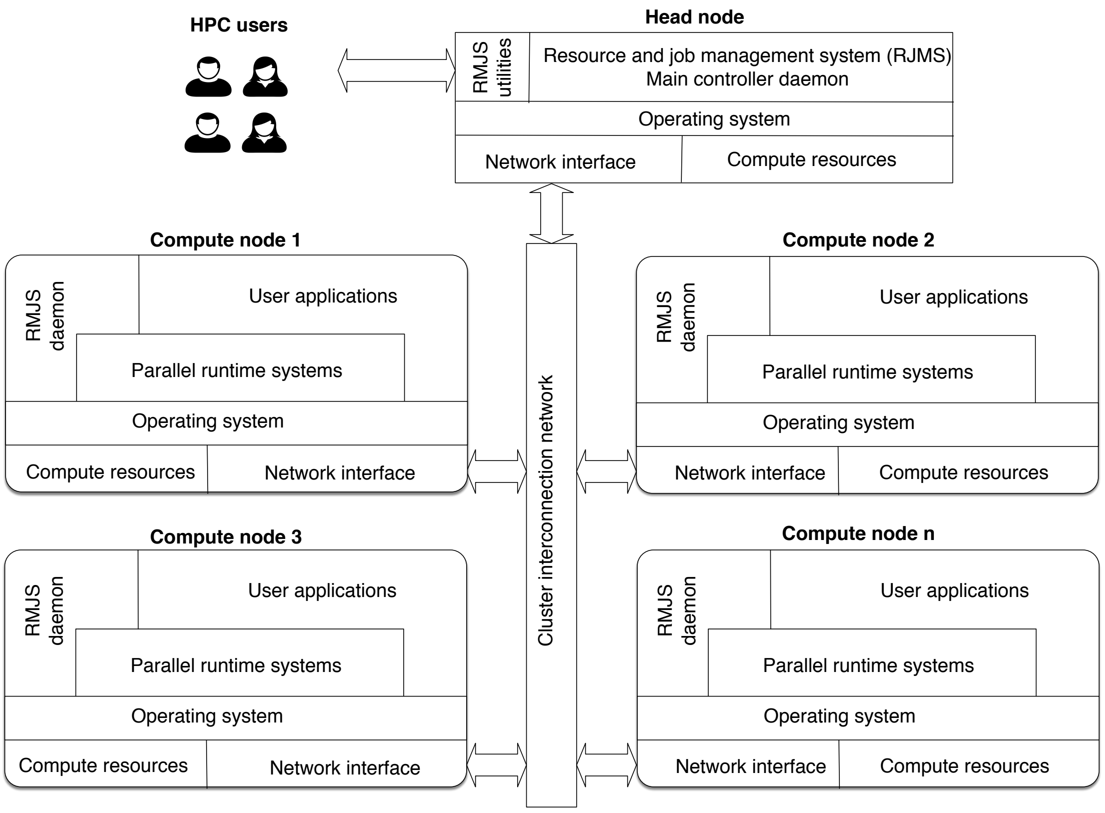


Figure 1.4 System components of modern HPC clusters. The main software components of HPC clusters include (1) Operating systems (usually a Linux based OS), (2) parallel runtime systems (commonly MPI and OpenMP runtime libraries), and (3) the daemons of the resource and job management system (RJMS). Other software components may also exist, such as compilers, profiling, and tracing tools. The main hardware components of HPC clusters include (1) powerful computing nodes (commonly multi- and many-core architectures with or without accelerators) and (2) a powerful interconnection fabrics, such as Infiniband [Pfi01] or Intel Omnipath [BDH+15].

1.4 Research Approach

The work presented in this doctoral dissertation was conducted in four main stages, as shown in Figure 1.5. In the first stage, we aimed to explore the relation between DLS techniques (as ALS) and queuing-based scheduling techniques (as BLS). The absence of methods, models, and tools to examine and analyze the interaction and the mutual impact of BLS and ALS techniques was the main challenge [EMC17b]. We introduced a two-level scheduling simulator that addressed this challenge and allowed us to conclude that **idle times of computing resources towards the end of applications' execution** have a strong negative impact on the performance at both the application and batch levels (see Chap-

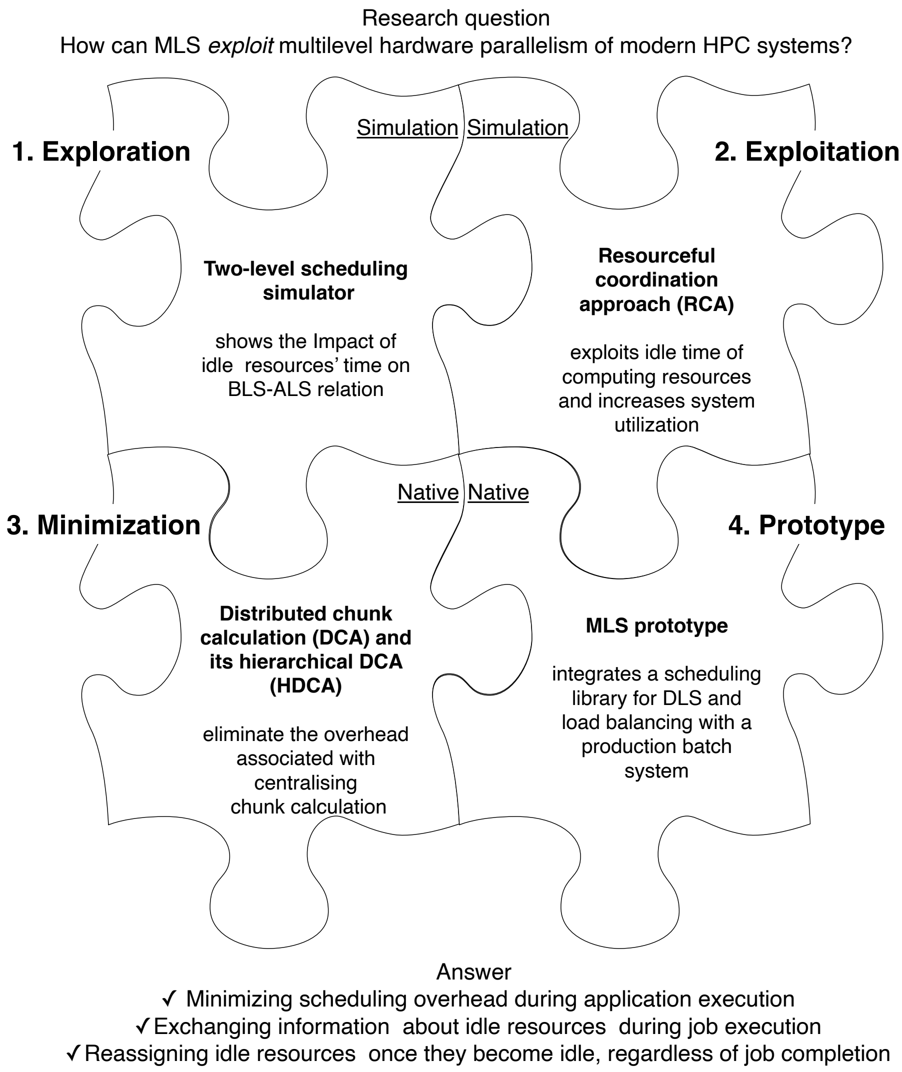

Research question
How can MLS *exploit* multilevel hardware parallelism of modern HPC systems?
1. Exploration
Simulation
Simulation
2. Exploitation
Two-level scheduling simulator
shows the Impact of idle resources' time on BLS-ALS relation
Resourceful coordination approach (RCA)
exploits idle time of computing resources and increases system utilization
Native
Native
3. Minimization
4. Prototype
Distributed chunk calculation (DCA) and its hierarchical DCA (HDCA)
eliminate the overhead associated with centralising chunk calculation
MLS prototype
integrates a scheduling library for DLS and load balancing with a production batch system
Answer
✓ Minimizing scheduling overhead during application execution
✓ Exchanging information about idle resources during job execution
✓ Reassigning idle resources once they become idle, regardless of job completion

Figure 1.5 The four research stages of the work presented in this doctoral dissertation (Exploration, Minimization, Exploitation, and Prototype). The main outcomes of the four stages and how they contribute to the answer of the research question are shown within the puzzle pieces.

ter 3).

In the second stage, the goal was to minimize the idle times, which have been identified in the first stage. Several DLS techniques were introduced since the late of 1980s to address idle times towards the end of applications' execution [PD97]. Different DLS techniques fit for different application-system pairs. We focused on examining implementation approaches of DLS techniques rather than identifying a specific DLS technique that eliminates the idle times for a given application-system pair.

The main conclusion of the second stage was that typical implementation approaches of DLS techniques introduce additional overhead, which contributes to idle times of computing resources. We introduced a distributed chunk calculation approach (DCA) and its hierarchical version (HDCA) to eliminate the additional overhead. DCA avoids the overhead of centralizing chunk calculation and assignment at a single computing resource (see Chapters 4 and 5).

Achieving a perfectly balanced execution of a given parallel loop is an extremely challenging task [BVD03]. DLS techniques allow PEs to have **nearly** equal finishing times by assigning chunks of independent loop iteration to free processing elements (PEs). However, achieving the exact same finishing time is practically infeasible [MC20].

In the third stage, the goal was to exploit idle time when PEs do not have the same finishing times. We introduced a resourceful coordination approach (RCA) that allows one application to share its idle computing resources with other applications through the batch system. RCA solves the problem discussed in Section 1.2 by enabling coordination between the application and batch schedulers (see Chapter 6). The coordination, in this case, refers to sharing information about idle computing resources (by application schedulers) and decisions of reassigning these computing resources to other pending applications (by the batch scheduler).

In the last stage, we provided a scheduling prototype that combines all our proposed scheduling approaches. For instance, DCA was implemented in an MPI-based scheduling library, called LB4MPI [MEC+20; MC20]. Also, RCA was implemented in a production batch scheduler, called Slurm [YJG03]. Notification messages were sent from LB4MPI to Slurm once a resource becomes idle, and consequently, Slurm was able to reassign that resource to other pending jobs. By combining DCA and RCA, the scheduling prototype presented in Chapter 7 represents a *production* scheduler that employs MLS to exploit modern HPC systems efficiently.

1.4.1 Evaluation Methodology

The work presented in this doctoral dissertation was evaluated via simulation and native experiments. Both evaluation methods are used to assess performance of scheduling techniques. Simulation experiments allow exploration of various scenarios with minimum cost. For instance, executing large workloads on an HPC system requires the full reservation of that system and can take several days to complete. In the exploration stage (see Figure 1.5), we evaluated twelve combinations of four ALS and three BLS techniques. The cost of executing such experiments as native experiments is not affordable, i.e., one experiment takes 13 days (see Chapter 3). Similarly, for the exploitation stage (see Figure 1.5), the proposed RCA at the batch level was evaluated via simulation (see Chapter 6).

The main advantage of native experiments is the realistic and trustworthy results [BFM+06]. Native experiments let scheduling techniques experience all variability of a real execution environment, which can be abstracted, simplified, or ignored in simulation. In the minimization stage (see Figure 1.5), we exploited such an advantage and evaluated the proposed DCA and HDCA via native experiments (see Chapters 4 and 5). We also used native experiments to assess the potential of the MLS prototype (see Chapter 7).

1.5 Contributions

Throughout the work in this doctoral dissertation, the following contributions have been made to solve the research problem discussed in Section 1.2.

1. **Two-level scheduling simulation approach:** A novel simulation approach that bridges two different scheduling simulators by exchanging scheduling information among the bridged scheduling simulators [EMC17b]. The proposed approach is exemplified with a two-level simulator that bridges two well-known simulators: SimGrid [EMC16; MEC+20] for ALS and GridSim [KMR07; KR10] for BLS. The newly introduced two-level scheduling simulator stores simulation events produced by both simulators. It also integrates all simulation events into a single file in the OTF2 [EWG+11] format. This format is compatible with trace visualization tools, such as Vampir [KBD+08].

   The significance of this contribution is: enabling the simulations of HPC workloads at fine (tasks within applications) and coarse (jobs within a workload) scales, i.e., it allowed us to explore the relation between ALS and BLS techniques by examining various combinations of these techniques (see Chapter 3). **The two-level simulation approach contributes to the solution of the MLS problem** by identifying idle times of computing resources as a root-cause of the performance degradation at that batch and application levels. Thus, our research focused on coordinating scheduling decisions between batch and application schedulers to minimize and exploit these idle times.

2. **Distributed chunk calculation approach (DCA):** The proposed DCA ensures that every PE can calculate its chunk independently, i.e., the calculated chunk size at any PE does not rely on any information about the chunk size calculated at other PEs. The proposed DCA requires all DLS techniques to have a *straightforward* chunk calculation formula. A *straightforward chunk calculation formula* requires only constants and input parameters, and it does not require prior information about previously calculated chunk sizes. We provide the mathematical transformation needed to ensure that all the chunk calculation formulas of the selected DLS techniques are *straightforward* formulas (see Chapter 4).

   The significance of this contribution is replacing the common master-worker execution model that is used mainly to implement DLS techniques on *distributed-memory* systems. The proposed DCA overcomes certain well-known limitations of the master-worker model. **The DCA contributes to the solution of the MLS problem** by providing a generic execution model that eliminates the overhead of centralizing chunk calculation and assignment on a single computing resource. Thus, it reduces idle times of computing resources.

3. **Hierarchical distributed chunk calculation approach (HDCA)**: DLS techniques assume a *centralized work queue*. All PEs obtain chunks of iteration to execute from that work queue. Similar to the hierarchical master-worker execution model for DLS [WYL+12], HDCA maintains local work queues for each group of PEs that share the same physical memory address space. The local work queues are always filled with new work from the global central queue. The novelty of the proposed HDCA is that the responsibility of maintaining local work queues is shared among all PEs within the same group. In the hierarchical master-worker execution model, such responsibility is assigned only to specific PEs (local masters).

   The significance of this contribution is enabling efficient and scalable implementations of hierarchical DLS techniques. **The HDCA contributes to the solution of the MLS problem** by eliminating another source of overhead, and consequently, minimizing idle times of computing resources.

4. **Resourceful coordination approach (RCA):** RCA enables the cooperation between the currently independent batch and application level schedulers. RCA enables application schedulers to share their allocated but idle computing resources with other applications through the batch system. RCA

avoids resource shrinking operations and associated performance penalties typical of dynamic resource and job management systems.

The significance of this contribution is that the proposed RCA increases the entire system utilization and decreases the system makespan when the applications suffer from a severe load imbalance. For long-executing HPC applications, the proposed RCA showed that exploiting idle times of computing resources (which are in the order of a few seconds) can significantly improve the entire system utilization. To the best of our knowledge and prior to this work, it was commonly accepted that the short idle times of computing resources can only be exploited by Big Data workloads [MGG+17]. RCA highlights the potential of exploring such idle times for HPC workloads as well (see Chapter 6). **The RCA contributes to the solution of the MLS problem** by providing a mechanism to coordinate scheduling decisions of batch and application schedulers to exploit idle times of computing resources [EC21].

5. **The multilevel scheduling (MLS) prototype:** is a software solution that implements the MLS concepts and addresses the *absence of coordination between schedulers at different levels* by employing:

   a) The proposed DCA to minimize application execution times.

   b) The proposed RCA to increase system utilization.

   The MLS prototype connects the job scheduler of Slurm [YJG03] with the LB4MPI scheduling library [MEC+20; MC20].

The MLS prototype contributes to the solution of the MLS problem by gathering, implementing, and applying all the contributions of this doctoral dissertation in a **production HPC environment**, i.e., the MLS prototype confirms the usefulness of the MLS solution in real HPC production systems.

1.6 Outline of the Thesis

The remainder of this doctoral dissertation is organized as follows. In Chapter 2, the two selected scheduling classes of queuing-based scheduling (at the batch level) and dynamic loop scheduling (at the application level) are introduced. Chapter 2 also focuses on the performance goals for each scheduling class and various performance metrics used in the literature to assess the techniques of both scheduling classes.

Chapter 3 describes the first contribution of this doctoral work, which is the two-level scheduling simulation approach. The need and advantages of bridging two different simulators [MEC+20; KMR07] are discussed. The limited benefit of existing HPC workload traces for the two-level simulation is also discussed. The strategy of using a task variation factor to overcome such a limitation is presented. The chapter ends with a performance evaluation of twelve combinations of four DLS techniques and three queuing-based scheduling techniques.

The distributed chunk calculation approach [EC19a] and its hierarchical version [EC19b] are described in Chapters 4 and 5, respectively. Both chapters start by discussing the limitations of existing DLS implementations that motivate the proposed DCA and HDCA. Both chapters end with a performance evaluation of the proposed approach in different scenarios.

The resourceful coordination approach (RCA) is described in Chapter 6 with details on how it is integrated into the Slurm simulator [SIJ+17]. Chapter 6 also describes how the effective system performance (ESP) benchmark [WOK+00b] is used to assess the proposed RCA in simulation.

In Chapter 7, the MLS prototype is introduced. The detailed modifications and extensions made to LB4MPI and Slurm are presented and discussed. The chapter ends with an evaluation and discussion regarding the performance of the MLS prototype. Chapter 8 presents the conclusion of this thesis and an outlook on future research.

1.7 Publications

Following is a list of the publications that are directly and tightly-connected to the contributions of this doctoral dissertation.

[EC21] A. Eleliemy and F. M. Ciorba. A Resourceful coordination Approach for Multilevel Scheduling. In Proceedings of the International Conference on High Performance Computing & Simulation (HPCS 2021), virtual event, 2021.

[EC20] A. Eleliemy and F. M. Ciorba. A Distributed Chunk Calculation Approach for Self-scheduling of Parallel Applications on Distributed-memory Systems. Journal of Computational Science (JOCS), 2021.

[EC19b] A. Eleliemy and F. M. Ciorba. Hierarchical Dynamic Loop Scheduling on Distributed-Memory Systems Using an MPI+MPI Approach. In Proceedings of the 20th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2019) of the 33rd IEEE International Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW 2019), Rio de Janeiro, Brazil, 2019.

[EC19a] A. Eleliemy and F. M. Ciorba. Dynamic Loop Scheduling Using MPI Passive-Target Remote Memory Access. In Proceedings of the 27th Euromicro International Conference on Parallel, Distributed and Networked-based (PDP 2019), Pavia, Italy, 2019.

[EMC17b] A. Eleliemy, A. Mohammed, and F. M. Ciorba. Exploring the Relation Between Two Levels of Scheduling Using a Novel Simulation Approach. In the proceedings of the 16th International Symposium on Parallel and Distributed Computing (ISPDC 2017), Innsbruck, Austria, 2017.

[EMC17a] A. Eleliemy, A. Mohammed, and F. M. Ciorba. Efficient Generation of Parallel Spin-images Using Dynamic Loop Scheduling. In Proceedings of the 8th International Workshop on Multicore and Multithreaded Architectures and Algorithms (M2A2 2017) in conjunction with the 19th IEEE International Conference for High Performance Computing and Communications (HPCC 2017), Bangkok, Thailand, 2017.

During my doctoral work, I have also contributed to other research efforts. I consider the following publications, which I have co-authored, are indirectly related to my doctoral work. I could make benefit of them to my work in simulation, performance analysis, and scheduling in general. These publications are as follows:

[MEC+20] A. Mohammed, A. Eleliemy, F. M. Ciorba, F. Kasielke, and I. Banicescu. An Approach for Realistically Simulating the Performance of Scientific Applications on High Performance Computing Systems. Journal of Future Generation Computer Systems (FGCS), 111:617–633, 2020.

[MEC+18] A. Mohammed, A. Eleliemy, and F. M. Ciorba. Experimental Verification and Analysis of Dynamic Loop Scheduling in Scientific Applications. In Proceedings of the 17th International Symposium on Parallel and Distributed Computing (ISPDC 2018), Geneva, 2018.

[MEC18] A. Mohammed, A. Eleliemy, and F. M. Ciorba. Performance Reproduction and Prediction of Selected Dynamic Loop Scheduling Experiments. In Pro-

ceedings of the International Conference on High Performance Computing & Simulation (HPCS 2018), Orléans, France, 2018.

[EFM+16] A. Eleliemy, M. Fayze, R. Mehmood, I. Katib, and N. Aljohani Loadbalancing on Parallel Heterogeneous Architectures: Spin-image Algorithm on CPU and MIC. In Proceedings of the 9th Eurosim Congress on Modeling and Simulation (EUROSIM 2016), Oulu, Finland, 2016.

2

Scheduling in HPC Systems

Scheduling can be defined as mapping units of work to computing resources over a specific period of time [BW91; Ull75]. Scheduling exists in various forms at different levels of hardware parallelism of HPC systems (core, node, and system). Hence, each level requires and employs techniques for appropriate scheduling of the computational work at the respective level [BBHB+07].

This chapter focuses on dynamic loop self-scheduling (DLS) at the application level and queuing-based job scheduling at the batch level. The most well-known techniques from each class are presented in this chapter. Moreover, the performance metrics that can be used to assess those scheduling techniques are reviewed.

2.1 Application Level Scheduling (ALS)

An application refers to a computer program that executes on one or multiple computing resources to accomplish a specific job. Computer applications often consist of multiple tasks representing the finest granularity of computations. A task cannot be divided into a finer granularity and cannot execute on multiple computing resources simultaneously. Application level scheduling (ALS) refers to mapping tasks of a particular application to a set of computing resources assigned to execute that application.

The majority of applications that execute on HPC systems are scientific applications that often contain large computationally-intensive parallel loops. These loops represent the prime source of parallelism, and their execution dominates the entire application performance [FTY+90]. Scientific applications, such as computational field simulation on unstructured grids, N-body, and Monte-Carlo simulations, are typical examples in which loop scheduling is crucial for the performance [BVD03; BFH95]. In the context of loop scheduling, a loop iteration is the finest granularity that can be mapped to a computing resource. Hence, a loop iteration can refer to a task.

Loop scheduling aims to minimize loop execution time and balance the loop execution across all PEs, i.e., all PEs should have nearly equal finishing times. Loop scheduling techniques are designed to mitigate all sources of load imbalance by mapping *chunks* of independent loop iterations to different PEs. Loop scheduling techniques can be static or dynamic. The time when scheduling decisions are taken is the crucial difference between both categories. Table 3.1 summarizes all notation that describes the chunk size calculation.

Table 2.1 Notation used to describe the selected loop scheduling techniques

Symbol	Description
N	Total number of loop iterations
P	Total number of processing elements
S	Total number of scheduling steps
B	Total number of scheduling batches
i	Index of current scheduling step, $0 \leq i \leq S-1$
b	Index of currently scheduled batch, $0 \leq b \leq B-1$
h	Scheduling overhead associated with assigning loop iterations
R_i	Remaining loop iterations after i-th scheduling step
S_i	Scheduled loop iterations after i-th scheduling step $S_i + R_i = N$
lp_{start}	Index of currently executed loop iteration, $0 \leq lp_{\text{start}} \leq N-1$
L	A DLS technique, $L \in \{STATIC, FSC, GSS, TAP, TSS, FAC, TFSS, FISS, VISS, AF, RND, PLS\}$
K_0^L	Size of the largest chunk of a scheduling technique L
K_{S-1}^L	Size of the smallest chunk of a scheduling technique L
K_i^L	Chunk size calculated at scheduling step i of a scheduling technique L
p_j	Processing element j, $0 \leq j \leq P-1$
Wp_j	Relative weight of processing element j, $0 \leq j \leq P-1$, $\sum_{j=0}^{P-1} Wp_j = P$
h	Scheduling overhead for assigning a single iteration
σ_{p_i}	Standard deviation of the loop iterations' execution times executed on p_j
μ_{p_i}	Mean of the loop iterations' execution times executed on p_j
T_{p}	Parallel execution time of the entire application
$T_{\text{p}}^{\text{loop}}$	Parallel execution time of the application's parallelized loops

2.1.1 Static Loop Scheduling (SLS)

Static loop scheduling (SLS) takes scheduling decisions before application execution. The chunk sizes and their assignment are known before the execution. Block, cyclic and block-cyclic represent various examples of SLS techniques [LTS+93]. Block [LTS+93], also known as STATIC, is a straightforward technique that divides the loop into P chunks of equal size, as shown in Eq. 2.1. Each chunk is assigned to a corresponding PE, i.e., the i_{th} chunk is assigned to the i_{th} PE.

$$K_i^{STATIC} = \frac{N}{P} \tag{2.1}$$

Cyclic and block-cyclic also assign the same amount of loop iterations to each PE, i.e., each PE gets a total number of iterations that is equal to $\frac{N}{P}$. However, in cyclic, the loop iterations are distributed one by one in a cyclic fashion. In contrast, block-cyclic scheduling distributes blocks of loop iterations in a cyclic fashion. Because SLS techniques take scheduling decisions before application execution, they incur the minimum scheduling overhead, and they have less capability to balance the execution of loops in highly irregular execution environments.

2.1.2 Dynamic Loop Self-scheduling (DLS)

Dynamic loop scheduling-self (DLS) techniques take scheduling decisions during application execution. Compared to SLS, DLS techniques incur significant scheduling overhead, but they are more capable of balancing the loop execution than SLS techniques, especially in highly irregular execution environments. DLS techniques have been used in different applications, such as N-body simulation [BFH95], computational fluid dynamics [BVD03], solar map generation [BWA16], spin-image generation [EMC17a], and heat conduction [BV02]. Furthermore, DLS techniques can be divided into non-adaptive and adaptive techniques.

2.1.2.1 Non-adaptive DLS

The *non-adaptive* techniques utilize the information that is obtained before the application execution. The non-adaptive techniques include self-scheduling (SS) [PPC86], fixed size self-scheduling (FSC) [KW85], guided self-scheduling (GSS) [PK87], taper (TAP) [Luc92], trapezoid self-scheduling (TSS) [TN93], factoring (FAC) [FHSF92], weighted factoring (WF) [FHSU+96] trapezoid factoring self-scheduling (TFSS) [CAB+01], fixed increase self-scheduling (FISS) [PD97], variable increase self-scheduling (VISS) [PD97], random (RND) [CIB18], and performance-based loop scheduling (PLS) [SYT07].

SS [PPC86] is a dynamic self-scheduling technique where the chunk size is always one iteration, as shown in Eq. 2.2. SS has the highest scheduling overhead because it has the maximum number of chunks, i.e., the total number of chunks is N. However, SS can achieve a highly load-balanced execution in highly irregular execution environments.

$$K_i^{SS} = 1 \tag{2.2}$$

As a middle point between STATIC and SS, FSC assumes an optimal chunk size that achieves a balanced execution of loop iterations with the smallest overhead. To calculate such an optimal chunk size, FSC considers the variability in iterations' execution time and the scheduling overhead of assigning loop iterations to be known before applications' execution. Eq. 2.3 shows how FSC calculates the optimal chunk size.

$$K_i^{FSC} = \frac{\sqrt{2} \cdot N \cdot h}{\sigma \cdot P \cdot \sqrt{\log P}} \tag{2.3}$$

GSS [PK87] is also a compromise between the highest load balancing that can be achieved using SS and the lowest scheduling overhead incurred by STATIC. Unlike FSC, GSS assigns decreasing chunk sizes to balance loop executions among all PEs. At every scheduling step, GSS assigns a chunk that is equal to the number of remaining loop iterations divided by the total number of PEs, as shown in Eq. 2.4.

$$\begin{aligned} K_i^{GSS} &= \frac{R_i}{P}, \text{ where} \\ R_i &= N - \sum_{j=0}^{i-1} k_j^{GSS} \end{aligned} \tag{2.4}$$

TAP [Luc92] is based on a probabilistic analysis that represents a general case of GSS. It considers the average of loop iterations' execution time μ and the standard deviation σ to achieve a higher load balance than GSS. Eq. 2.5 shows how TAP tunes the GSS chunk size based on μ and σ.

$$\begin{aligned} K_i^{TAP} &= K_i^{GSS} + \frac{v_\alpha^2}{2} - v_\alpha \cdot \sqrt{2 \cdot K_i^{GSS} + \frac{v_\alpha^2}{4}}, \text{ where} \\ v_\alpha &= \frac{\alpha \cdot \sigma}{\mu} \end{aligned} \tag{2.5}$$

TSS [TN93] assigns decreasing chunk sizes similar to GSS. However, TSS uses a linear function to decrement chunk sizes. This linearity results in low scheduling overhead in each scheduling step compared to GSS. Eq. 2.6 shows the linear function of TSS.

$$\begin{aligned} K_i^{TSS} &= K_{i-1}^{TSS} - \left\lfloor \frac{K_0^{TSS} - K_{S-1}^{TSS}}{S-1} \right\rfloor, \text{ where} \\ S &= \left\lceil \frac{2 \cdot N}{K_0^{TSS} + K_{S-1}^{TSS}} \right\rceil \\ K_0^{TSS} &= \left\lceil \frac{N}{2 \cdot P} \right\rceil, K_{S-1}^{TSS} = 1 \end{aligned} \tag{2.6}$$

FAC [FHSF92] schedules the loop iterations in batches of equally-sized chunks. FAC evolved from comprehensive probabilistic analyses, and it assumes prior knowledge about μ and σ. Another practical implementation of FAC denoted, FAC2, assigns half of the remaining loop iterations for every batch, as shown in Eq. 2.7. The initial chunk size of FAC2 is half of the initial chunk size of GSS. If more time-consuming loop iterations are at the beginning of the loop, FAC2 may better balance their execution than GSS.

$$\begin{aligned} K_i^{FAC2} &= \begin{cases} \left\lceil \frac{R_i}{2 \cdot P} \right\rceil, \text{if } i \mod P = 0 \\ K_{i-1}^{FAC2}, \text{otherwise.} \end{cases}, \text{ where} \\ R_i &= N - \sum_{j=0}^{i-1} k_j^{FAC2} \end{aligned} \tag{2.7}$$

WF [FHSU+96] is based on FAC. However, each PE executes variably-sized chunks of a given batch according to its relative weights. The processor weights, W_{pj} , are determined prior to applications' execution and do not change during the execution. WF2 is the practical implementation of WF that is based on FAC2, as shown in Eq 2.8.

$$K_i^{WF2} = K_i^{FAC2} \cdot W_{pj} \tag{2.8}$$

TFSS [CAB+01] combines certain characteristics of TSS [TN93] and FAC [FHSF92]. Similar to FAC, TFSS schedules loop iterations in batches of equally-sized chunks. However, it does not follow the analysis of FAC, i.e., every batch is not half of the remaining number of iterations. Batches in TFSS decrease linearly, similar to chunk sizes in TSS. As shown in Eq. 2.9, TFSS calculates the chunk size as the sum of the next P chunks that would have been computed by the TSS divided by P.

$$K_i^{TFSS} = \begin{cases} \frac{\sum_{j=i}^{i+P-1} K_j^{TSS}}{P} & \text{if } i \mod P = 0 \\ K_{i-1}^{TFSS}, & \text{otherwise.} \end{cases} \tag{2.9}$$

GSS [PK87], TAP [Luc92], TSS [TN93], FAC [FHSF92], and TFSS[CAB+01] employ a decreasing chunk size pattern. This pattern introduces additional scheduling overhead due to the small chunk sizes towards the end of the loop execution. On distributed-memory systems, the additional scheduling overhead is more substantial than on shared-memory systems. FISS [PD97] is the first scheduling technique devised explicitly for distributed-memory systems. FISS follows an increasing chunk size pattern calculated as in Eq. 2.10. FISS depends on an initial value B defined by the user (suggested to be equal to the FAC's total number of batches).

$$\begin{aligned} K_i^{FISS} &= K_{i-1}^{FISS} + \lceil \frac{2 \cdot N \cdot (1 - \frac{B}{2+B})}{P \cdot B \cdot (B-1)} \rceil \text{, where} \\ K_0^{FISS} &= \frac{N}{(2+B) \cdot P} \end{aligned} \tag{2.10}$$

VISS [PD97] follows an increasing pattern of chunk sizes. Unlike FISS, VISS relaxes the requirement of defining an initial value B. VISS works similarly to FAC2, but instead of decreasing the chunk size, VISS increments the chunk size by a factor of two per scheduling step. Eq. 2.11 shows the chunk calculation of VISS.

$$\begin{aligned} K_i^{VISS} &= \begin{cases} K_{i-1}^{VISS} + \frac{K_{i-1}^{VISS}}{2} & \text{if } i \mod P = 0 \\ K_{i-1}^{VISS}, & \text{otherwise.} \end{cases} \text{, where} \\ K_0^{VISS} &= K_0^{FISS} \end{aligned} \tag{2.11}$$

RND [CIB18] is a DLS technique that utilizes a uniform random distribution to arbitrarily choose a chunk size between specific lower and upper bounds. The lower and the upper bounds were suggested to be $\frac{N}{100 \cdot P}$ and $\frac{N}{2 \cdot P}$, respectively [CIB18]. In the current work, we suggest a lower and an upper bound as 1 and $\frac{N}{P}$, respectively. These bounds make RND have an equal probability of selecting any chunk size between the chunk size of STATIC and the chunk size of SS, which are the two extremes of DLS techniques in terms of scheduling overhead and load balancing. Eq. 2.12 represents the integer range of the RND chunk sizes.

$$K_i^{RND} \in [1, N/P] \tag{2.12}$$

PLS [SYT07] combines the advantages of SLS and DLS. It divides the loop into two parts. The first loop part is scheduled statically. In contrast, the second part is scheduled dynamically using GSS. The static workload ratio (SWR) is used to determine the amount of the iterations to be statically scheduled. SWR is calculated as the ratio between minimum and maximum iteration execution time of five randomly chosen iterations. PLS also uses a performance function (PF) to statically assign parts of the workload to each processing element p_j based on the PE's speed and its current CPU load. In the present work, all PEs are assumed to have the same load during the execution. This assumption is valid given the exclusive access to the HPC infrastructure used in this work. Eq. 2.13 shows the chunk calculation of PLS.

$$\begin{aligned} K_i^{PLS} &= \begin{cases} \frac{N \cdot SWR}{P}, & \text{if } R_i > N - (N \cdot SWR) \\ K_i^{GSS}, & \text{otherwise.} \end{cases} \text{, where} \\ SWR &= \frac{\text{minimum iteration execution time}}{\text{maximum iteration execution time}} \end{aligned} \tag{2.13}$$

2.1.2.2 Adaptive DLS

Adaptive techniques regularly obtain information **during the application execution**, and the scheduling decisions are taken based on that new information. The adaptive techniques incur a significant scheduling overhead compared to non-adaptive techniques and outperform the non-adaptive ones in highly irregular execution environments. One can find two main adaptive DLS techniques in the literature: adaptive weighted factoring (AWF) [BVD03] and adaptive factoring (AF) [Ban00].

AWF is similar to WF [FHSU+96]. i.e., each PE executes variably-sized chunks of a given batch according to its relative weight. However, the weight is updated during execution based on the performance of the processor. AWF is devised for time-stepping applications., i.e., processor weights are only updated at the end of each time-step. Variants of AWF(AWF-B and AWF-C) relaxed this constraint by updating processor weights at every batch and chunk, respectively [CB08]. Additional variants of AWF, such as AWF-E and AWF-D, are similar to AWF-B and AWF-C, respectively. However, AWF-E and AWF-D consider the overhead of scheduling in measuring the relative weights.

AF [Ban00] is an adaptive DLS technique based on FAC. However, in contrast to FAC, AF learns both μ and σ for each computing resource during application execution to ensure full adaptivity to all factors that cause load imbalance. AF adapts chunk size based on the continuous updates of loop iteration execution μ and their standard deviation σ during application execution. Therefore, the pattern of AF's chunk sizes is unpredictable. Figure 2.1 shows examples of calculated chunk size patterns generated by different DLS techniques. Eq. 2.14

shows the chunk calculation of AF.

$$
\begin{aligned}
K_i^{AF} &= \frac{D + 2 \cdot E \cdot R_i - \sqrt{D^2 + 4 \cdot D \cdot E \cdot R_i}}{2\mu_{p_i}}, \text{ where} \\
D &= \sum_{p_i=1}^{P} \frac{\sigma_{p_i}^2}{\mu_{p_i}} \\
E &= \left(\sum_{p_i=1}^{P} \frac{1}{\mu_{p_i}} \right)^{-1}
\end{aligned}
\tag{2.14}
$$

2.1.3 Performance Metrics

For ALS, the primary performance metric is the parallel execution time T_p of the entire application. T_p is defined as the time when the latest PE finishes. This doctoral dissertation focuses on applications with a single computationally-intensive loop that dominates the application's execution. Therefore, we consider the parallel loop execution time T_p^{loop} of the *main loop* of any given application to be the main metric that assesses the application performance. When processors execute the main loop of a given application, they often experience uneven processor finishing times. This case is also known as **load imbalanced** execution of loop iterations. Load imbalance is another primary performance metric for parallel applications.

The load imbalance is often measured by two metrics: (1) the *coefficient of variation (c.o.v)* of PEs' finishing time [FHSF92] and (2) the *percent load imbalance* [DHJ07; CBL08]. The c.o.v. is the ratio between the standard deviation of processor finishing time and the average processor finishing time, as shown in Eq. 2.15.

$$
\text{c.o.v} = \frac{\sigma}{\mu} \tag{2.15}
$$

High values of the **c.o.v** indicate high imbalanced load execution, while values close to zero indicate balanced execution. The percent load imbalance is calculated as shown in Eq 2.16 [DHJ07].

$$
\text{Load imbalance } = \left(1 - \frac{\textbf{mean of processor finishing times}}{\textbf{max of processor finishing times}} \right) * 100 \tag{2.16}
$$

Similar to the *c.o.v* metric, high values of *percent load imbalance* indicate sever imbalanced execution, while values close to zero indicate balanced execution.

A slightly different form of this metric has been reported in the literature. The load imbalance is measured directly as a ratio between the max and the

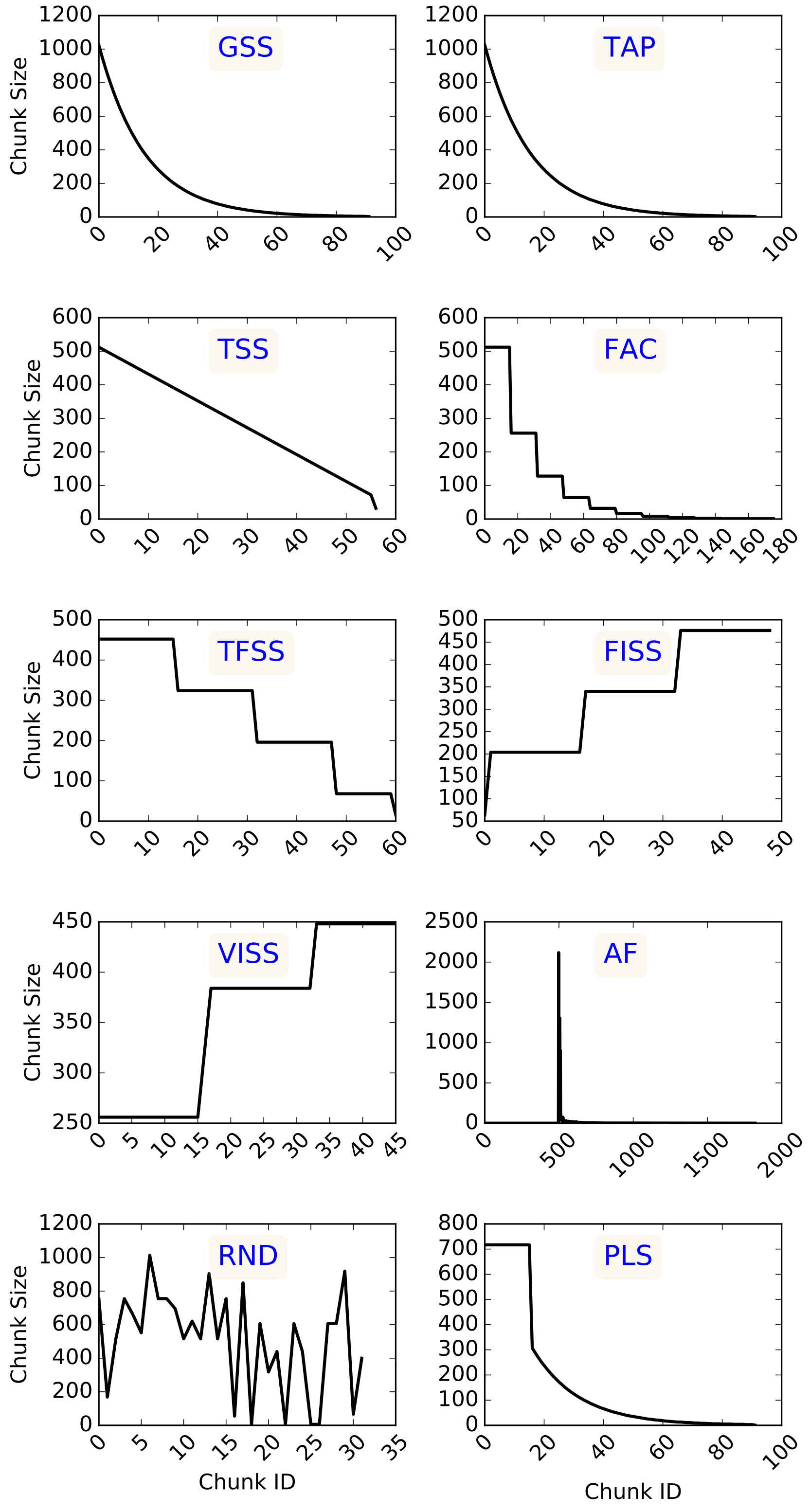


Figure 2.1 Chunk sizes generated by different DLS techniques. The data was obtained from the main loop of Mandelbrot [Man80] with 512*512 loop iterations and executing on 16 nodes (16 cores per node) such that one MPI rank is mapped to each core.

mean of processor finishing times [PGW+17]. In that case, the metric is called (*max*/*mean*), and when the value of *max*/*mean* is close to one, the load execution is balanced.

2.2 Batch Level Scheduling (BLS)

Users of HPC systems execute their applications as batch jobs. A *batch job* represents a *request of specific computing resources* for a *limited time* to execute particular application binaries [FBP15][Rod17, page. 6]. Batch level scheduling (BLS) refers to mapping users' jobs to the available HPC resources. Resource and job management systems (RJMSs), also known as a batch system, are critical components of HPC systems. RJMSs are responsible for *BLS*, job life cycle management, resource management, and job execution [RBA+18]. One may consider RJMSs as operating systems for HPC systems [GH12]. There are two different classifications of RJMSs: (1) static vs. dynamic [FR96; PIR+14] and (2) planning vs. queuing [HKK+03] systems.

2.2.1 Static vs. Dynamic Batch Systems

Static RJMSs are systems that provide static resource allocation to jobs, i.e., the resource allocation cannot be changed once the job starts. In contrast, dynamic RJMS change resource allocation during job execution. The concept of static and dynamic resource allocation is tightly coupled with the four types of batch jobs [FRS+97]: (1) Rigid jobs which are the most common type of job found in HPC systems. A Rigid job is a request for a specific number of computing resources that are necessary to execute the application binaries. (2) Moldable jobs which are similar to rigid jobs. However, RJMSs have the flexibility to change the number of the requested computing resources before the application starts. Once applications start, the batch system cannot change their resource allocation. (3) Malleable jobs which refer to the preferred jobs for any batch system, i.e., the resource allocation of a malleable job can be changed by the batch system at any time. (4) Evolving jobs which refer to jobs that request an additional computing resource from the batch system during their execution. Static RJMSs support the first two types of jobs (rigid and moldable jobs), while dynamic RJMS support the other two types (malleable and evolving jobs). Most batch systems support only static allocation [PIR+14]. A few production batch systems, such as Slurm [YJG03], only provide *certain sort of* support for dynamic allocation. In Slurm, resource expansion is done by allowing a running job to submit a new job with a dependency indicator and merging the allocations. Slurm requires all the resources assigned dynamically for the job to be released together [Pra16, page. 17].

2.2.2 Planning vs. Queuing Batch Systems

Planning batch systems, such as the computing center software (CCS) [KR01], create a *schedule* with start times of all requests. The execution estimate of submitted jobs is a mandatory information for planning systems. With every incoming request or request that ends before it was estimated, planning systems compute a new schedule. The earliest suitable gap (ESG) and local search (LS) based optimization routine are examples of planning-based scheduling [KR11].

Queuing batch systems, such as Slurm [YJG03] and PBS [Hen95], hold several queues with different configurations (limits on requested resources or on requested time). Users of queuing batch systems submit their queues to specific queues. Batch queuing systems assign free resources to the waiting jobs in the queues. They apply various queuing-based job scheduling techniques, also known as priority scheduling techniques, to select a job for execution. First come first serve (FCFS), earliest deadline first (EDF), and shortest job first (SJF) are examples of queuing-based scheduling [KMR07; ABS+11]. Most of the existing batch systems are queuing systems [HKK+03].

2.2.3 Queuing-based Job Scheduling

In FCFS, the batch scheduler sorts the queue based on the job submission time. Jobs with the earliest submission time become at the head of the queue. In EDF, the batch scheduler sorts the queue based on the job due date (deadline). The job with the soonest deadline becomes the head of the queue. In SJF, the batch scheduler sorts the queue based on the job expected execution time. The job with the minimum expected execution becomes the head of the queue. The batch system does not start the job at the head of the queue unless all its required resources are free. In many cases, the available resources may be sufficient to start other jobs rather than the job at the head of the queue. These cases motivate the backfilling (BF) scheduling technique [FW98].

BF is a supporting scheduling technique that allows scheduling of jobs out of order from a given queue as long as those jobs do not delay the start time of jobs placed at the beginning of the queue [FW98]. BF helps to execute small jobs (which request a small number of computing resources) when insufficient available computing resources are needed to execute the highest priority jobs. BF is classified into conservative BF and EASY BF. Conservative BF only chooses for execution the small jobs (with short execution time and requests a few computing resources) that their execution will not cause a delay to any of the waiting jobs, including the job at the head of the queue. In contrast, EASY BF only ensures that the waiting job at the queue's head will not be delayed when the small jobs are executed.

Most of the production batch systems, such as Slurm [YJG03], LSF [IBM16], and PBS [Hen95], allow user to define custom priority scheduling. For instance, the batch system may be configured to higher priorities to the jobs submitted by a certain user or group of users. Also, many fairness policies may be applied. For instance, a fair-share scheduling technique prioritizes queued jobs such that an under-serviced user is scheduled first. The goal of such a fair-share scheduling technique is maintaining the same average job waiting time across all users.

2.2.4 Other Job Scheduling Techniques

Gang scheduling [FR95; FJ97] allows all jobs to execute concurrently on the same set of computing resources using a time-slicing mechanism. Each job receives the request computing resources for a time slice (quantum). The scheduler then switches the context to allow another job to execute on the same computing resources set for another quantum. Gang scheduling relies on stopping one or more low-priority jobs to let high-priority jobs execute (also known as Preemption).

Bin packing [CGJ83] scheduling selects groups of jobs to launch simultaneously on one or a set of computing resources. The packed jobs are selected to maximize the utilization of the allocated resources. Gang and bin packing scheduling are not commonly used. FCFS and EASY BF are the most common job scheduling techniques for real productions HPC system [GGR+15].

2.2.5 Performance Metrics

System makespan (T_{batch}) is measured as the total execution time of the entire batch. System makespan is shown in Eq. 2.17 where T_i is the time when the first job starts and T_j is the time when the last job in the batch completes.

$$T_{batch} = T_j - T_i \tag{2.17}$$

Short system makespan indicates better system performance. However, one may not be able to use it to assess the HPC systems' scheduling techniques. In production, HPC systems continuously accept new jobs as users submit them. Hence, there is no fixed workload with a specific end. System utilization (SU) is a crucial metric that one may use to assess batch systems' performance. It refers to the percentage of the resources used over a time frame [FTK14; Xha10]. Eq. 6.1 shows the calculation of SU where T_k is the time that a computing resource k spent executing jobs, P is the total number the computing resources. SU ranges from 0% to 100%.

$$SU = \frac{\sum_{k=0}^{P-1} T_k}{P * T_{batch}} * 100\% \tag{2.18}$$

Higher values of system utilization indicate better system performance.

System throughput measures the number of jobs completed per unit time. Batch schedulers should maintain high values of system throughput. Hence, they indicate better system availability.

Average job waiting time is the average time that jobs spend waiting for resources before execution. Eq. 2.19 shows the job average waiting time, where J_i^{start} and J_i^{submit} are the start and the submit time of job J_i, respectively, and N is the total number of jobs.

$$Average\ job\ waiting\ time = \frac{\sum_{i=0}^{N-1} J_i^{start} - J_i^{submit}}{N} \tag{2.19}$$

A lower average waiting time indicates better system performance.

2.3 Related State of the Art in Scheduling

The dynamic resource ownership management (DROM) is a recent research effort that allows RMJS to address efficient resource usage challenge [DGGL+18]. DROM provides effortless malleability for RMJS that requires no change in applications' source codes. DROM exploits the finest level of parallelism to support application malleability, i.e., changing the number of the threads assigned to a computing resource to create a new room for other applications on the same computing resource. One may use DROM with load balancing libraries similar to LeWI [GCL09] (LeWI is a runtime library that uses standard mechanisms, such as OMPT [Ope20] to monitor application execution.). LeWI can enhance application performance and increase resource utilization of individual computing nodes. A holistic dynamic scheduling policy, called slowdown driven (SD-policy) [DJC19] was proposed based on DROM. The SD-policy applies *backfilling* by selecting small jobs to share nodes with other running jobs. The SD-policy depends on DROM to achieve efficient node-sharing.

DROM and the LeWI library are similar to the MLS prototype because they target the same challenge of efficient resource usage. However, DROM relies on the malleability of the parallel runtime systems used by the applications, such as OpenMP or OmpSs to change the number of active threads without affecting running applications. This may not be suitable for applications that do not use a malleable parallel runtime system, such as the message-passing interface (MPI). In contrast, the MLS prototype does not require applications to be malleable. Furthermore, it enables coordination between the scheduling of different applications via batch systems. For instance, waiting or running applications (need more computing resources) may communicate their needs to the RJMS, which requests other MPI-based applications to stop scheduling any workload on the required computing resources certain period. In this scenario, the schedulers of different applications coordinate with each other through the RJMS. When an application scheduler decides not to schedule any workload on a particular resource, the process can be entirely suspended by the operating system, and other applications can use their computing resource.

A notable research effort implemented an elastic execution framework for MPI applications [CMHG+16]. The framework introduced certain extensions to the MPI standard and to Slurm [YJG03]. These extensions permit a dynamic change of the number of processes of a given application in a way that addresses several challenges of the original dynamic process support of the MPI standard. The elastic framework requires application scientists to use the new MPI functions to support application malleability. Such a requirement could be a drawback or a limitation of the elastic MPI framework. A large-scale study that examined more than one hundred MPI applications showed that most of the MPI applications only use MPI 1.0 features [LMM+19]. For instance, non-blocking collectives and neighborhood collectives are MPI 3.0 features and found to be in less than 1% of the examined applications. The cost of rewriting working codes can be one of the reasons behind that fact.

This elastic MPI framework has the same goals as the MLS prototype. However, the MLS prototype shifts the responsibility of releasing or requesting computing resource to the application scheduler rather than the application code itself. Moreover, in the MLS prototype, allowing one application to share idle computing resources with other applications does not require shrinking operations at that application's side, which keeps overhead low.

3

Two-level Scheduling Simulator

Studying mutual impacts of various scheduling levels requires conducting several exploratory experiments. These experiments involve trials of many combinations of scheduling techniques. In many cases, the associated cost of such an exploratory study is unaffordable. Simulation approaches mitigate such costs and enable the study of complex systems [STL+15; MEC+20].

This chapter introduces a novel scheduling simulation approach that bridges two different scheduling simulators by exchanging scheduling information among them [EMC17b]. Based on our approach, we have developed and assessed a two-level scheduling simulator that employs well-known simulation toolkits: **SimGrid** [CGL+14] and **GridSim** [BM02]. Our two-level scheduling simulator enables the simulations of HPC workloads at fine (tasks within applications) and coarse (jobs within a workload) scales. We visualize the simulation events collected from both simulators by converting them into an OTF2-based trace [EWG+11] that is compatible with trace visualization tools, such as Vampir [KBD+08].

3.1 Application and Batch Level Scheduling Simulations

SimGrid [CGL+14] is a widely used simulation toolkit for ALS [HCB17; SBS+13; BSC+12]. SimGrid supports the development of parallel and distributed applications in heterogeneous and homogeneous environments. Recent releases of SimGrid have three different interfaces: MetaSimGrid (MSG), SimDag (SD), and Simulated MPI (SMPI). MSG simulates applications as a group of concurrent processes. SD simulates directed acyclic graphs (DAGs). SMPI executes unmodified applications written using the message passing interface (MPI) in a simulation mode. SimGrid also has a new interface called S4U that is planned to replace the other three interfaces in the future.

GridSim [BM02] is another simulation toolkit that is widely used for BLS. GridSim facilitates simulation of grids, clusters, and single processing elements. It offers support for a broad range of heterogeneous resources, including shared and distributed memory architectures. GridSim is built on top of a reliable discrete event simulation library called SimJava [How98]. The GridSim toolkit is fully implemented in Java, which promotes its portability and extensibility.

We use a SimGrid-based scheduling simulator that is based on the SD interface [EMC16] for the ALS simulations. This specific SimGrid simulator has the advantage of being experimentally verified and capable of achieving a close agreement to native executions in various scenarios [MEC+18; MEC+20]. For the BLS simulations herein, we use a GridSim-based scheduling simulator, called Alea [KMR07; KR10; KSS19]. This GridSim-based simulator has the advantage of offering a wide range of implemented and verified BLS techniques, such as FCFS, EDF, SJF, and BF [FW98].

The SimGrid [CGL+14] and GridSim [BM02] simulation toolkits are preferably used (not restricted) to support ALS and BLS, respectively. Certain research efforts (described below) studied extensions of one of these two simulation toolkits to support the simulation of other scheduling levels.

3.1.0.1 ALS Simulation Based on a BLS Simulation Toolkit

Alea [KMR07] is a GridSim-based simulator. A remarkable research effort [SBC+11] extended Alea to support ALS. Four scheduling techniques were implemented in the extension, including SS [PPC86], FSC [KW85], GSS [PK87], and FAC [FHSF92]. The extension carried over all the Alea's advantages to the ALS domain, such as application tasks being expressed in standard workload format (SWF) [Fei20] and the effect of system failures being examined with different ALS techniques. However, the extension supported ALS in such a way that it can no longer support BLS.

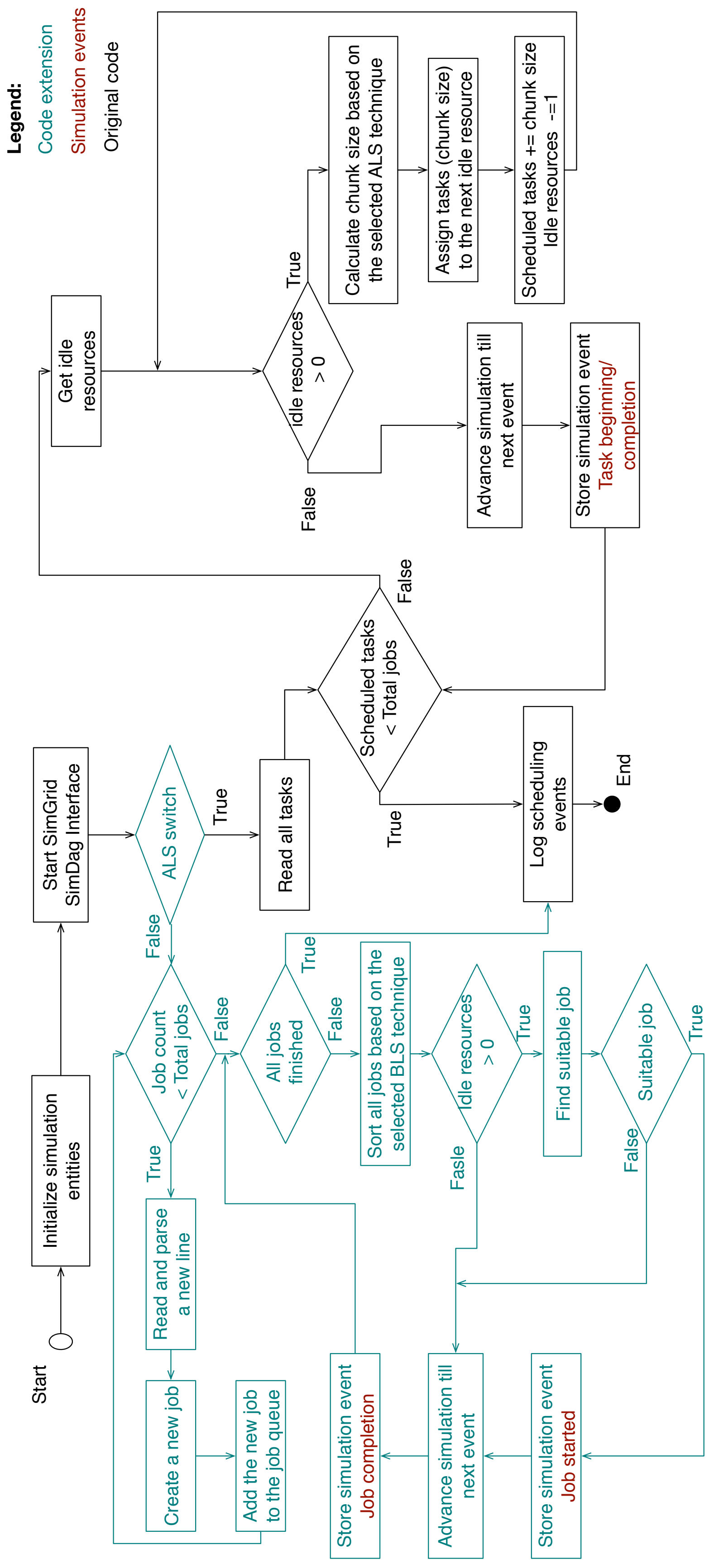
3.1.0.2 BLS Simulation Based on an ALS Simulation Toolkit

Simbatch [Can08] is a SimGrid-based simulator. Simbatch uses the MSG interface of SimGrid to support simulations and the development of BLS techniques. Simbatch's uniqueness comes from the fact that it swaps the focus of SimGrid from the ALS perspective to the BLS perspective. However, Simbatch supports two basic batch scheduling techniques FCFS [ABS+11] and BF [FW98]. Furthermore, Simbatch cannot support ALS.

Both SimGrid [CGL+14] and GridSim [BM02] have been used to support ALS and BLS simulation. **However, none of these simulators was used to support holistic simulation**. In the context of this doctoral dissertation, holistic simulation refers to ALS and BLS's simultaneous simulation. This simulation is essential to understand the BLS-ALS relation and exploit the multilevel parallelism aspect of contemporary HPC systems.

Two critical questions arise: (1) Which simulator should we extend to support BLS-ALS simulations? And (2) How does the selected simulator support BLS-ALS simulations? The answer of the first question is not straightforward, because the selected simulator has to be the most suitable one for both BLS and ALS. An important question raises regarding how suitability is defined and measured. Both the SimGrid [EMC16] and GridSim [BM02] are event-based simulators, i.e., both will have the same simulation results when one simulates the same events as the other. Therefore, we judge the "suitability" with the following two evaluation criteria: (1) the simulation wall clock and (2) the customization effort. The simulation wall clock is the time one simulator takes to conduct the simulation and produce the simulation results. The customization effort is the development effort associated with extending a simulator to support ALS or BLS simulation. The customization effort is proportional to the number of lines of code.

Figures 3.1 and 3.2 show the proposed extensions (in the GridSim [BM02] and SimGrid [CGL+14] simulators) that enable simulation of ALS and BLS. The extensions in both simulators rely on a flag called the ALS switch. If a user sets the ALS switch to true, the simulation is considered as an ALS simulation; otherwise, it is considered as a BLS simulation. One may wonder about the reason for having such a flag, i.e., can we extend each of the two simulators by implementing scheduling techniques at both levels directly? Of course, both simulators require implementing the scheduling techniques at both levels. For instance, we have implemented ALS techniques, such as FSC [KW85], GSS [PK87], and FAC [FHSF92], for the GridSim simulator. Also, we have implemented BLS techniques, such as FCFS, SJF, and EDF, for the SimGrid simulator. However, the implementation of these scheduling techniques in both simulators is insufficient to enable simulation of ALS and BLS. The entire simulation flow in both cases is not the same.



Legend:
Code extension
Simulation events
Original code
Start
Initialize simulation entities
Start SimGrid SimDag Interface
ALS switch
True
False
Read all tasks
Scheduled tasks < Total jobs
True
False
Log scheduling events
End
Get idle resources
Idle resources > 0
True
False
Calculate chunk size based on the selected ALS technique
Assign tasks (chunk size) to the next idle resource
Scheduled tasks += chunk size Idle resources -=1
Advance simulation till next event
Store simulation event Task beginning/ completion
Job count < Total jobs
True
False
Read and parse a new line
Create a new job
Add the new job to the job queue
All jobs finished
True
False
Sort all jobs based on the selected BLS technique
Idle resources > 0
True
Fasle
Find suitable job
Suitable job
True
False
Store simulation event Job completion
Advance simulation till next event
Store simulation event Job started

Figure 3.1 The execution workflow of the SimGrid simulator and the proposed extensions to support BLS.

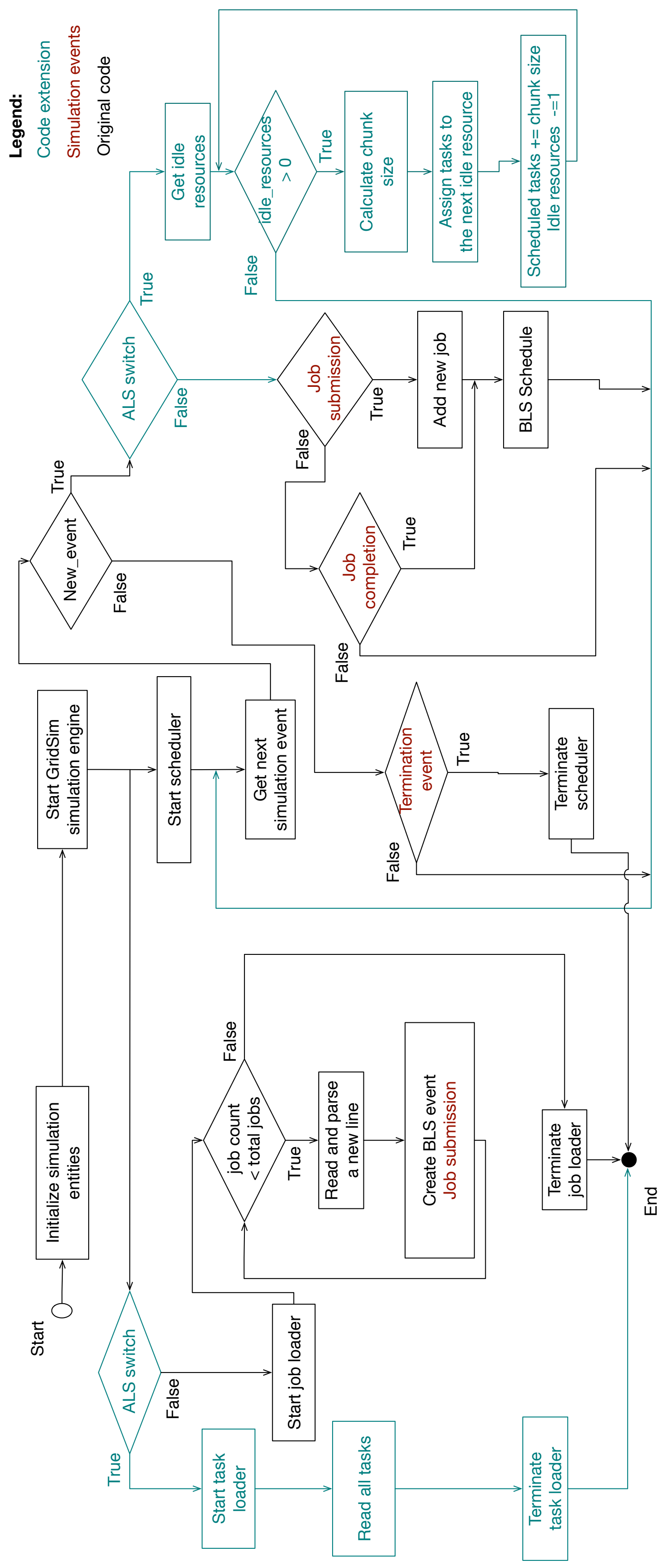


Figure 3.2 The execution workflow of the GridSim simulator and the proposed extensions to support ALS.

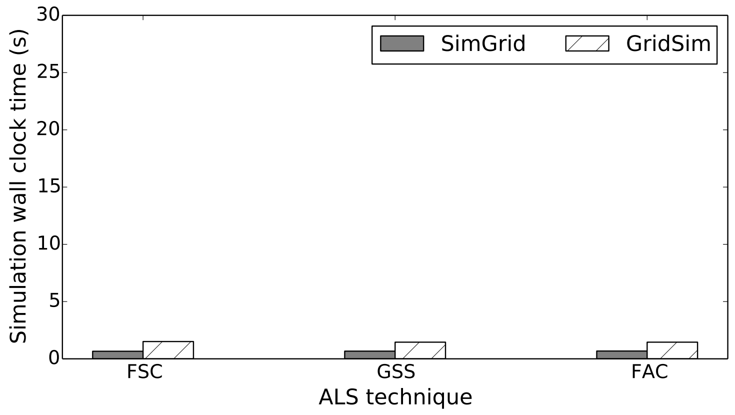
For ALS, all tasks are ready for scheduling at the same time and can be scheduled in any order. Also, each task is executed on a single PE (see Section 2.1). For BLS, jobs are submitted at different times, the job execution order is crucial, and each job may be executed on multiple PEs (see Section 2.2). The ALS switch ensures the appropriacy of the simulation flow. For instance, in Figure 3.1, if the ALS switch is true, the simulator reads all input tasks at once. The simulation flow goes as follows. (1) Get all idle PEs, (2) Calculate a chunk size for each PE based on the selected ALS technique, (3) Assign a chunk to each PE, (4) Update the total scheduled tasks, and (5) Advance simulation to the next event. These steps are repeated until all tasks are executed. When the ALS switch is set to false, the simulator reads all jobs. The simulation flow then takes another path as follows. (1) Sort all jobs based on the selected BLS technique, (2) Get idle PEs, (3) Find a suitable job, (4) Assign selected job to the idle PEs, (4) Update the total scheduled jobs, and (6) Advance simulation to the next event. These steps are repeated until all jobs are executed.

In Figure 3.2, the GridSim simulator is a multithreaded application that is based on Java. GridSim employs several objects that interact via sending/receiving scheduling events. When the ALS switch is set to true, a task loader object is created. The task loader reads all input tasks at once. For each task, it sends a scheduling event. The scheduler entity checks for idle resources. When the total number of idle resources is larger than 0, the scheduler entity schedules new tasks on the idle resources.

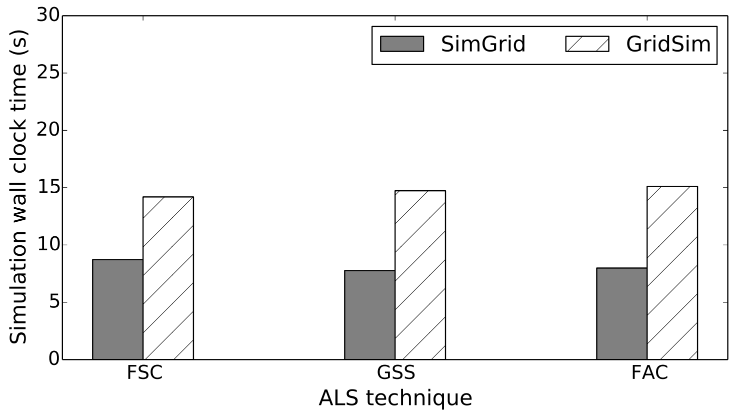
For the BLS simulation, the job loader reads all jobs. The scheduler entity checks job events, such as job submission and completion. With each of these events, the scheduler entity tries to schedule the waiting jobs by assigning them the PEs they require.

To evaluate both simulators for ALS, Lublin [LF03] has been used to generate tasks of two synthetic applications, each containing 1,115 and 65,703 tasks, respectively. We have considered three ALS techniques, namely FSC [KW85], GSS [PK87], and FAC [FHSF92]. For BLS, we have used two real HPC workloads from the parallel workload archive (PWA) [FTK14] that contains several workloads from large scale HPC systems worldwide. The first workload belongs to the High-Performance Computing Center North (HPC2N) in Sweden and contains 3,100 jobs. The second workload belongs to the Czech National Grid Infrastructure (NGI) MetaCentrum and contains 17,800 jobs. Also, We have considered three BLS techniques, namely FCFS, EDF, and SJF [ABS+11].

Figure 3.3 shows the performance of the SimGrid and the GridSim simulators in terms of simulation wall clock time for the selected ALS techniques. One can notice that the SimGrid simulator outperforms the GridSim simulator in both cases (small- and large-scale applications). In contrast, Figure 3.4 shows that the GridSim simulator outperforms the SimGrid simulator in the case of large-scale workloads. The reason for this advantage is that the BLS simulator has to sort



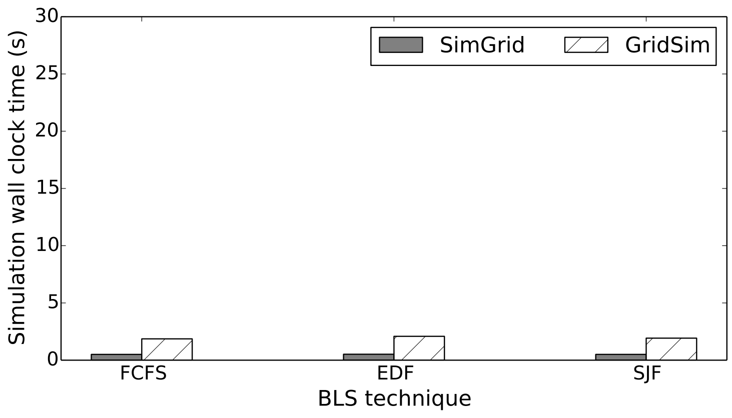
(a) Small-scale application – Total number of tasks 1,115



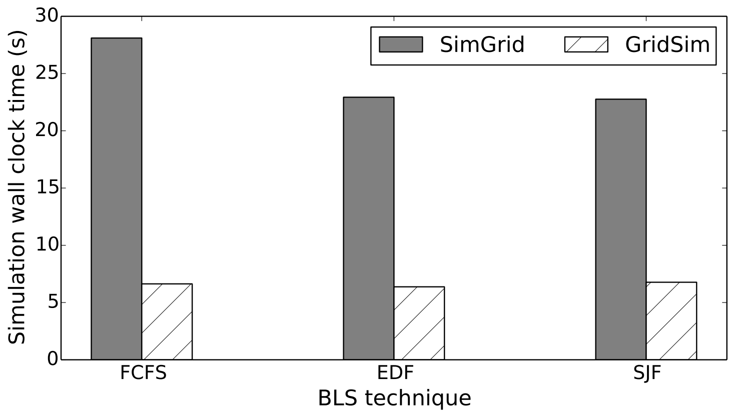
(b) Large-scale application – Total number of tasks 65,703

Figure 3.3 Performance of the SimGrid and GridSim simulators in terms of simulation wall clock time for the selected ALS techniques.

all the jobs with every incidence of simulation events, such as job submission and completion. This repetitive sort is mandatory because simulation events change the priority of the waiting jobs. The GridSim simulator exploits its underlying toolkit that is written in Java. This allows the GridSim simulator to use complex data structures, such as priority queues. The use of such complex data structures reduces the high cost of the repetitive sort. Bringing the same advantage to the



(a) Small-scale workload (HPC2N) – Total number of jobs 3,100



(b) Large-scale workload (NGI) – Total number of jobs 17,800

Figure 3.4 Performance of the SimGrid and GridSim simulators in terms of simulation wall clock time for the selected BLS techniques.

SimGrid simulator requires more development efforts to build and integrate the same complex data structures that the GridSim simulator uses.

In the ALS simulations, scheduled tasks have no dependencies or priority to execute. Therefore, these tasks are not required to be sorted for execution in any order. Consequently, the SimGrid simulator, written in C, outperforms the GridSim simulator.

We conclude that both simulators can support ALS and BLS. However, the GridSim simulator has several advantages in simulating BLS techniques, while SimGrid has other benefits in simulating ALS techniques [EMC16].

3.2 Proposed Scheduling Simulation Approach

Considering each simulator's advantages, we propose a novel simulation approach by employing a simultaneous execution of two scheduling simulators. Hence, each simulator is responsible for simulating scheduling at a certain level (cluster and node level). The proposed simulation approach is inspired by the multiscale modeling approach that provides knowledge about complex systems by modeling the interaction between phenomena at different scales [BMB+13; CFK+18; TES+19]. The proposed approach allows simulators to feed each other with their scheduling decisions when needed throughout the simulation. Figure 3.5 illustrates an example in which the BLS simulator simulates a batch of jobs and requires as input three critical parameters: a set of batch jobs, a set of

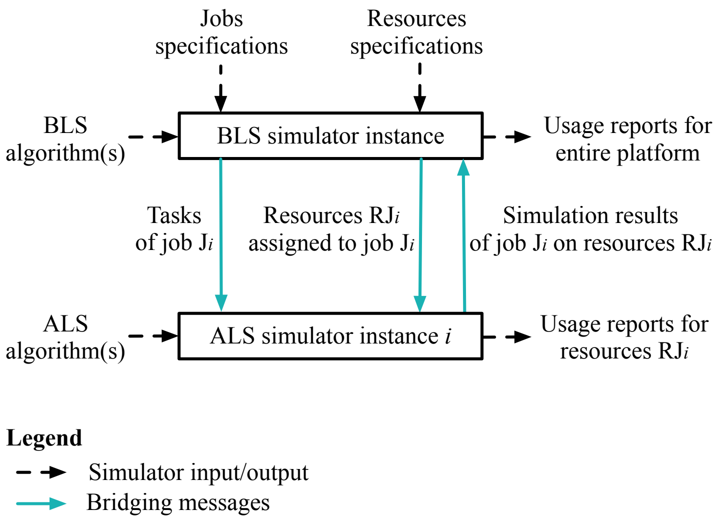

Jobs specifications
Resources specifications
BLS algorithm(s)
BLS simulator instance
Usage reports for entire platform
Tasks of job J*i*
Resources RJ*i* assigned to job J*i*
Simulation results of job J*i* on resources RJ*i*
ALS algorithm(s)
ALS simulator instance *i*
Usage reports for resources RJ*i*
Legend
Simulator input/output
Bridging messages

Figure 3.5 Bridging simulator instances for allocating resources RJ$_i$ to job J$_i$ using a certain BLS techniques and executing J$_i$ on RJ$_i$ according to a given ALS techniques.

cluster resources, and a selected BLS technique. The BLS simulator allocates the cluster resources to execute a specific job from the batch at a particular time. The BLS simulator feeds its decision to the ALS simulator, instantiated for that particular job, with three parameters: tasks of that particular job, description of the allocated resources, and the selected ALS technique.

3.3 Bridging an ALS Simulator with a BLS Simulator

A new two-level scheduling simulator is designed and implemented by connecting and integrating two different simulators. The SimGrid-based simulator [EMC16] is used to simulate ALS techniques, while the GridSim-based simulator [KMR07] is used to simulate BLS techniques. The connection between these simulators poses the following implementation challenges. (1) Interfacing two different programming models: structured and object-oriented programming were used for developing the SimGrid (in C) and the GridSim (in Java) simulators, respectively. (2) Synchronizing the independent simulation clocks of the simulators' instance. SimGrid and GridSim are based on discrete events, and each keeps its simulation clock that is only advanced when an internal event occurs. (3) Merging the output results generated by the multiple instances of the two simulators to enable a proper informative presentation. We propose a connection layer that manages simulator instances, synchronizes the clocks of the simulator instances, and exchanges necessary information regarding jobs, tasks, and other simulation parameters.

Table 3.1 summarizes the notation we use to explain the proposed connection layer in the following scenario. A batch J consists of four jobs $\{J_0, J_1, J_2, J_3\}$. Each job consists of three tasks. In each job, the sum of the length of the first two tasks is equal to the length of the third task, i.e., $LT_1 + LT_2 = LT_3$. A cluster R consists of five homogeneous resources $\{R_0, R_1, R_2, R_3, R_4\}$. The set of resources required by job J_i is denoted RJ_i, $0 \leq i < 5$. The following resource assignments are requested: $RJ_0 = \{R_0, R_1\}$, $RJ_1 = \{R_2, R_3\}$, $RJ_2 = \{R_2, R_4\}$, and $RJ_3 = \{R_0, R_4\}$. The arrival time of job J_i is AT_i, $0 \leq i < 5$, where $AT_0 = AT_1 = 0$ and $AT_2 < AT_3$. The finishing time of job J_i is FT_i, $0 \leq i < 5$, and $FT_0 = FT_1 > AT_3 > AT_2$. FCFS and STATIC are used at BLS and ALS, respectively.

Since $AT_0 = AT_1 = 0$, the connection layer manages the BLS and ALS simulator instances by starting two separate instances of the SimGrid-based simulator. These independent instances simulate the execution of jobs J_0 and J_1 on RJ_0 and RJ_1 using STATIC. Given that $FT_1 > AT_2$ and $RJ_1 \cap RJ_2 = \{R_2\}$, the connection layer *holds* the simulation of J_2 until the SimGrid simulation instance for J_1 reports its completion. Since $AT_3 > AT_2$, J_2 starts before J_3. Thus, the connection layer holds the simulation of J_3 until the SimGrid simulation instances for J_0 and J_2 report their completion. Given that $RJ_3 \cap RJ_0 = \{R_0\}$ and $RJ_3 \cap RJ_2 = \{R_4\}$, the time at which simulation of J_3 begins depends on the time at which the

Table 3.1 Notation of the proposed connection layer for the two-level scheduling simulation approach

Symbol	Description
J	Set of batch jobs
M	Number of cluster resources
N	Number of jobs
J_i	Set of batch jobs $\{J_i \mid 0 \leq i < N\}$
R	Set of cluster resources $\{R_j \mid 0 \leq j < M\}$
RJ_i	Set of resources allocated to job J_i $RJ_i \subseteq R, RJ_i \neq \emptyset, 0 \leq i < N$
AT	Set of jobs arrival times $\{AT_i \mid 0 \leq i < N\}$
FT	Set of jobs finishing times $\{FT_i \mid 0 \leq i < N\}$
ST	Set of jobs starting times $\{ST_i \mid 0 \leq i < N\}$
LJ_i	Length of job J_i (in GFLOP), where $0 \leq i < N$
TJ_i	Set of all tasks belonging to job J_i, where $0 \leq i < N$
LT_k	Length of task T_k (in GFLOP) of job J_i, where $0 \leq k < \lvert TJ_k \rvert$ and $0 \leq i < N$
Υ	Task variation factor $0 \leq \Upsilon < 1$
System_makespan$_b$	Time to complete all jobs of a certain workload, where each job has an equal number of tasks $\max(FT) - \min(AT) \mid \Upsilon = 0, \forall J_i \in J$
System_makespan$_\Upsilon$	Time to complete all jobs of a certain workload, where the sizes of tasks within each job varies according to Υ $\max(FT) - \min(AT) \mid 0 < \Upsilon < 1, \forall J_i \in J$

simulation of J_0 or J_2 completes. The finishing times of J_0 and J_2 are dominated by the scheduling decisions of the ALS techniques. Recall that for jobs J_0 and J_2, the sum of the first two tasks equals the length of the third task. Due to using STATIC as ALS and having homogeneous resources, load imbalance arises in executing the three tasks of J_0 and J_2 on the sets of resources RJ_0 and RJ_2. Consequently, the BLS scheduler, FCFS, needs to delay the beginning of the execution of J_3. The influence between BLS and ALS becomes visible via the fact that STATIC as ALS affects the individual performance of J_0 and J_2 and the performance achieved by FCFS, as well. In this scenario, if the FCFS technique passed certain information to the STATIC technique to prioritize the release of resources, the STATIC technique would assign the smallest chunk of tasks to the resources needed to be released for other jobs, such as R_0 and R_4.

The connection layer synchronizes the running simulators using two strategies: simulation suspend/resume and event injection, as illustrated in Figure 3.6. A *simulation suspend/resume entity* registered in GridSim-based is used to suspend and resume the BLS simulation. It performs a busy loop that ends if and only if all running instances of the SimGrid-based simulator report their completion and results. Because the suspend entity is a registered GridSim entity, its busy loop can pause the simulation clock of the GridSim-based simulator until the busy loop ends.

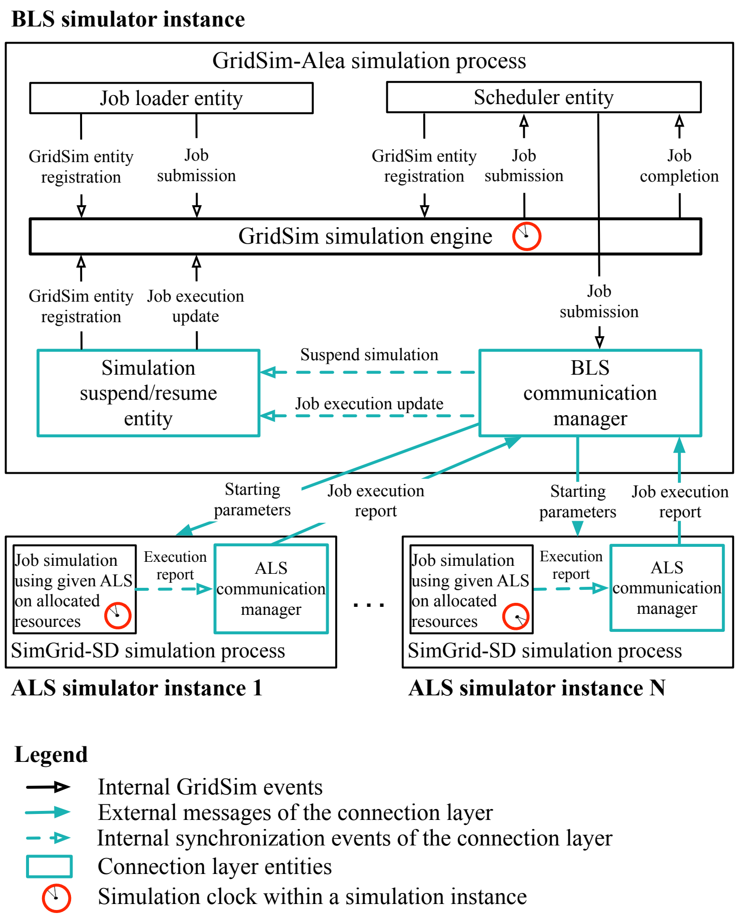

BLS simulator instance
GridSim-Alea simulation process
Job loader entity
Scheduler entity
GridSim entity registration
Job submission
GridSim entity registration
Job submission
Job completion
GridSim simulation engine
GridSim entity registration
Job execution update
Job submission
Simulation suspend/resume entity
Suspend simulation
Job execution update
BLS communication manager
Starting parameters
Job execution report
Starting parameters
Job execution report
Job simulation using given ALS on allocated resources
Execution report
ALS communication manager
SimGrid-SD simulation process
ALS simulator instance 1
. . .
Job simulation using given ALS on allocated resources
Execution report
ALS communication manager
SimGrid-SD simulation process
ALS simulator instance N
Legend
Internal GridSim events
External messages of the connection layer
Internal synchronization events of the connection layer
Connection layer entities
Simulation clock within a simulation instance

Figure 3.6 The two-level scheduling simulator. The two-level simulator consists of a single BLS and several ALS simulation instances. The connection layer synchronizes the independent simulation clocks of the GridSim and SimGrid simulators.

The internal synchronization events in Figure 3.6 are created by the BLS communication manager and used to update the simulation suspend/resume entity.

Thus, the suspend/resume entity can incrementally inject the execution reports of the running SimGrid simulation instances into the GridSim engine (see Figure 3.6). Furthermore, the suspend/resume entity can end the GridSim engine busy loop when there are no more running SimGrid simulation instances. The simulation suspend/resume entity injects the execution reports as GridSim events. Therefore, the GridSim engine can use them to advance its simulation clock. Figure 3.6 depicts the GridSim and SimGrid simulators' independent simulation clocks and their synchronization by connection layer..

The connection layer uses socket-based communication and application arguments to exchange the information between the GridSim simulator and the SimGrid simulator instances. The connection layer launches SimGrid simulator instances as independent application processes and passes certain parameters as application arguments to each established process.

3.4 From High Level to Detailed HPC Workload Representation

A holistic simulation of ALS and BLS requires information relevant to both levels. For the available HPC workloads [FTK14], it is often the case of keeping only the information relevant to BLS. The workloads in PWA [Fei05] only keeps information such as job ID, submission time, wait time, allocated resources and user ID. Additional details regarding the ALS are essential. For instance, ALS simulations require details regarding the characteristics of the application and the number of parallel tasks within the application.

Since this information is not presented in the PWA workloads, we make the following assumptions:

1. All jobs in the workload are **computationally-intensive**. Consequently, all communication or I/O tasks that may exist in the original jobs are not considered. This assumption is not a limitation of the proposed approach. It is simply used to convert the existing workloads to one of the possible cases where jobs are computationally-intensive, as such jobs are among the main incentives for using HPC systems.

2. Although the number of tasks and the length of each task are application-dependent, we consider the case of ideal parallelism. All available hardware parallelism is exploited, execution is perfectly load balanced, communication is virtually instantaneous, and the resources allocated to tasks

are identical. We consider that case of ideal parallelism as our baseline case.

Also, other cases are generated and examined by introducing a variation at the task length using the task variation factor Υ. By considering job J_i and its allocated set of resources RJ_i, the elements of the set TJ_i of tasks of job J_i can be randomly generated according to a probability distribution with a mean $\mu = \frac{LJ_i}{|RJ_i|}$ and a standard deviation $\sigma = \mu \times \Upsilon$.

3.5 Performance Evaluation and Discussion

We consider two of the most recent workloads in PWA [Fei05]. Table 3.2 summarizes the characteristics of these two workloads. We also consider combinations

Table 3.2 Characteristics of the workloads selected from the parallel workload archive (PWA)

Workload	W_1	W_2
Provenance	Curie supercomputer operated by CEA	Thunder Linux cluster operated by LLNL
Period of time	Feb, 2011 – Oct, 2012	Jan, 2007 – Jun, 2007
Total number of jobs	312,000	121,000

of three BLS techniques: FCFS, EDF, and SJF [ABS+11], and four ALS techniques: STATIC [LTS+93], SS [PPC86], GSS [PK87], and FAC [FHSF92]. For the coarse-grain analysis, jobs of the most intensive 24 hours in terms of job arrival time have been selected from both W_1 and W_2. These most intensive 24-hour intervals of W_1 and W_2 are referred to as W_1^{24} and W_2^{24}.

In all experiments, a simulated platform that consists of four hosts is used. Each of the hosts has a processor that contains 64 cores. The maximum performance of one host is 3 TFLOP/s. A fully connected network topology is used to connect the four hosts. The network model used is an InfiniBand model with a link bandwidth and latency of 50 Gbps and 500 ns, respectively.

Figures 3.7 and 3.8 show the total system makespan of W_1^{24} and W_2^{24} using the twelve combinations of BLS-ALS techniques. Each job in the two workloads is divided into a number of identical length tasks equal to the number of allocated resources. The task length variation factor Υ is not used in these experiments. The results showed in Figures 3.7 and 3.8 correspond to the best case scenario in which all submitted applications are perfectly optimized for their allocated

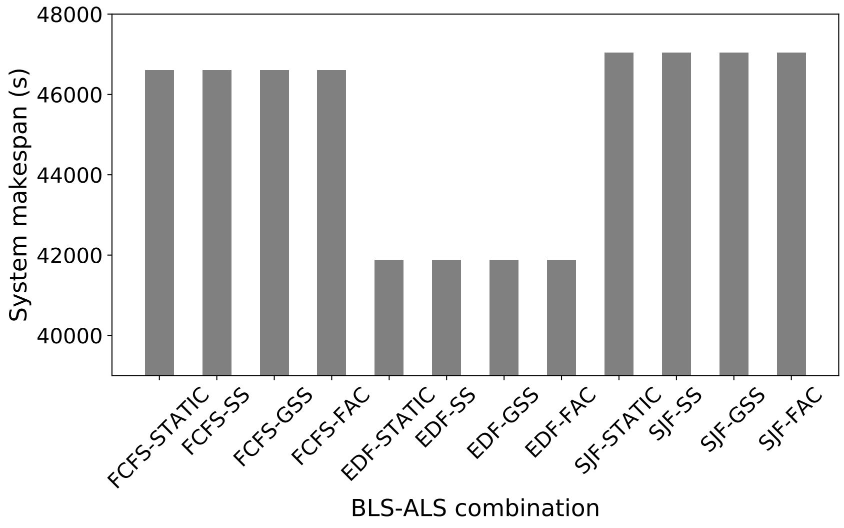


Figure 3.7 The system makespan of the W_1^{24} workload for several BLS-ALS combinations. The W_1^{24} workload consists of 1,700 jobs. Three BLS techniques (FCFS, EDF, and SJF) and four ALS (STATIC, SS, GSS, and FAC) techniques are used to form twelve BLS-ALS scheduling combinations.

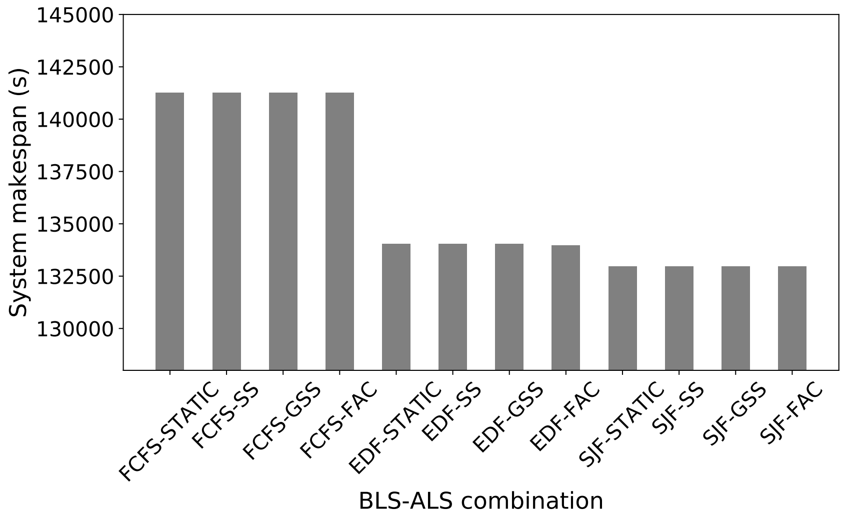


Figure 3.8 The system makespan of the W_2^{24} workload for several BLS-ALS combinations. The W_2^{24} workload consists of 3,100 jobs. Three BLS techniques (FCFS, EDF, and SJF) and four ALS (STATIC, SS, GSS, and FAC) techniques are used to form twelve BLS-ALS scheduling combinations.

resources. Although such a scenario is highly desirable both at the cluster operation level and at the user level, it is infeasible in practice. The task length variation factor Υ is used to vary the lengths of tasks within a certain job to represent more realistic applications.

Figures 3.9 and 3.10 show the effect of increasing Υ from 0.0 (as considered

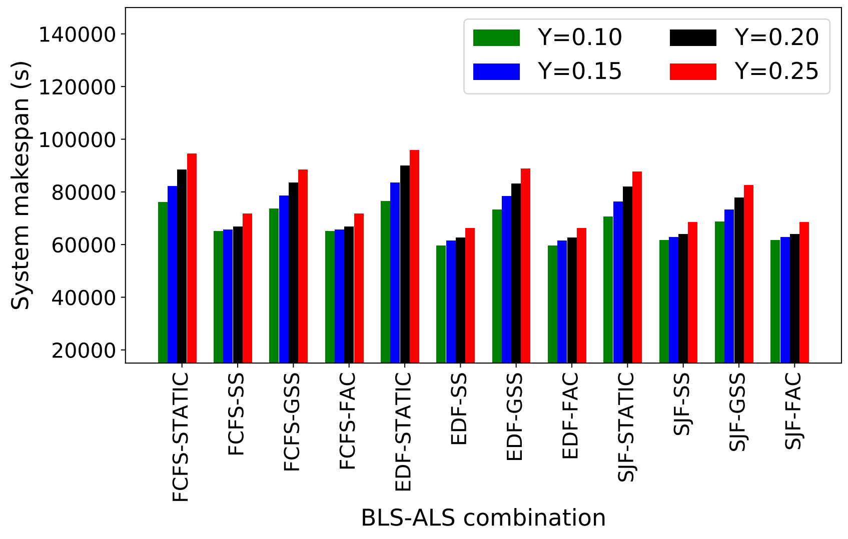


Figure 3.9 **Effect of changing the task variation factor** Υ from 0.1 to 0.25 on the total workload makespan for the twelve combinations of selected BLS and ALS techniques for the jobs within W_1^{24}.

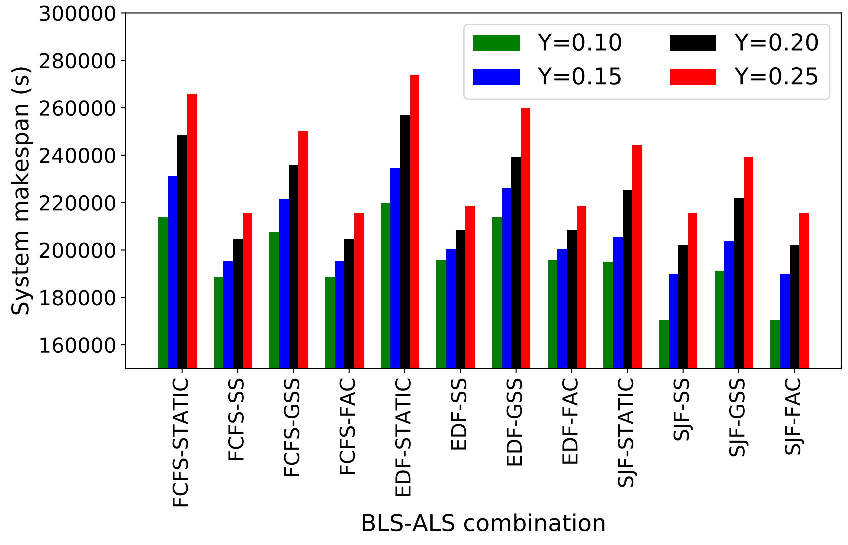


Figure 3.10 **Effect of changing the task variation factor** Υ from 0.1 to 0.25 on the total workload makespan for the twelve combinations of selected BLS and ALS techniques for the jobs within W_2^{24}.

in Figure 3.7 and Figure 3.8) to 0.1, 0.15, 0.2, and 0.25, respectively for the twelve combinations of BLS and ALS techniques. One can infer that increasing Υ leads to an increase in the total makespan of both workloads W_1^{24} and W_2^{24}, regardless of the BLS-ALS combination used. The amount of time corresponds to this increase in the total makespan is not constant across all BLS-ALS combinations

System_makespan$_\Upsilon$ represents the amount of time required to complete all jobs of a particular batch of jobs in the presence of Υ, while system_makespan$_b$

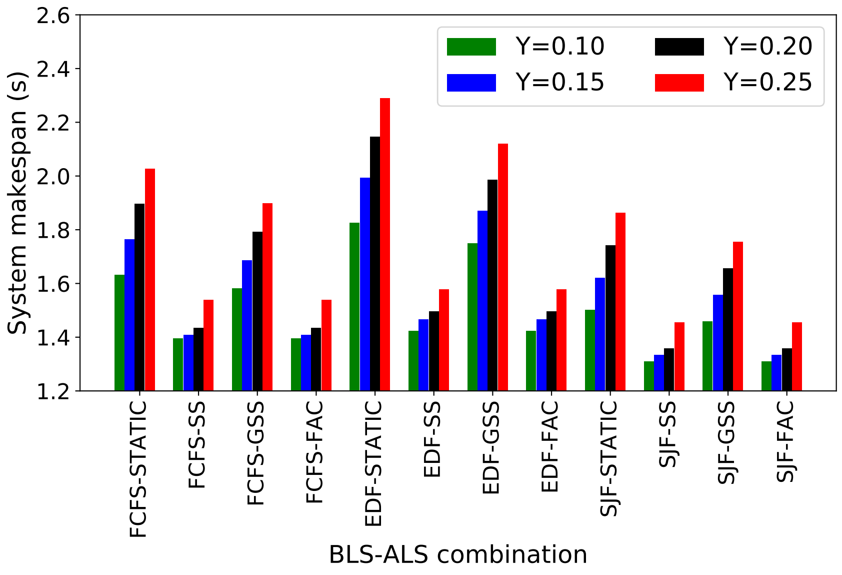


Figure 3.11 The ratio between system_makespan$_\Upsilon$ and system_makespan$_b$ for the twelve combinations of selected BLS and ALS techniques for the jobs within W_1^{24}.

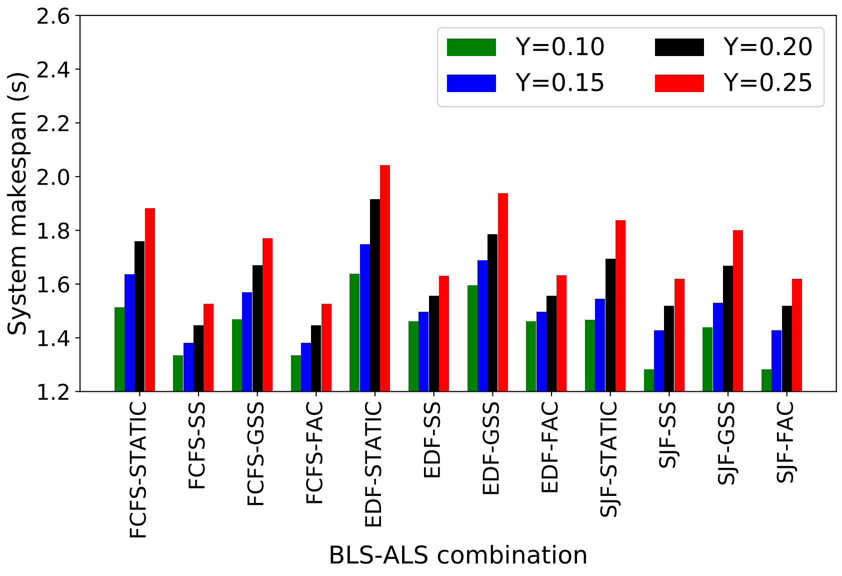


Figure 3.12 The ratio between system_makespan$_\Upsilon$ and System_makespan$_b$ for the twelve combinations of selected BLS and ALS for the jobs within W_2^{24}.

is the amount of time required to complete all jobs of a particular batch of jobs in the absence of Υ. The ratio $\frac{\text{system\_makespan}_\Upsilon}{\text{system\_makespan}_b}$ identifies the BLS-ALS combinations that better absorb the effect of increasing Υ. One can notice that SS's presence in the BLS-ALS combination fortifies the BLS technique's performance, i.e., the BLS-ALS combination absorbs the effect of increasing Υ.

For the fine-grain analysis, the connection layer between GridSim and Sim-

Grid simulators was extended with an additional task: to collect all text-based traces generated from the SimGrid-SD-based simulator and to combine them into a single text-based trace file. The main challenge associated with this task is that each instance of the SimGrid simulator does not have the global view of the entire batch workload simulation. For instance, to simulate jobs J_1 and J_2 on the sets of resources JR_1 and JR_2 at times t_1 and t_2, respectively, the connection layer runs two instances of the SimGrid simulator. Each SimGrid instance simulates its corresponding job as J_i on the set of resource JR_i at time t_x.

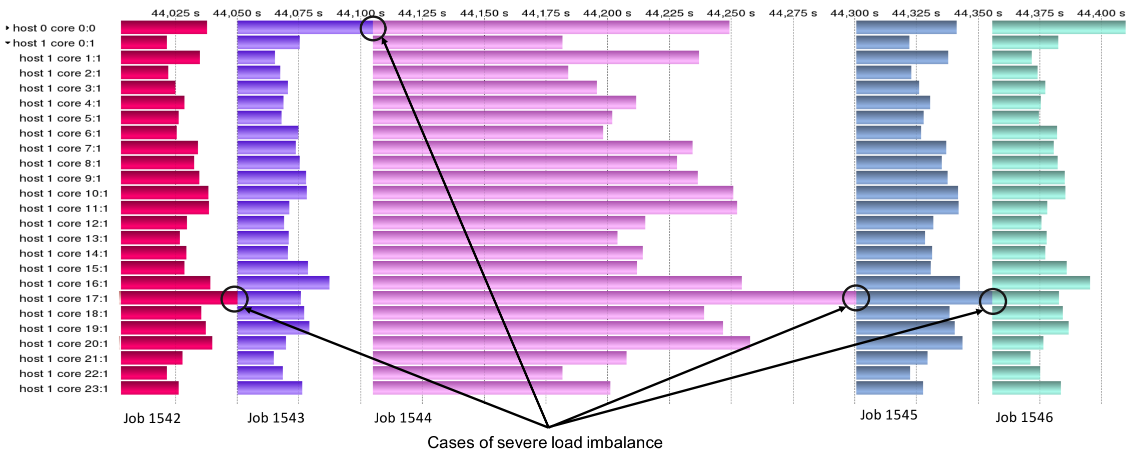


Figure 3.13 Snapshot of the Vampir visualization tool showing the generated OTF2 trace of the proposed two-level scheduling simulator. The execution of different jobs and their tasks are shown according to their allocated resources at node and core levels, respectively. Tasks of the same job are represented using horizontal bars of the same color, while the white space between the job bars represents the idle state of the allocated cores. For simplicity, this snapshot shows five different jobs running over four simulated nodes (hosts). The illustration only contains 24 cores of host 1, while host 0 is collapsed, and hosts 2 and 3 are not shown. The scheduling algorithms shown herein are FCFS and GSS at BLS and ALS, respectively. Jobs are obtained from workload W_1^{24}.

We developed a tool to convert the collected traces into a single binary trace in the OTF2 [EWG+11] format. Using OTF2 traces with the Vampir [KBD+08] trace visualizer, we visualized the cluster utilization from the node to the core level and from batch level to application level scheduling, as shown in Figure 3.13. A snapshot captured from Vampir is included in Figure 3.13 and shows the execution of five out of 1,700 running jobs, namely J_{1542}, J_{1543}, J_{1544}, J_{1545}, and J_{1546}, from the W_1^{24} workload. The execution was performed with FCFS-GSS. Figure 3.13 illustrates a case of severe load imbalance of certain jobs, its effects on the starting times of subsequent jobs in the batch, and, consequently, the effects on the entire system performance and utilization.

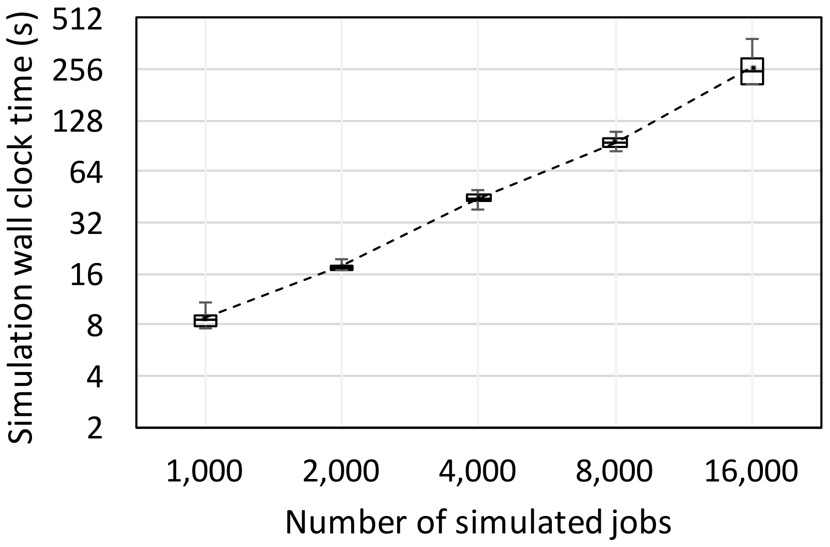


Figure 3.14 The simulation wall clock time of the two-level scheduling simulator on an increasing number of jobs from workloads W_1.

Scalability in terms of increasing the number of jobs is a critical aspect of the proposed two-level. An initial scalability assessment of the two-level scheduling simulator is presented in Figure 3.14. In these experiments, the simulation wall clock time is reported for executing an increasing number of jobs (from 1,000 to 16,000) from the workloads W_1. The simulation wall clock is defined as the total time required to simulate the execution of all jobs of a given workload. The experiments were conducted with the least performing BLS-ALS combination, i.e., FCFS-STATIC, with $\Upsilon = 0.25$. selected from the results in Figure 3.9. Figure 3.14 includes the average, maximum, and minimum simulation wall clock time, where each experiment was executed ten times. The results shows the relation between the increase in the number of simulated jobs and the proportional increase in the simulation wall clock time produced by the two-level scheduling simulator.

3.6 Summary

We presented a novel simulation approach that bridges two different scheduling simulators by exchanging the bridged simulators' scheduling information. The proposed simulation approach was exemplified with a two-level scheduling simulator that connects two existing simulators: SimGrid [EMC16] and GridSim [KMR07]. We used the proposed two-level scheduling simulator to explore twelve combinations of 4 ALS techniques and 3 BLS techniques. We conclude that ALS techniques affect the performance of their applications and the performance of the entire batch system. Hence, the overall performance of any given BLS technique is affected by the load imbalance at every individual job.

This conclusion guides our work into the following directions: (1) **avoiding** or minimizing load imbalance, and if it is not avoidable, (2) **exploiting** load imbalance. The first direction is covered by the work presented in Chapters 4 and 5, and the second direction is covered by the work presented in Chapter 6.

4

Distributed Chunk Calculation Approach (DCA)

The advancements in modern HPC systems at both hardware and software levels raise questions regarding the benefits of these advancements for successful techniques proposed in the past. We need to revisit and re-evaluate these techniques to fully leverage modern HPC systems' capabilities at both hardware and software levels. As discussed in Chapter 1, we focus on DLS techniques at the application level. We showed in Chapter 3 that DLS techniques have a substantial impact on application performance and batch system performance.

This chapter examines the typical execution approaches for DLS techniques, specifically the master-worker execution model. We discuss the influence of centralizing the chunk calculation at the master on the DLS techniques' performance. Motivated by the advancements in the MPI [For20] standard, we propose a distributed chunk calculation approach (DCA) [EC19a] to eliminate the use of the master-worker model. The DCA contributes to our envisioned MLS solution by minimizing the idle time of computing resources.

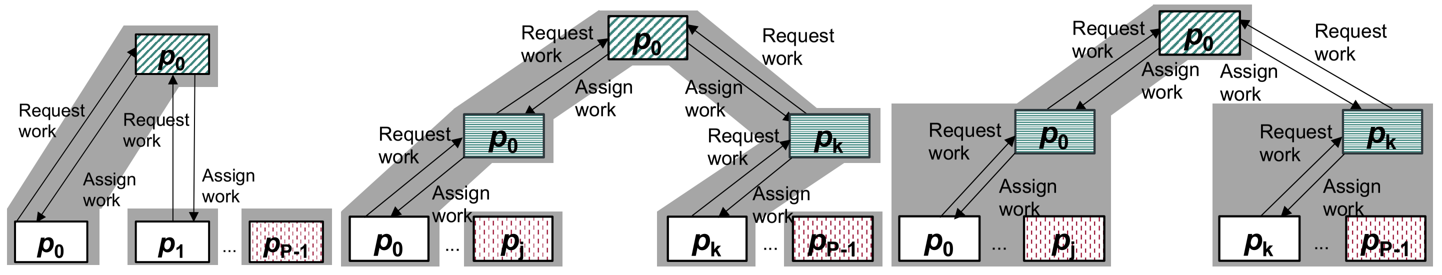
4.1 Execution Models of DLS Techniques

The self-scheduling aspect is the distinguishing aspect of all DLS techniques. Self-scheduling means that once a PE becomes free, it calculates a new chunk of loop iterations to be executed. The calculated chunk size is not associated with a specific set of loop iterations. The PE must synchronize with all other PEs to map the calculated chunk size to a specific set of unscheduled loop iterations. There are two operations at every scheduling step: chunk calculation and chunk assignment. In principle, only the chunk assignment requires global synchronization between all PEs, while the chunk calculation does not require synchronization and can be distributed across all PEs.

In practice, existing DLS execution approaches, especially for distributed-memory systems, do not consider the separation between chunk calculation and chunk assignment. Hence, the master-worker execution model dominates all existing DLS execution approaches. In the master-worker execution model, the master performs chunk calculation and chunk assignment. This centralization renders the master process a performance bottleneck in three scenarios. (1) The master process has a decreased processing capability (this may happen in heterogeneous systems). (2) The master process receives a large number of concurrent work requests (this may happen in large scale distributed-memory systems). (3) System variation delays the processing at the master.

The distributed self-scheduling scheme (DSS) [CAB+01] is an example of employing the master-worker execution model to implement DLS techniques for distributed-memory systems. DSS relies on the master-worker execution model, similar to the one illustrated in Figure 4.1(a). DSS enables the master to consider the processing elements' speed and their loads when assigning new chunks. DSS was later enhanced by a hierarchical distributed self-scheduling scheme (HDSS) [CPY+05] that employs a hierarchical master-worker model, as illustrated in Figure 4.1(b).

DSS and HDSS assume a dedicated master configuration in which the master PE is reserved for handling the worker requests. Such a configuration may enhance the scalability of the proposed self-scheduling schemes. However, it results in low CPU utilization of the master. HDSS [CPY+05] suggested deploying the global-master and the local-master on one physical computing node with multiple processing elements to overcome the low CPU utilization of the master (see Figure 4.1(b)). DSS and HDSS were implemented using MPI two-sided communications. In both DSS and HDSS, the master is a central entity that performs both the chunk calculation and the chunk assignment.



(a) Conventional master worker execution model

(b) Global and local masters are located on a single physical compute node

(c) Local masters are distributed across multiple physical compute nodes

LEGEND

▨ Global master ▤ Local master ▥ Busy worker □ Available and requesting worker

■ Physical compute node ⇄ Two-sided messages

Figure 4.1 Variants of the master-worker execution model, as reported in the literature. Replication of certain processing elements is just to indicate their double role where the master participates in the computation as a worker.

Another MPI-based library that implements several DLS techniques is called the load balancing tool (LB tool) [CB05]. At the conceptual level, the LB tool is based on a single-level master-worker execution model (see Figure 4.1(a)). However, it does not assume a dedicated master. It introduces the breakAfter (user-defined) parameter and indicates how many iterations the master should execute before serving pending worker requests. This parameter is required for dividing the time of the master between computation and servicing of worker requests. The optimal value of this parameter is application- and system-dependent. The LB tool also employs two-sided MPI communications.

LB4MPI [MEC+20; MC20] is an extension of the LB tool [CB05] that includes certain bug fixes and additional DLS techniques. Both LB and LB4MPI employ a master-worker execution in which the master is a central entity that performs both the chunk calculation and the chunk assignment operations.

The dynamic load balancing library (DLBL) [BCP+05] is another MPI-based library used for cluster computing. It is based on a parallel runtime environment for multicomputer applications (PREMA) [BCC+04]. DLBL is the first tool that employed MPI one-sided communication for implementing DLS techniques. Similar to the LB tool, the DLBL employs a master-worker execution model. The master expects work requests. It then calculates the chunk's size to be assigned and, subsequently, calls a handler function on the worker side. The worker is responsible for obtaining the new chunk data without any further involvement from the master. This means that the master is still a central entity that performs both chunk calculation and chunk assignment.

4.2 From Centralized to Decentralized DLS Techniques

The idea of DCA is to ensure that the calculated chunk size at a specific PE does not rely on any information about the chunk size calculated at any other PE. The chunk calculation formulas (Eq. 2.1 to 2.13) can be classified into *straightforward* and *recursive* formulas. A straightforward chunk calculation formula only requires some constants and input parameters. A recursive chunk calculation formula requires information about previously calculated chunk sizes. For instance, STATIC, SS, FSC, and RND have straightforward chunk calculation formulas that do not require any information about previously calculated chunks, while GSS [PK87], TAP [Luc92], TSS [TN93], FAC [FHSF92], TFSS [CAB+01], FISS [PD97], VISS [PD97], AF [Ban00], and PLS [SYT07] employ recursive chunk calculation formulas. Certain transformations are required to convert these recursive formulas into straightforward formulas to enable DCA. For GSS and FAC, the transformations were already introduced in the literature [FHSF92] (Eq. 4.1 and 4.2).

$$K_i'^{GSS} = \left\lceil \left(\frac{P-1}{P}\right)^i \cdot \frac{N}{P} \right\rceil \tag{4.1}$$

$$K_i'^{FAC2} = \left\lceil \left(\frac{1}{2}\right)^{i_{\text{new}}} \cdot \frac{N}{P} \right\rceil, \; i_{\text{new}} = \left\lfloor \frac{i}{P} \right\rfloor + 1 \tag{4.2}$$

As shown in Eq. 2.5, TAP calculates K_i^{GSS} and tunes that value based on μ, σ, and α. Based on Eq. 4.1, the chunk calculation formula of TSS can be expressed as a straightforward formula as follows.

$$\begin{aligned} K_i'^{TAP} &= K_i'^{GSS} + \frac{v_\alpha^2}{2} - v_\alpha \cdot \sqrt{2 \cdot K_i'^{GSS} + \frac{v_\alpha^2}{4}}, \text{ where} \\ v_\alpha &= \frac{\alpha \cdot \sigma}{\mu} \end{aligned} \tag{4.3}$$

For TSS, a straightforward formula for the chunk calculation is shown in Eq. 4.4.

$$K_i'^{TSS} = K_0^{TSS} - i \cdot \left\lfloor \frac{K_0^{TSS} - K_{S-1}^{TSS}}{S-1} \right\rfloor \tag{4.4}$$

The mathematical derivation that converts Eq. 2.6 into Eq. 4.4 is as follows. The TSS chunk calculation formula can be represented as follows, where C is a constant.

$$\begin{aligned} K_i^{TSS} &= K_{i-1}^{TSS} - C \\ C &= \left\lfloor \frac{K_0^{TSS} - K_{S-1}^{TSS}}{S-1} \right\rfloor \\ K_1^{TSS} &= K_0^{TSS} - C \\ K_2^{TSS} &= K_1^{TSS} - C = (K_0^{TSS} - C) - C = K_0^{TSS} - 2 \cdot C \\ K_i^{TSS} &= K_0^{TSS} - i \cdot C \\ K_i^{TSS} &= K_0^{TSS} - i \cdot \left\lfloor \frac{K_0^{TSS} - K_{S-1}^{TSS}}{S-1} \right\rfloor = K_i'^{TSS} \end{aligned}$$

TFSS [CAB+01] is devised based on TSS [TN93] and FAC [FHSF92]. Therefore, the straightforward formula of TSS (see Eq. 2.6) can be used to derive the straightforward formula of TFSS, as shown in Eq. 4.5.

$$K_i'^{TFSS} = \frac{\sum_{j=i}^{i+P-1} K_j'^{TSS}}{P} \tag{4.5}$$

For FISS [PD97], a straightforward formula for the chunk calculation is shown in Eq. 4.6.

$$K_i'^{FISS} = K_0^{FISS} + i \cdot \lceil \frac{2 \cdot N \cdot (1 - \frac{B}{2+B})}{P \cdot B \cdot (B-1)} \rceil \tag{4.6}$$

The mathematical derivation that converts Eq. 2.10 into Eq. 4.6 is as follows. Given that A is a constant, the FISS chunk calculation formula can be represented as follows, where C is a constant.

$$\begin{aligned}
&K_i^{FISS} = K_{i-1}^{FISS} + C \\
&C = \lceil \frac{2 \cdot N \cdot (1 - \frac{B}{2+B})}{P \cdot B \cdot (B-1)} \rceil \\
&K_1^{FISS} = K_0^{FISS} + C \\
&K_2^{FISS} = K_1^{FISS} + C = (K_0^{FISS} + C) + C = K_0^{FISS} + 2 \cdot C \\
&K_i^{FISS} = K_0^{FISS} + i \cdot C \\
&K_i^{FISS} = K_0^{FISS} + i \cdot \lceil \frac{2 \cdot N \cdot (1 - \frac{B}{2+B})}{P \cdot B \cdot (B-1)} \rceil = K_i'^{FISS}
\end{aligned}$$

For VISS [PD97], a straightforward formula for the chunk calculation is shown in Eq. 4.7.

$$\begin{aligned}
&K_i'^{VISS} = K_0^{FISS} \cdot \frac{1-(0.5)^{i_{new}}}{0.5}, \text{where } \ i > 0 \\
&i_{new} = i \mod P \\
&K_0'^{VISS} = K_0^{FISS}
\end{aligned} \tag{4.7}$$

To derive Eq. 4.7, we calculate K_1^{VISS}, K_2^{VISS}, and K_3^{VISS}, according to Eq. 2.11.

$$\begin{aligned}
&K_1^{VISS} = K_0^{FISS} + \frac{K_0^{FISS}}{2}, \text{assume } \ K_0^{FISS} = a \\
&K_1^{VISS} = a + \frac{a}{2} \\
&K_2^{VISS} = K_1^{VISS} + \frac{K_1^{VISS}}{2} = (a + \frac{a}{2}) + (\frac{a + \frac{a}{2}}{2}) \\
&K_3^{VISS} = K_2^{VISS} + \frac{K_2^{VISS}}{2} = ((a + \frac{a}{2}) + (\frac{a + \frac{a}{2}}{2})) + \frac{((a + \frac{a}{2}) + (\frac{a+\frac{a}{2}}{2}))}{2}
\end{aligned}$$

According to the geomertic summation theorem

$$K_i^{VISS} = K_0^{FISS} \cdot \frac{1-(0.5)^i}{0.5}$$

since VISS assigns chunks in batches

$$K_i^{VISS} = K_0^{FISS} \cdot \frac{1-(0.5)^{i_{new}}}{0.5} = K_i'^{VISS}, \text{where } i > 0,$$

and $i_{new} = i \mod P$.

For PLS, the loop iteration space is divided into two parts. In the first part, the PLS chunk calculation formula is equivalent to STATIC, i.e., the chunk calculation formula is a straightforward formula that is ready to support DCA. In the second part, PLS uses the GSS chunk calculation formula. Therefore, we replace K_i^{GSS} in Eq. 2.13 with $K_i'^{GSS}$ from Eq. 4.1 to derive the PLS chunk calculation (Eq. 4.8).

$$K_i'^{PLS} = \begin{cases} \frac{N \cdot SWR}{P}, & \text{if } R_i > N - (N \cdot SWR) \\ K_i'^{GSS}, & \text{otherwise.} \end{cases} \tag{4.8}$$

AF adapts the calculated chunk size according to μ_{p_i} and σ_{p_i}, which can be determined only during loop execution. Moreover, at every scheduling step, AF uses R_i with μ_{p_i} and σ_{p_i} to calculate the chunk size. This leads to an unpredictable pattern of chunk sizes and makes it impossible to find a straightforward formula for AF. Accordingly, we could not determine a way to implement AF with a *fully distributed chunk calculation*. In our implementation, AF with DCA requires additional synchronization of R_i across all PEs. All PEs can simultaneously calculate D and E from Eq. 2.14. However, each PE needs to synchronize with all other PEs to calculate each K_i^{AF}.

4.3 Distribution of the Chunk Calculation

In the proposed DCA, we replace the master role with a coordinator role. The coordinator holds a central work queue that contains the global scheduling information, such as the last scheduling step i and the last loop index start $lpstart$. One can consider the value of the last scheduling step i as the only required input parameter for the newly derived chunk calculation formulas (see Section 4.2). DCA has three major steps:

Step 1. Workers synchronize through the coordinator to exclusively get a local copy of i and increment the global i by one.

Step 2. Workers use their local copies of i to calculate their chunk sizes.

Step 3. Workers synchronize once again to exclusively get a local copy of the last loop index start lp_{start} and increment the global lp_{start} by the calculated chunk size.

At that point, every worker can execute loop iterations from lp_{start} to lp_{start} + the calculated chunk size. In the proposed DCA, the coordinator acts like a worker with one exception that is holding the central work queue.

Figure 4.2 illustrates the DLS execution using the proposed DCA. Processors p_0 and p_1 calculate K_0 and K_1 simultaneously. The time required to calculate K_0

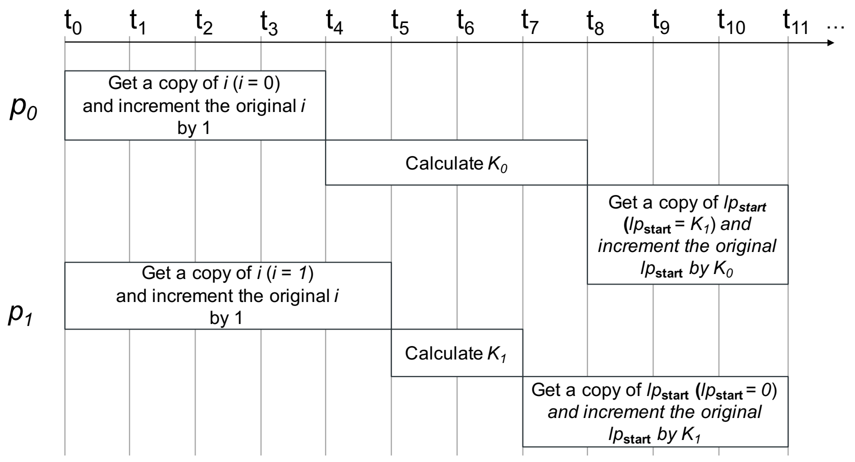

t0
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11 ...
p0
p1
Get a copy of i (i = 0) and increment the original i by 1
Calculate K0
Get a copy of lpstart (lpstart = K1) and increment the original lpstart by K0
Get a copy of i (i = 1) and increment the original i by 1
Calculate K1
Get a copy of lpstart (lpstart = 0) and increment the original lpstart by K1

Figure 4.2 Schematic execution of the proposed distributed chunk calculation approach (DCA) on two processors that calculate one chunk each.

overlaps with the time taken to calculate K_1. In the traditional master-worker execution model, there is no such overlap since all the chunk calculations are centralized and performed by the master in sequence. The time required to serve the first work request (including chunk calculation and chunk assignment) delays the second work request. Moreover, the time required to serve the work requests is proportional to the processing capabilities of the master processor, which may result in additional delays.

DCA may result in a different ordering of assigning and executing loop iterations than the traditional master-worker execution model. For instance, when GSS is the chosen scheduling technique in Figure 4.2 and $N = 10$, p_0 obtains a local copy of the last scheduling index $i = 0$ at t_4. Also, p_1 obtains at t_5 a local copy of the last scheduling index $i = 1$. Both p_0 and p_1 use their copies of i and calculate $K_0 = 5$ and $K_1 = 3$, respectively. DCA does not guarantee that p_0 and p_1 will execute loop iterations from $lp_{\text{start}} = 0$ to $lp_{\text{start}} = 4$ and $lp_{\text{start}} = 5$ to $lp_{\text{start}} = 7$. Figure 4.2 shows the case when the chunk calculation on p_0 is longer than on p_1, and results in assigning p_1, loop iterations between $lp_{\text{start}} = 0$ and $lp_{\text{start}} = 2$, while p_0 is assigned loop iterations between $lp_{\text{start}} = 3$ and $lp_{\text{start}} = 7$. Given that DLS techniques address, by design, independent loop iterations with no restrictions on the execution order of the loop iteration, DCA does not affect the correctness of the loop execution.

4.3.0.1 Proposed Implementation of DCA

The latest advancements in the MPI 3.1 standard [For20], namely the revised and the clear semantics of the MPI remote memory access (RMA) [HDT+15;

ZBG16], enabled its usage in different scientific applications [HGC14; SWZ+16; ZG16]. The MPI RMA model, also known as one-sided communication, provides the necessary function calls to implement the proposed DCA. In the MPI RMA model, each process's memory is by default private, and other processes cannot directly access it. The MPI RMA model allows MPI processes to expose different regions of their memory, called windows. One MPI process (origin) can directly access a memory window without any involvement of the other (target) process that owns the window.

The MPI RMA has two synchronization modes: passive- and active-target. In the active-target synchronization, the target process determines the time boundaries, called epochs, when its window can be accessed. In the passive-target synchronization, the target process has no time limits when its window can be accessed. The DCA can benefit from the passive-target synchronization because it requires a minimal amount of synchronization, and it efficiently allows the most significant overlap of computation and communication. Moreover, it yields the development of DLS techniques for distributed-memory systems to be very similar to their original implementations for shared-memory systems.

Figure 4.3 illustrates the main steps of the proposed DCA as follows. (1) the processing element p_j obtains a copy of the last scheduling step index, i, and atomically increments the global i by one. (2) p_j *only* uses its local copy of i (before the increment) to calculate K_i with the new derived straightforward formulas for the selected DLS technique. (3) p_j obtains a copy of the last loop index start, lp_{start}, and atomically accumulates the size of the calculated

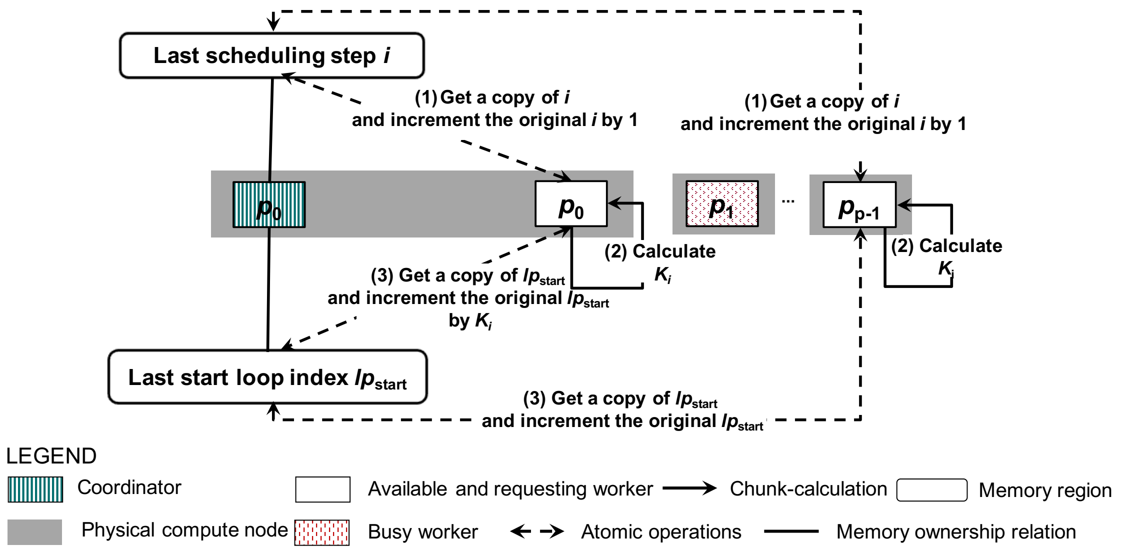

Last scheduling step *i*
(1) Get a copy of *i* and increment the original *i* by 1
(1) Get a copy of *i* and increment the original *i* by 1
p_0
p_0
p_1
p_{p-1}
(2) Calculate K_i
(2) Calculate K_i
(3) Get a copy of lp_{start} and increment the original lp_{start} by K_i
Last start loop index lp_{start}
(3) Get a copy of lp_{start} and increment the original lp_{start}
LEGEND
Coordinator
Physical compute node
Available and requesting worker
Busy worker
Chunk-calculation
Atomic operations
Memory region
Memory ownership relation

Figure 4.3 The proposed DCA using MPI RMA and passive-target synchronization.

chunk, K_i, into it. Finally, p_j executes loop iterations between lp_{start} (before accumulation) and $\min(lp_{\text{start}} + K_i, N)$. The atomic operations in Steps 1 and 3 guarantee the exclusive access to i and lp_{start}. The coordinator MPI process can use `MPI_Win_create` to expose the shared variables, such as i and lp_{start}, to all other MPI processes. The passive-target synchronization mode (`MPI_Win_-lock(MPI_LOCK_SHARED)`) can be used with certain MPI atomic operations, such as `MPI_Get_accumulate`, to grant the exclusive access to i and lp_{start} by all MPI processes.

4.4 Performance Evaluation and Discussion

We evaluated two implementations. The first implementation, denoted `One_-Sided_DLS`, employs the proposed DCA, and uses one-sided MPI communication in the passive-target synchronization mode. The second implementation, denoted `Two_Sided_DLS`, employs a master-worker model and uses the two-sided MPI communication. Both implementations assume a non-dedicated coordinator (or a non-dedicated master) processing element.

We considered two computationally-intensive parallel applications in the evaluation. The first application, called PSIA [EFM+16; EMC17a], uses a parallel version of the well-known spin-image algorithm (SIA) [Joh97]. SIA converts a 3D object into a set of 2D images. The generated 2D images can be used as descriptive features of the 3D object. The second application calculates the Mandelbrot set [Man80]. The Mandelbrot set is used to represent geometric shapes that have the self-similarity property at various scales. Studying such shapes is essential and of interest in different domains, such as biology, medicine, and chemistry [JTS09].

Both applications contain a single large parallel loop that dominates their execution times. Algorithm 4.1 and 4.2 show the pseudocodes of both applications.

Table 4.1 summarizes the execution parameters used for both selected applications. These parameters were selected empirically to guarantee a reasonable average iteration execution time that is larger than 0.2 for PISA and 0.02 seconds for Mandelbrot.

Two types of computing resources are used in our evaluation. The first type, denoted KNL, refers to a standalone Intel Xeon Phi 7210 manycore processors with 64 cores, 96 GB RAM (flat mode configuration), and 1.3 GHz CPU frequency. The second type, denoted Xeon, refers to two-socket Intel Xeon E5-2640

Table 4.1 Execution parameters of PSIA and Mandelbrot selected to evaluate the proposed DCA

Application	Input Size	Output size	Other parameters [EMC17a; Man80]

PSIA	800,000 3D points [WLD+10]	288,000 images	5x5 2D image
			bin-size = 0.01
			support-angle = 2

Mandelbrot	No input data	One image	image-width = 1152x1152 number of iterations = 1000 Z exponent = 4

processors with 20 cores, 64 GB RAM, and 2.4 GHz CPU frequency.

These platform types are part of a fully-controlled computing cluster, called miniHPC [1]. The miniHPC cluster consists of 26 nodes: 22 Xeon nodes and 4 KNL nodes. All nodes are interconnected in a non-blocking fat-tree topology. The network characteristics are: Intel Omni-Path fabric, 100 GBit/s link bandwidth, and 100 ns network latency. Each KNL node has *one* Intel Omni-Path host fabric interface adapter. Each Xeon node has *two* Intel Omni-Path host fabric interface adapters. All host fabric adapters use a single PCIe x16 100 Gbps port. As this computing cluster is actively used for research and educational purposes, only

Algorithm 4.1 Parallel spin-image calculations. The main loop is highlighted in the blue color.

```
Inputs : W: image width, B: bin size, S: support angle, OP: list of 3D points,
         M: number of spin-images
Output: R: list of generated spin-images
for $i = 0 \rightarrow M$ do
    P = OP[i];
    tempSpinImage[W, W];
    for $j = 0 \rightarrow length(OP)$ do
        X = OP[j];
        $np_i$ = getNormalVector(P);
        $np_j$ = getNormalVector(X);
        if $acos(np_i \cdot np_j) \leq S$ then
            $k = \left\lceil \frac{W/2 - np_i \cdot (X - P)}{B} \right\rceil$;
            $l = \left\lceil \frac{\sqrt{||X - P||^2 - (np_i \cdot (X - P))^2}}{B} \right\rceil$;
            if $0 \leq k < W$ and $0 \leq l < W$ then
                tempSpinImage[k, l]++;
    R.append(tempSpinImage);
```

[1] https://hpc.dmi.unibas.ch/HPC/miniHPC.html

Algorithm 4.2 **Mandelbrot set calculations. The main loop is highlighted in the blue color.**

```
Inputs : W: image width, CT: Conversion Threshold
Output: V: Visual representation of Mandelbrot set calculations
for $counter = 0 \rightarrow W^2$ do
    $x$ = counter / W;
    $y$ = counter mod W;
    $c$= complex(x_min + x/(W*(x_max-x_min)) , y_min + y/(W*(y_max-y_-min)));
    $z$ = complex(0,0);
    for $k = 0 \rightarrow CT\ OR\ |z| < 2.0$ do
        $z = z^4 + c$;
    if $k = CT$ then
        set $V(x, y)$ to black;
    else
        set $V(x, y)$ to blue;
```

40% of the cluster could be *dedicated* to the present work, at the time of writing, specifically 288 cores out of the total 696 available cores.

In the present work, the total number of cores is fixed to 288 cores, whereas the ratio between the KNL and the Xeon cores is varied. Two ratios have been considered: 2:1 represents the case when the KNL cores are the dominant type of computing resources, and 1:2 represents the complementary case where the Xeon cores are the dominant computing resources. Table 4.2 illustrates these two ratios. Also, 48 KNL cores and 16 Xeon cores per node are used, while the remaining cores on each node were left for other system-level processes.

Two mapping scenarios are considered for the assessment of the `One_Sided_DLS` approach vs. the `Two_Sided_DLS` approach. In the first mapping scenario, the process that plays the coordinator's role for `One_Sided_DLS` or the master's role for `Two_Sided_DLS` is mapped to a KNL core. The CPU frequency of a single KNL core is 1.3 GHz, while the CPU frequency of a single Xeon core is 2.4 GHz. Therefore, this mapping represents a case when the coordinator (or the master) process is mapped to one of the cores that has the lowest processing capabilities.

Table 4.2 **Ratios between the KNL and Xeon core count**

Ratio	**KNL cores**	**Xeon cores**	**Total cores**
2:1	192	96	288
1:2	96	192	288

In the second mapping scenario, the process that plays the coordinator's role (or the master) is mapped to a Xeon core, which is the most powerful processing element in the considered system.

Comparing the results of both scenarios shows the adverse impact of reduced processing capabilities of the master on the performance of the DLS techniques using `Two_Sided_DLS`. On the contrary, the same mapping for the coordinator process did not affect the performance of the DLS techniques using `One_Sided_DLS`.

The straightforward parallelization (STATIC) is used as a baseline to assess the performance of the selected DLS techniques on the target heterogeneous computing platform. STATIC assigns $\lceil N/P \rceil$ loop iterations to each processing element. The considered implementation of STATIC follows the self-scheduling execution model where every worker obtains a single chunk of size $\lceil N/P \rceil$ loop iterations at the beginning of the application execution. By employing STATIC, the percentage of the parallel execution time of the selected applications' main loops $T_{\mathrm{p}}^{\mathrm{loop}}$ are 98% and 99.4% of the parallel execution times for PSIA and Mandelbrot, respectively. Such high percentages show that the performance of both applications is dominated by the execution time of the main loop. Hence, for the remaining results in this section, the analysis concentrates on the parallel loop execution time, $T_{\mathrm{p}}^{\mathrm{loop}}$. All experiments were repeated 20 times, and the median results are reported in all figures.

For the PSIA application, Figure 4.4(a) shows that SS, GSS, and TSS implemented with `One_Sided_DLS` outperformed their respective versions using `Two_Sided_DLS`. For instance, when the ratio of the KNL cores to the Xeon cores was 2:1, the parallel loop execution time, $T_{\mathrm{p}}^{\mathrm{loop}}$, of SS required 109 and 233 seconds with `One_Sided_DLS` and `Two_Sided_DLS`, respectively. Similarly, when the ratio was 2:1, the parallel loop execution time, $T_{\mathrm{p}}^{\mathrm{loop}}$, of GSS and TSS increased from 185 and 125 seconds to 236 and 136 seconds, respectively.

When the ratio was 1:2, the total processing capabilities of the system increased because the number of Xeon cores increased. However, the parallel loop execution time, $T_{\mathrm{p}}^{\mathrm{loop}}$, of SS, GSS, and TSS implemented using `Two_Sided_DLS` did not take the advantage of increasing the total number of Xeon cores. For instance, using `One_Sided_DLS`, changing the ratio from 2:1 to 1:2 reduced the $T_{\mathrm{p}}^{\mathrm{loop}}$ of SS from 109 to 68.5 seconds. FAC2 and WF behaved similarly using both, `One_Sided_DLS` and `Two_Sided_DLS`.

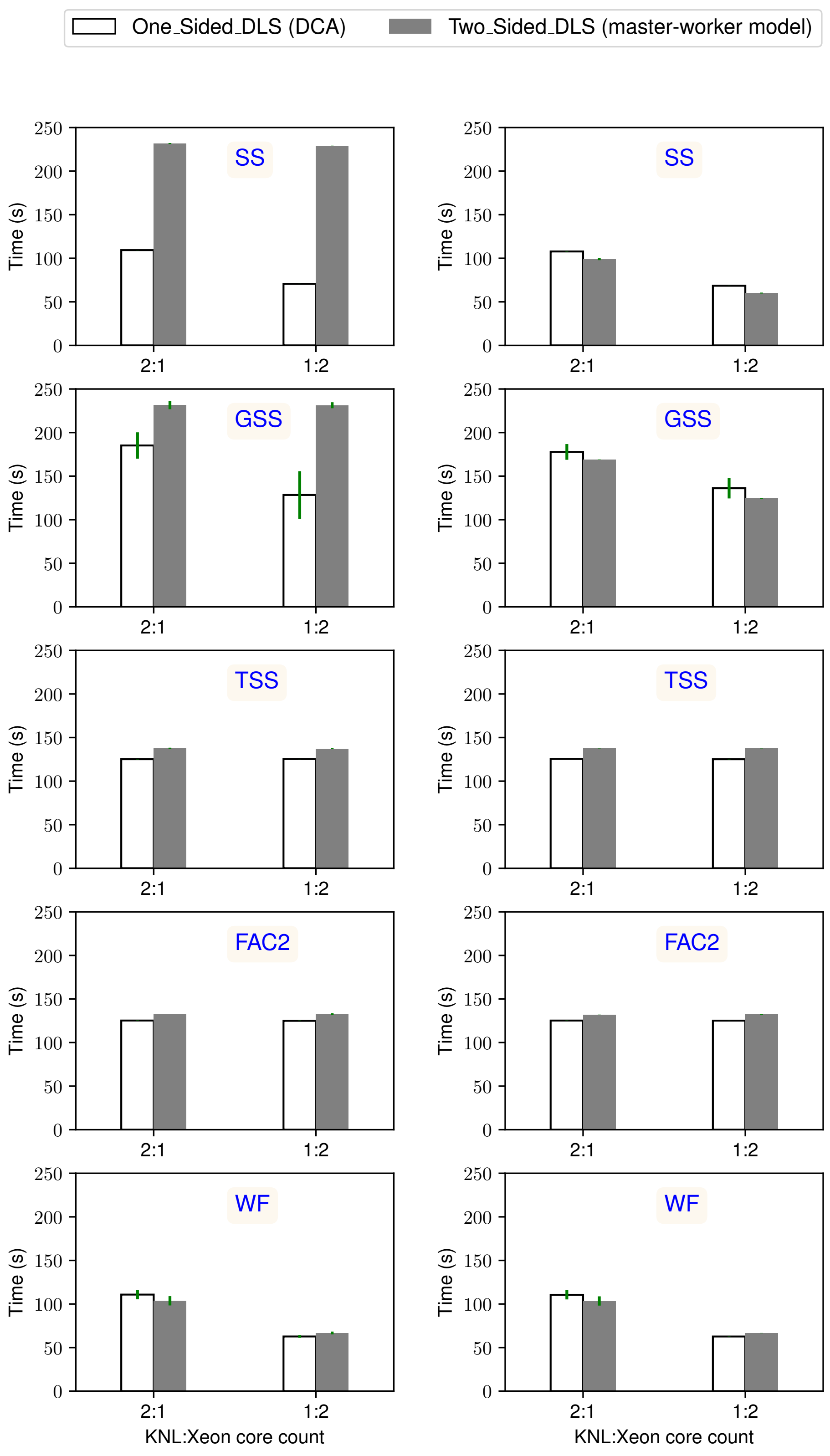
The performance degradation of the DLS techniques with `Two_Sided_DLS` is due to mapping the master to a KNL core, which has the lowest processing capabilities. Recall that in `Two_Sided_DLS`, the master is responsible for serving work requests, and therefore, it has to divide the time between serving the work requests and performing its own chunks. Therefore, if the master has a lower processing capability than the other processes, it becomes a performance bottleneck. Also, recall that `One_Sided_DLS` is designed to addresses this scenario. The coordinator process executes its own chunks and is not responsible for the chunk calculation to the other processes.

Figure 4.4(b) shows that the DLS techniques with `One_Sided_DLS` perform comparably to their versions with `Two_Sided_DLS`. For instance, using the ratio 2:1, the `One_Sided_DLS` implementation of SS, GSS, TSS, FAC2, and WF required 108, 177, 125, 125, and 110 seconds, respectively. The `Two_Sided_DLS` implementation of the same techniques required 105, 175, 135.6, 125, and 106.45 seconds, respectively. Also, using the ratio 1:2, the DLS techniques behaved similarly regardless of their implementation approach.

For the Mandelbrot application, Figure 4.5 confirms the same performance advantages of the proposed approach as for the PSIA application. The DLS techniques implemented with `One_Sided_DLS` performed equally whether the coordinator was mapped to a KNL core or a Xeon core. The performance of certain DLS techniques with `Two_Sided_DLS` degraded when the master was mapped to a KNL core compared to their performance when the master was mapped to a Xeon core.

Overall, figures 4.4 and 4.5 highlight two important observations. *First observation:* the performance variation for executing a certain experiment using the `One_Sided_DLS` approach is higher than the performance variation when executing the same experiment using the `Two_Sided_DLS` approach. The reason behind such variation is how concurrent messages are implemented at the MPI layer in `One_Sided_DLS` and `Two_Sided_DLS`. In the current work, the Intel MPI is used to implement both approaches, `One_Sided_DLS` and `Two_Sided_DLS`. Intel MPI uses the *Lock Polling* strategy to implement `MPI_Win_lock` in which the origin process repeatedly issues lock-attempt messages to the `coordinator` process until the lock is granted [ZBG16].

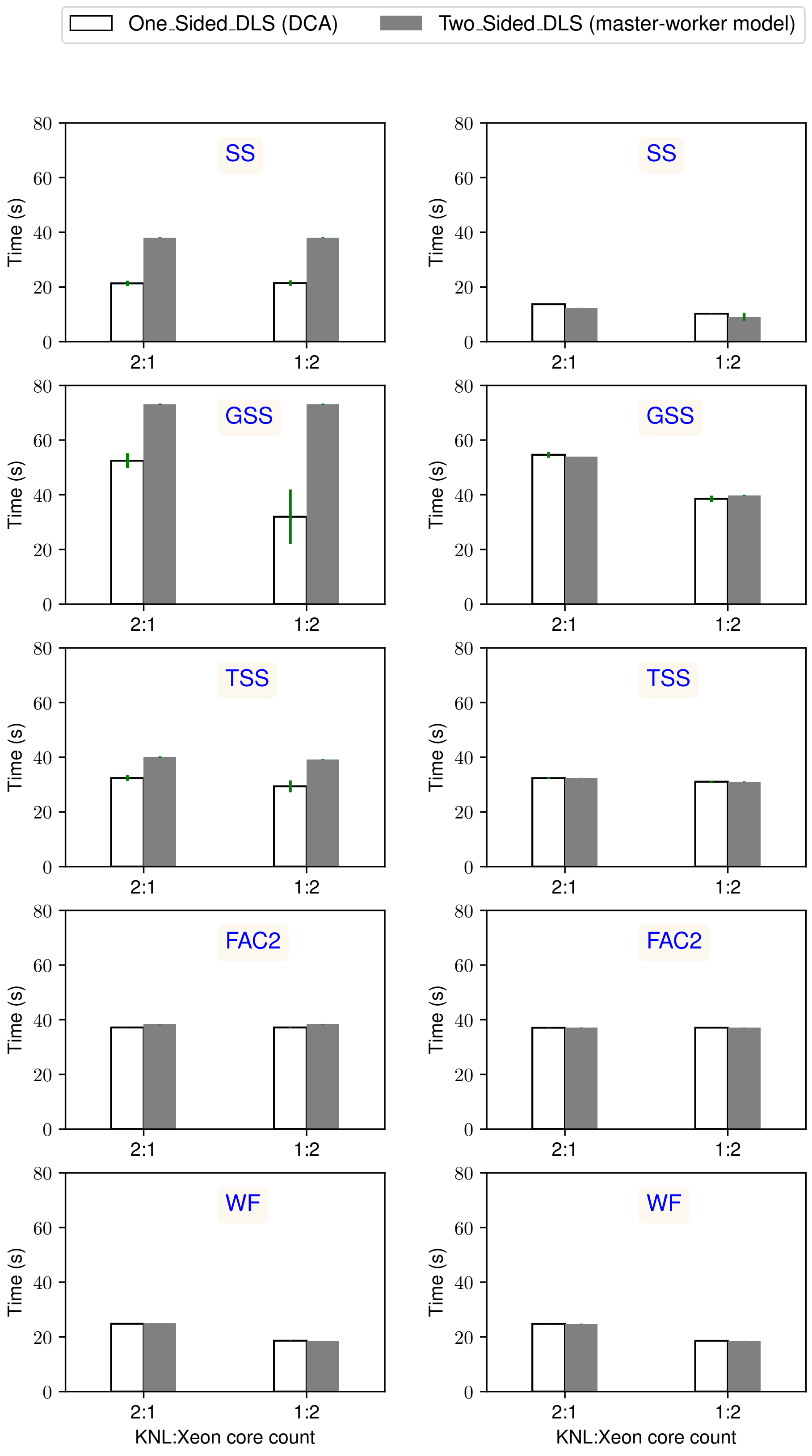
On the contrary, `Two_Sided_DLS` uses `MPI_Send`, `MPI_Recv` and `MPI_Iprobe` functions. For Intel MPI, in the case of simultaneous sends of multiple work requests to the master process, the master checks the outstanding work requests using `MPI_Iprobe` and serves them by giving a priority to the request of the process with the smallest MPI rank. The `One_Sided_DLS` has a high probability of granting the lock to different MPI processes at each trial, whereas `Two_Sided_DLS`



(a) The coordinator|master is mapped to a KNL core

(b) The coordinator|master is mapped to a Xeon core

Figure 4.4 Performance of the proposed approach vs. the existing master-worker based approach for PSIA. The x-axis represents the two ratios between the KNL cores and the Xeon cores.



(a) The coordinator|master is mapped to a KNL core

(b) The coordinator|master is mapped to a Xeon core

Figure 4.5 Performance of the proposed approach vs. the existing master-worker based approach for Mandelbrot. The x-axis represents the two ratios between the KNL cores and the Xeon cores.

always prioritizes requests from the process with the smallest MPI rank. The GSS has the largest non-linear decrement among the decrements of the selected DLS techniques. Therefore, GSS is highly-sensitive to the chunk assignment.

Second observation: FAC2 and WF exhibit a reduced sensitivity to mapping the master to a KNL or to a Xeon core. This low sensitivity could be due to the factoring-based nature of these techniques. Among all the assessed DLS techniques, FAC2 and WF assign chunks in batches, which increases the possibility for the master to have chunks of the same size as the other processing elements.

4.5 Summary

The distributed chunk calculation approach (DCA) [EC19a] can be applied to different DLS techniques. DCA requires the mathematical chunk calculation formulas to be straightforward, i.e., they must only depend on constants and the index of the last scheduling step i. Many DLS techniques have their mathematical chunk calculation formulas in a recursive format, i.e., they depend on the value of previously calculated chunk sizes [Luc92; TN93; CAB+01; PD97; Ban00]. However, we provided the mathematical transformations to change these recursive formulas into straightforward formulas.

We implemented the proposed DCA using MPI RMA and passive target synchronization mode to exploit the latest advancements in the MPI standard (MPI 3.1). When the master was not mapped to the processing element with the highest computing power, the results showed that the DLS techniques implemented with the proposed DCA outperformed their corresponding ones that were implemented with the conventional master-worker model execution model. When the master was mapped to the processing element with the highest computing power, the results showed that the DLS techniques implemented with the proposed DCA performed competitively against their corresponding ones that were implemented with the conventional master-worker model execution model. We conclude that the proposed DCA has a strong performance potential for irregular execution environment.

5

Hierarchical Distributed Chunk Calculation Approach (HDCA)

As discussed in Chapter 4, DLS techniques typically employ a master-worker execution model [CB05; BCP+05; CBR+04]. This model includes a processing entity called master. The master responsible for calculating and assigning *chunks* of loop iterations to all the other entities (workers). This model has scaling limitations [CBR+04], and therefore, the DLS techniques have evolved to employ a hierarchical master-worker execution model [CPY+05]. This model includes two levels of masters: global and local masters. The global master calculates and assigns chunks to local masters. Then, each of them becomes *responsible* for calculating and assigning sub-chunks to its group of workers. The global master and local masters may exploit different DLS techniques. The use of DLS techniques at two levels is also referred to as a *hierarchical DLS* technique.

Hybrid MPI+OpenMP is a common programming approach to implement hierarchical DLS techniques. However, it has specific performance challenges, such as the added overhead for managing two levels of parallelism using two different runtime systems. The OpenMP threads (workers) require synchronization before requesting and executing chunks, i.e., only the main thread is allowed to call MPI communication functions, such as MPI_Send and MPI_-Receive [WYL+12]. Otherwise, a complex implementation is needed to allow individual OpenMP threads to perform MPI calls.

In this chapter, we propose a novel approach for designing and developing hierarchical DLS techniques for distributed-memory systems. It extends the proposed DCA [EC19a] by allowing any group of workers that reside on a shared-memory system to form a shared work queue. The novelty of the proposed approach lies in the fact that the work queue's responsibility is shared among the workers of the group rather than one specific entity (local master).

We consider the shared-memory features offered in the MPI-3 standard. This feature enables assigning chunks to individual MPI processes in two stages. In the first stage, the fastest MPI process within a compute node obtains a chunk based on a selected DLS technique and then uses the obtained chunk to fill a local shared queue. In the second stage, the MPI processes within the same shared-memory system use a different or a similar DLS technique to obtain sub-chucks from the shared local work queue.

5.1 Hierarchical DLS Techniques

In the master-worker execution model, the number of requests that could be received by the master process is proportional to the total number of workers. For a large number of workers, the master may simultaneously receive a large number of work requests, and if the handling of the work requests is inefficient, the master becomes a performance bottleneck. To overcome such limitation, certain research efforts proposed the use of the hierarchical master-worker approach. For instance, a distributed self-scheduling scheme (HDSS) using the hierarchical master-worker model was introduced [CPY+05]. Unlike the LB-tool, HDSS dedicated the master process for handling the worker requests. The proposed scheme was implemented using MPI and its classical two-sided communication.

Another research effort discussed the adverse impact of the foreman-worker (master-worker) model [CBR+04] on DLS techniques. The authors suggested a new execution model using processor groups. The idea was to form a few groups of processors, where each group executes a specified portion of the iteration space using the master-worker model. The master process of each group has to periodically update a global master process called manager. The masters update the manager with the ratio of the remaining iterations and the available workers. When the reported ratio exceeds a certain threshold, the manager migrates workers between processor groups. Migrating workers to a certain group may result in a large number of workers within that group. This research effort [CBR+04] is similar to our proposed HDCA [EC19b] as both balance the loop execution using two levels scheduling levels. However, the present work differs by exploiting different DLS techniques at the first level, while the suggested execution model in [CBR+04] statically divides the loop iteration space among processor groups. Moreover, the present work avoids worker migration that may result in performance degradation when the cost of managing and serving requests from the migrated workers becomes relatively large. The suggested execution model was implemented using MPI, and it did not take the advantage of the shared-memory between processing elements.

Modern HPC systems are clusters of multi- and many-core systems connected via high-speed interconnection networks [CW10]. Developers often combine two different programming models to target such systems. A common approach is to use MPI [For20] for inter-node communication and OpenMP [Boa18] for programming the shared-memory systems [SB01]. In the context of DLS techniques, the hierarchical loop scheduling (HLS) [WYL+12] was one of the earliest efforts to use the MPI+OpenMP programming model. In HLS, a free worker (MPI process) requests a chunk from the master rank, which calculates and assigns the chunk based on a certain performance function [SYT07]. The workers (MPI processes) locally use OpenMP loop scheduling techniques, such as static, dynamic, and guided to execute the assigned chunk.

5.2 Maintaining Local Work Queues

The proposed approach applies two DLS techniques at the intra- and inter-node levels as follows. One MPI process creates a global shared-memory region called the global work queue. This global queue stores information regarding the latest scheduling step and the total scheduled loop iterations [EC19a]. Using `MPI_Win_allocate_shared`, the MPI processes within one compute node create another shared-memory region called the *local work queue*. This local queue stores information regarding the latest scheduling step and the total scheduled loop iterations by the MPI processes within that physical node. Whenever an MPI process becomes free, it obtains a sub-chunk from the local work queue. If there are no sub-chunks, the MPI process tries to obtain a chunk from the global work queue and fills the empty local work queue. In the proposed HDCA, the MPI processes do not wait for each other to fill the local work queue. The responsibility of obtaining work is not assigned to a specific MPI process, as the fastest MPI process always takes this responsibility. Figure 5.1 illustrates the proposed HDCA using the MPI shared memory feature, known as MPI+MPI approach [HDB+13].

Unlike existing MPI+OpenMP implementations of DLS techniques, the proposed approach avoids the implicit synchronization that is required at the end chunks' execution. Figure 5.2 illustrates the undesired implicit thread synchronization when using the MPI+OpenMP approach. OpenMP Thread 1 finished

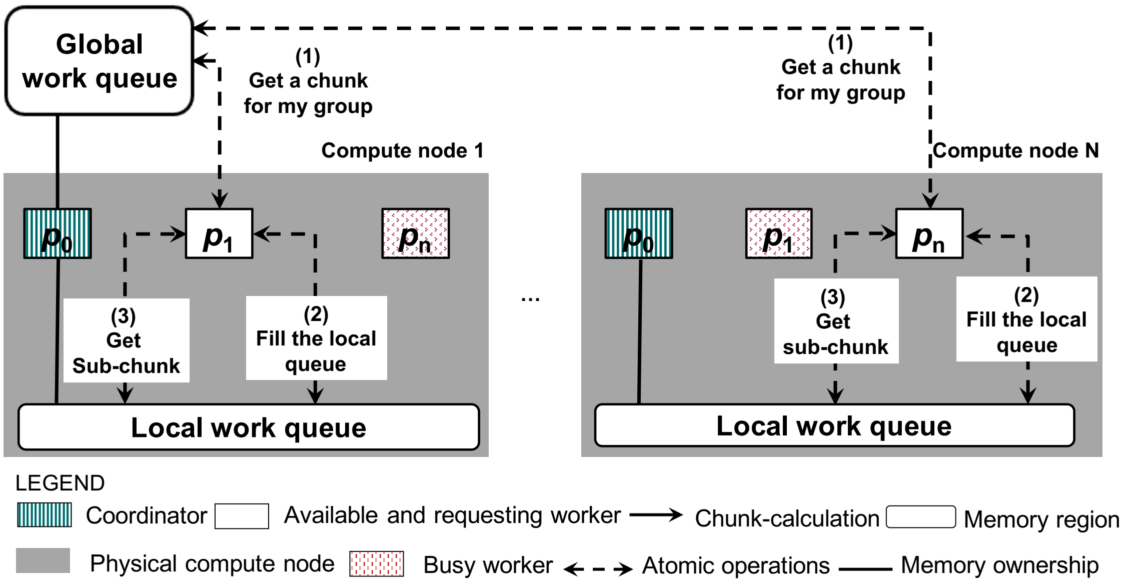


Figure 5.1 **The proposed hierarchical DLS techniques** using the MPI+MPI approach. Only one coordinator exposes a window of its memory as the global work queue (see Section 4.3).

its sub-chunk earlier than the rest of the threads. However, it has to wait for the slowest OpenMP Thread (OpenMP Thread 7). At the second chunk, the same scenario was repeated when OpenMP Threads 6 and 7 finished earlier than the rest.

Figure 5.3 shows the desired optimal execution scenario at the shared-memory level. Worker 1 finished earlier than the rest; however, it immediately obtained a new chunk to fill the local work queue, and then it obtained a sub-chunk for itself. Once any other worker finished its sub-chunk, it could directly obtain a sub-chunk from the most recent chunk obtained by Worker 1. In Figure 5.3, the parallel time to execute the loop t'_{end} is less than t_{end} in Figure 5.2.

Compared to the proposed approach, one could state that the main issue of the MPI+OpenMP approach is the *implicit barrier* at the end of executing each chunk of a loop iteration. Such an issue could be solved using the *nowait* clause that allows OpenMP threads to continue their execution when there are no more loop iterations to execute. However, the use of the *nowait* clause requires all OpenMP threads to initiate `MPI_Send` and `MPI_Recv` calls, i.e., the fastest OpenMP thread may differ from one chunk to another. Therefore, the implementation would require many synchronization statements to guarantee the exclusive request of new chunks for only one thread at a time. This leads to more complicated codes, which are hard to tune and maintain.

5.3 Performance Evaluation and Discussion

Hierarchical DLS techniques can be implemented using either the hierarchical master-worker [CPY+05] or the distributed chunk calculation model [EC19a]. We evaluated the use of two different implementations, MPI+OpenMP and MPI+MPI, to complement *the distributed chunk calculation approach*.

The MPI+OpenMP implementation complements the distributed chunk-calculation approach by the use of OpenMP at the shared-memory level. It maps one MPI process per each compute node. The mapped MPI processes communicate and cooperate to obtain chunks using one of the following DLS techniques: STATIC, SS, GSS, TSS, and FAC2. Every MPI process uses the OpenMP runtime to create a number of threads equal to the number of its computing cores. The threads use the OpenMP loop scheduling techniques (static, dynamic, and guided) to

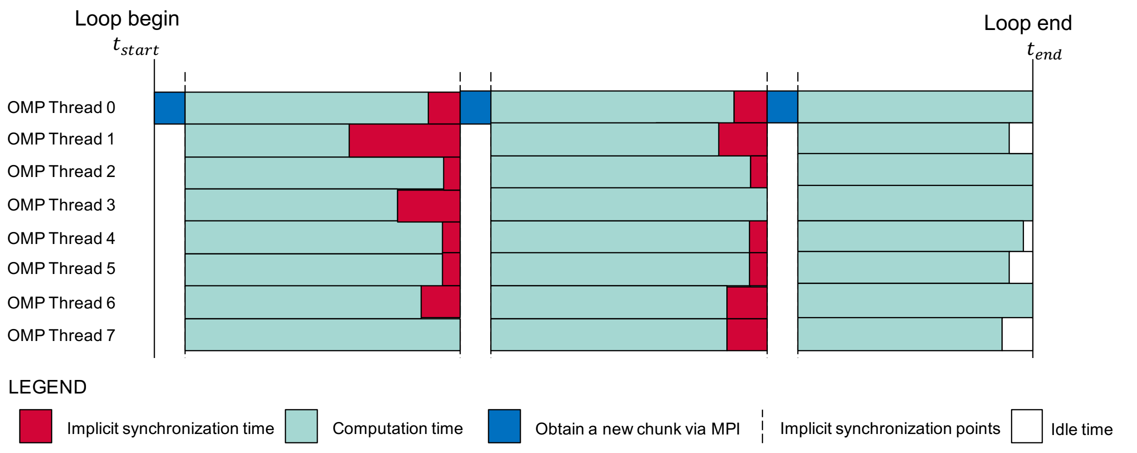


Figure 5.2 The undesired synchronization with the MPI+OpenMP implementation approach at the shared-memory level.

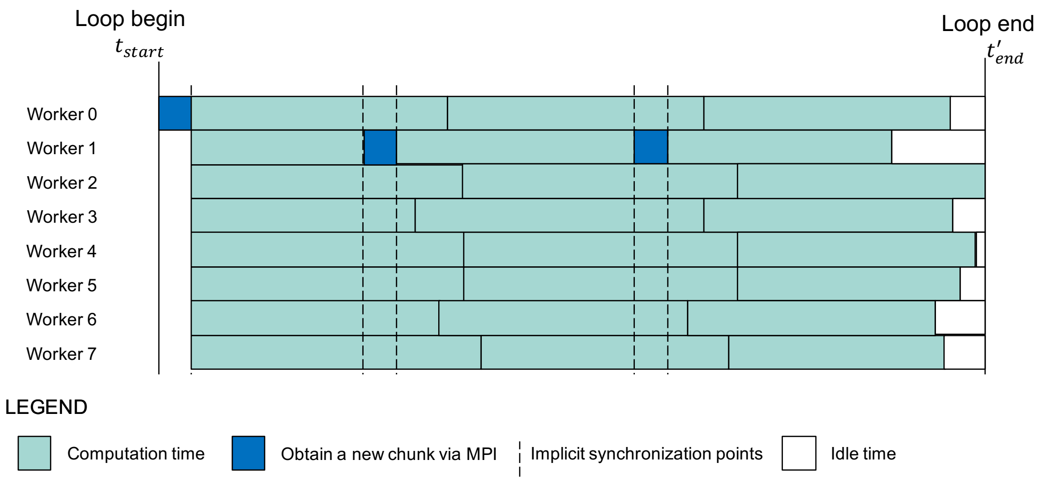


Figure 5.3 Illustration of an ideal execution scenario at the shared-memory level.

Table 5.1 Mapping between the DLS techniques and the OpenMP schedule clause options

DLS technique	OpenMP schedule clause
STATIC	schedule(static)
SS	schedule(dynamic,1)
GSS	schedule(guided,1)

execute the chunks obtained from their (owner) MPI process. The MPI+MPI implementation complements the distributed chunk calculation approach using MPI shared-memory capabilities, as explained in Section 5.2, i.e., it forms shared local queues at the compute node level (see Figure 5.1).

Similar to Chapter 4, we used PSIA and Mandelbrot to evaluate the proposed HDCA and used miniHPC as the target platform (see Section 4.4).

The OpenMP standard currently supports three loop scheduling techniques: static, dynamic, and guided (see Table 5.1). More loop scheduling techniques were implemented in an OpenMP runtime library called LaPeSD-libGOMP [CIB18]. However, for accurate performance measurements, we wanted to use the most optimized software installed on miniHPC. Given that miniHPC (the target system) is an Intel-based cluster, the Intel software stack was selected, and therefore, scheduling experiments that have TSS and FAC2 at the shared-memory level were only performed using the proposed MPI+MPI approach. The use of LaPeSD-libGOMP, instead of Intel OpenMP runtime library, enables more DLS techniques, and it is planned as future work.

In this section, the X+Y notation is used to represent scheduling combinations, where X is a DLS technique used at the inter-node level and Y is a DLS technique used to at the intra-node level. X and Y refer to only one DLS technique.

Figures 5.4 to 5.7 show the performance of executing Mandelbrot and PSIA with two levels of DLS techniques.

Figure 5.4 shows the first combination of DLS techniques where STATIC is used to schedule the workload across multiple compute nodes. An important observation is that when SS is selected to schedule the workload within one computing node, the proposed MPI+MPI approach has the poorest performance compared to the MPI+OpenMP. The reason is due to the use of `MPI_Win_-lock` and `MPI_Win_sync`. These functions provide exclusive access to the local work queue (see Figure 5.1), and consequently, maintain the work queue. The `MPI_Win_lock` uses a lock polling technique where an MPI process repeatedly issues lock-attempt messages until the lock is granted [ZBG16]. Consequently,

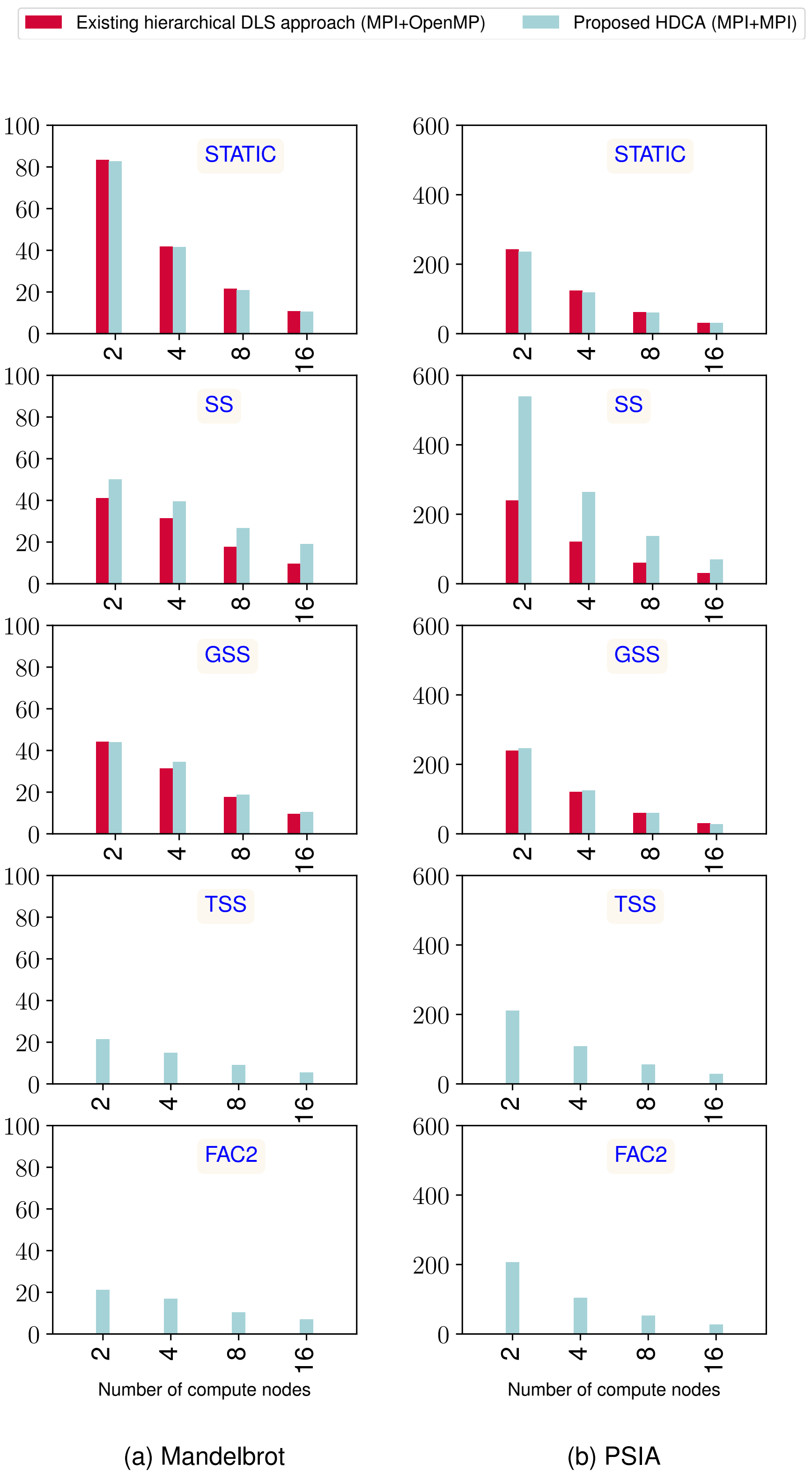


Figure 5.4 Parallel execution time of the main loop of both applications, Mandelbrot and PSIA. For the MPI+OpenMP approach, each worker is an OpenMP thread, and the total MPI processes per one compute node is one process. For the MPI+MPI approach, each worker is an MPI process, and the total MPI processes per one compute node is 16 processes. STATIC is the first level of scheduling (inter-node scheduling).

the number of lock-attempt messages increases when multiple processes try to acquire the same lock simultaneously, and more overhead is introduced.

Another observation is that all hierarchical DLS techniques, except SS, implemented with the proposed MPI+MPI approach have the same performance compared to their counterparts implemented using the MPI+OpenMP approach. The reason is that using STATIC at the inter-node level means there is only one scheduling round at that level. However, achieving the same results also indicates that the proposed approach did not introduce significant overhead to the DLS techniques.

Figure 5.5 shows the second combination of DLS techniques where GSS is used to schedule the workload across the compute nodes. For both applications, the proposed MPI+MPI approach outperformed the MPI+OpenMP. The results of the GSS+STATIC combination show the advantage of the proposed approach, where avoiding the unnecessary synchronization between the workers (OpenMP threads) has a significant adverse impact. For instance, in Mandelbrot and using the proposed approach, the parallel execution times of the GSS+STATIC combination were 19.6 and 3.1 seconds on the smallest and the largest system sizes, respectively. The same scheduling combination GSS+STATIC using MPI+OpenMP took 61.5 and 4.5 seconds on the smallest and the largest system sizes, respectively. In PSIA, the performance trend was repeated, i.e., the GSS+STATIC using the proposed MPI+MPI approach outperformed its counterpart implemented using the MPI+OpenMP approach. For instance, on the smallest systems size, the parallel execution times were 233 and 245 seconds using the proposed MPI+MPI approach and the MPI+OpenMP approach, respectively. However, the two approaches had the same performance when executing on the largest system size. The reason is the decreased load imbalance in PSIA compared to that in Mandelbrot. For the GSS+GSS combination, the DLS techniques implemented using the proposed MPI+MPI approach also outperformed their counterparts implemented using the MPI+OpenMP approach.

As discussed earlier in this Section, we decided to use the Intel software stack. Therefore, it was not possible to perform the remaining combinations: GSS+TSS and GSS+FAC2 using MPI+OpenMP, i.e., the Intel OpenMP runtime library only supports the following loop scheduling techniques: static, dynamic, and guided.

Figures 5.6 and 5.7 show the third and the fourth combinations of the DLS techniques where TSS and FAC2 are used to schedule the workload across multiple compute nodes, respectively. Similar to Figure 5.5, the proposed approach

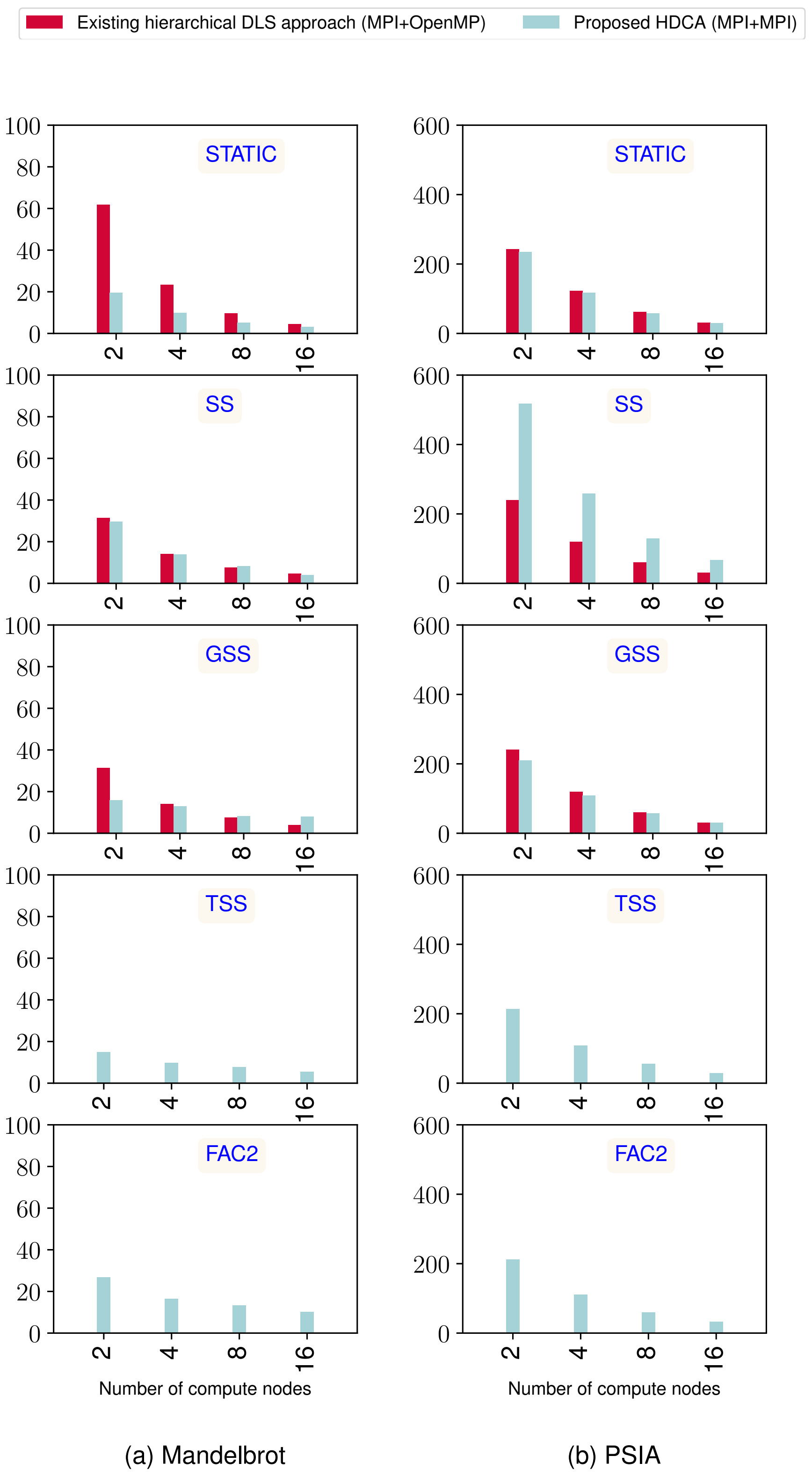


Figure 5.5 Parallel execution time of the main loop of both applications, Mandelbrot and PSIA. For the MPI+OpenMP approach, each worker is an OpenMP thread, and the total MPI processes per one compute node is one process. For the MPI+MPI approach, each worker is an MPI process, and the total MPI processes per one compute node is 16 processes. GSS is the first level of scheduling (inter-node scheduling).

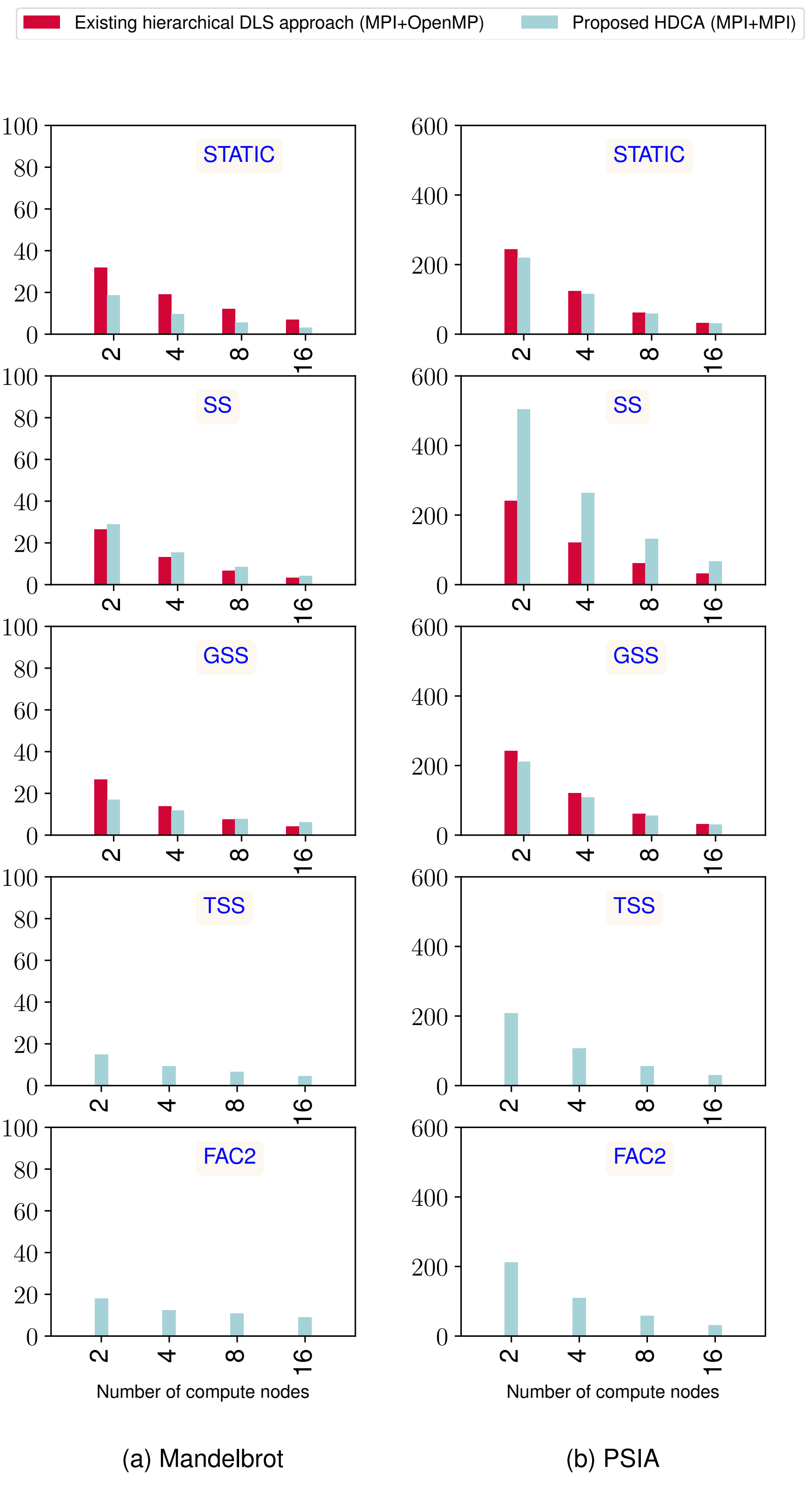


Figure 5.6 Parallel execution time of the main loop of both applications, Mandelbrot and PSIA. For the MPI+OpenMP approach, each worker is an OpenMP thread, and the total MPI processes per one compute node is one process. For the MPI+MPI approach, each worker is an MPI process, and the total MPI processes per one compute node is 16 processes. TSS is the first level of scheduling (inter-node scheduling).

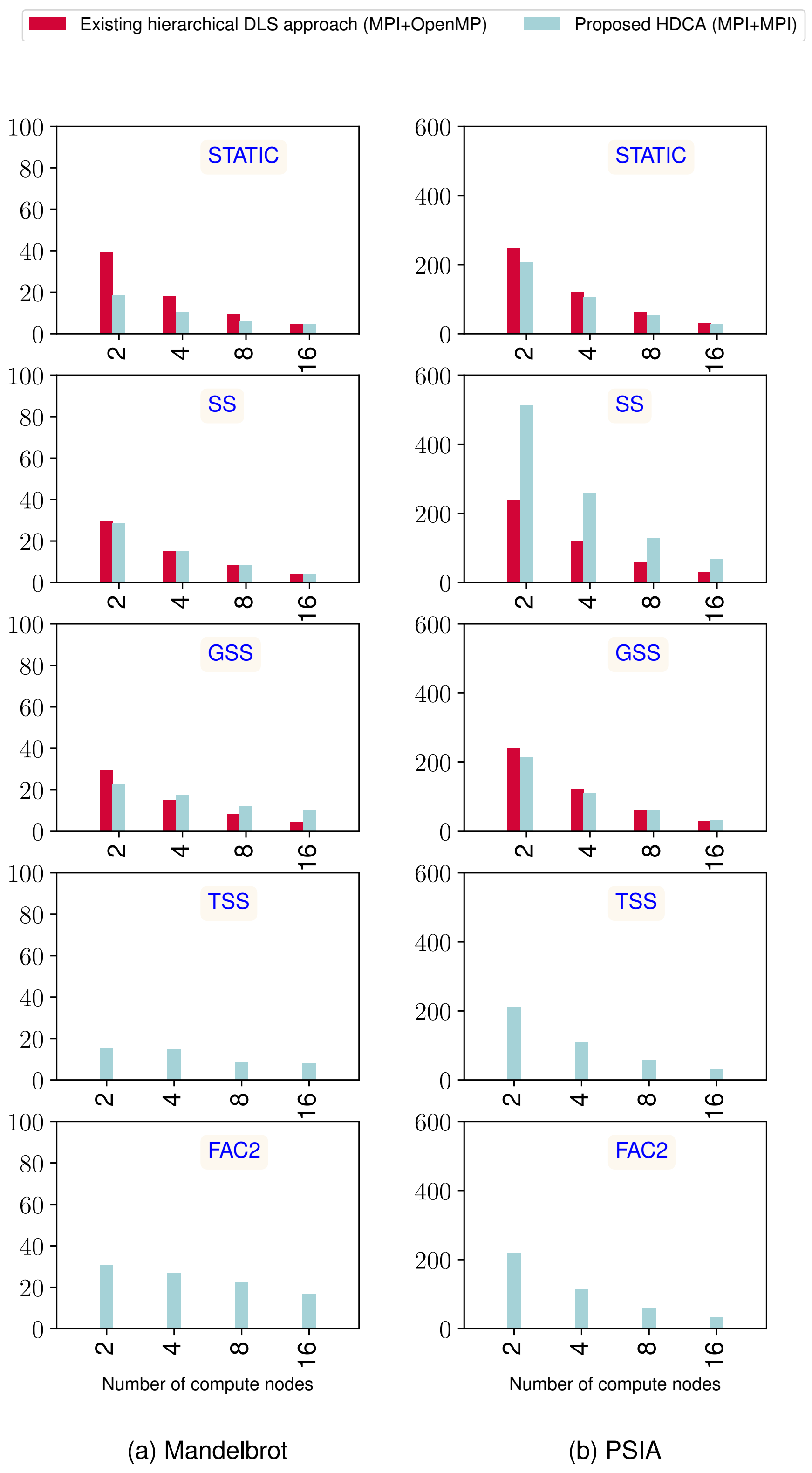


Figure 5.7 Parallel execution time of the main loop of both applications, Mandelbrot and PSIA. For the MPI+OpenMP approach, each worker is an OpenMP thread, and the total MPI processes per one compute node is one process. For the MPI+MPI approach, each worker is an MPI process, and the total MPI processes per one compute node is 16 processes. FAC2 is the first level of scheduling (inter-node scheduling).

significantly outperformed the MPI+OpenMP approach when STATIC is selected for scheduling the computational workload within one compute node. For the rest of the scheduling combinations, both approaches have the same performance. The only exception is when applying SS at the shared-memory level. The proposed approach has the worst performance compared to the MPI+OpenMP approach. The reason is that SS achieves the maximum load balance, and most of the workers (OpenMP threads) finish at the same time. This scenario will avoid the long synchronization time before getting new chunks.

The last observation is related to the performance of the PSIA when applying any combination that has the SS using the proposed approach. PSIA has less load imbalance than Mandelbrot, and the proposed approach has a significant overhead when employing SS. Consequently, the adverse impact of the large associated scheduling overhead of SS is more visible in PSIA than Mandelbrot.

5.4 Summary

The implementation of hierarchical DLS techniques is essential to enable scalable application performance. Because of the centralized work queue (loop iterations), efficient implementations of hierarchical DLS techniques may be better than non-hierarchical ones. When STATIC is used for the intra-node level scheduling, the proposed HDCA that employs the MPI+MPI approach outperformed the one that uses the hybrid MPI+OpenMP approach. This highlights and confirms the capability of our proposed HDCA to eliminate the unrequired synchronization at the intra-node level. On the contrary, the MPI+MPI approach shows a limited performance when many MPI processes on the same shared-memory system try to access the local work queue simultaneously. The important observation of the present work is that the scheduling overhead associated with using MPI shared-memory to implement DLS techniques is higher than OpenMP. Therefore, the use of the MPI+MPI approach is only recommended for developing hierarchical DLS techniques when its associated overhead is less than the synchronization overhead associated with the use of OpenMP.

6

Resourceful Coordination Approach (RCA) for Multilevel Scheduling

The multilevel Scheduling (MLS) refers to exchanging scheduling information between all scheduling levels, such as batch, application, and thread level. MLS helps to refine scheduling decisions at a certain level based on the available information regarding the current scheduling workload at other levels. We propose a resourceful coordination approach (RCA) that enables the cooperation between, currently independent, batch- and application-level schedulers. RCA enables application schedulers to share their allocated but idle computing resources with other applications through the batch system. With enabling this coordination, RCA avoids resource shrinking operations and associated performance penalties that are typical of dynamic resource and job management systems. To evaluate RCA, we bridged a Slurm-based simulator(at the batch-level) and a SimGrid-based simulator flowing the same principle of the two-level simulation approach that we presented in Chapter 3.

6.1 Coordination Between ALS and BLS

The resourceful coordination approach (RCA) requires information exchange between batch and application schedulers: (1) From the application schedulers to the batch scheduler. The application schedulers report the status of their free computing resources and the remaining amount of work. (2) From the batch scheduler to the application schedulers. The batch scheduler can take advantage of knowing the execution history of certain applications and can benefit from additional hints that the user may provide, such as expected applications' execution time, communication/computation ratio, etc. The information exchange allows the batch scheduler to reuse computing resources as soon as they become idle, and there are no more tasks from the job that can be assigned to them. User hints allow the batch scheduler to identify applications that experience minimal performance degradation when they exclude a specific number of allocated resources. The exclusion means that the application schedulers will not schedule further tasks on the excluded resource. This exclusion differs from shrinking the resource allocation of malleable jobs. In RCA, the application still owns the temporarily relinquished computing resource, but it allows other applications to use it. RCA allows application schedulers to accept or reject resource exclusion requests from the batch scheduler.

Figure 6.1 illustrates three executing applications (App1, App2, App3) and two queued applications (App4 and App5). First-come-first-serve (FCFS) is employed at the batch-level to schedule the five jobs. App4 has a higher priority than App5. App4 requests four computing resources, and only two resources

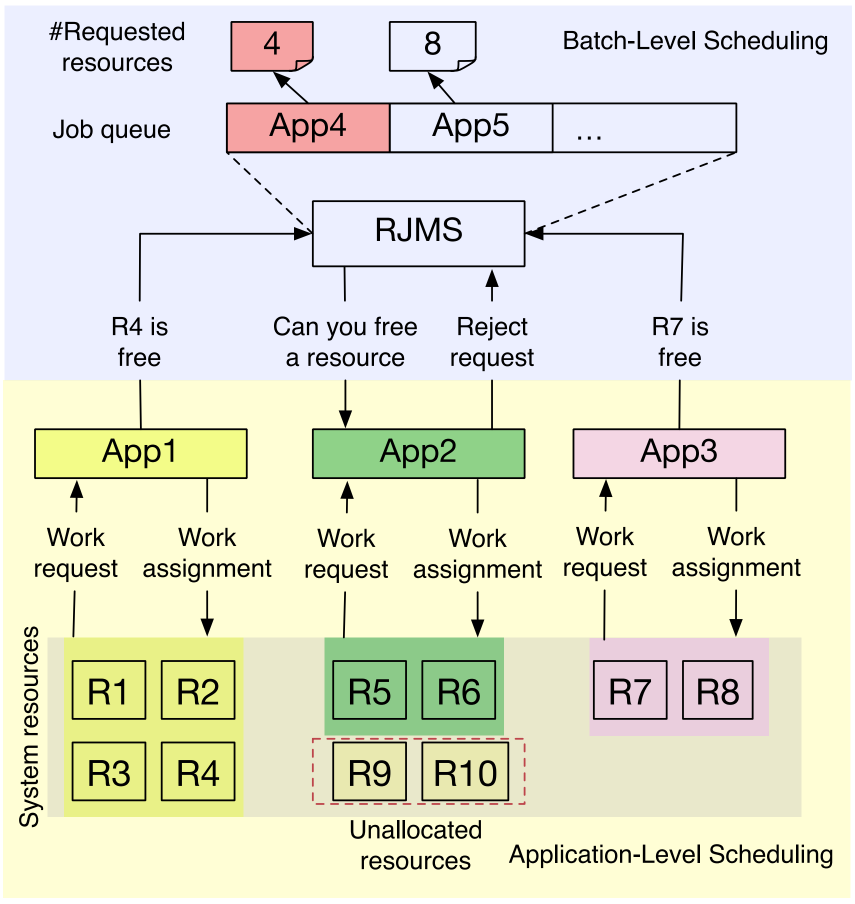


Figure 6.1 The proposed resourceful coordination approach (RCA). Applications (e.g. App1) cooperate with other applications (e.g. App4) by yielding idle resources (e.g. R4) through the batch system.

are available: R9 and R10. However, the batch system cannot start App4 due to insufficient free resources. Assuming that BF [FW98] is enabled, the batch system launches any job from the queue that requests at most two resources. In the example, App5 requests eight resources, and no other applications exist in the queue. Existing batch scheduling systems would leave App4 waiting in the queue and R9 and R10 idle until one of the executing applications finish. In contrast, in RCA, the batch system receives information form application schedulers during applications' execution. App1 and App3 report that R4 and R7 became free. R4 and R7 can be reassigned to other applications through the batch system. The information from App1 and App3 may be reported at different times. Once the batch system receives these two reports, and if App4 is still in the queue, the batch system can assign R4, R9, R10, and R7 to App4, which can then begin execution.

The batch system can identify (based on applications' execution history) applications that can relinquish specific resources without performance degradation. In the example illustrated in Figure 6.1, the batch system identified App2 as such an application. The application scheduler of App2 rejected the request and did not release any resources. In RCA, the batch scheduler does not control the ALS decisions. Application schedulers can reject the release of resource requests. Accepting or rejecting batch requests can be seen as a higher level of cooperation than reporting resource idle time that can be enabled or disabled based on users' preferences. Moreover, the batch system leaves the decision regarding which resource to be freed to the application scheduler. RCA aims to separate the concerns between BLS and ALS. BLS tries to provide the required number of resources to waiting jobs, while ALS decides which resource(s) is (are) ready to be released right away.

6.2 RCA Applied to a BLS Simulator and an ALS Simulator

Design details: At batch-level scheduling, the current work employs the widely used Slurm simulator [Luc11]. The current work extends and modifies one of the latest versions of the Slurm simulator [SDI+18]. Listing 6.1 shows the modifications required to support RCA.

In Algorithm 6.1, Line 2 shows the new code that has been added to allow the Slurm simulator to read ALS information, such as the ALS scheduling method. Line 3 represents the modified code that extends the Slurm simula-

Algorithm 6.1 Batch-level scheduling

```
slurm_sim_controller(){
1  read_slurm_sim_configuration(sim_config);
2  extract_als_configuration(als_config);
3  sim_read_job_trace(trace_head);
4  synchronize_with_app_simulator(als_config);
5  while True do
6      run_scheduling_round(); /*Listing 2*/
7      update_SimGrid_simulation_clock());
8      if no_jobs_to_submit() then
9          if no_running_apps() then
10             collect_simulation_trace();
11             end_app_simulator();
12             exit();
13     sim_submit_jobs();
14     sim_process_finished_jobs();
15     sim_cancel_jobs();
16     sim_schedule();
17     sim_run_priority_decay();
18     schdeule_plugin_run_once();
19     sim_sinfo();
20     sim_squeue();
} /*original, new, modified code*/
```

tor to accept workloads in the standard workload format (SWF) [FTK14]. This modification enables the simulation of various workloads from production HPC systems that are available in the public workload archive [FTK14]. Lines 4 to 12 represent a newly added code that connects the SimGrid-based simulator with the Slurm simulator. Hence, the SimGrid-based simulator works as an *internal clock* for the Slurm simulator. SimGrid simulations are event-based simulations, and consequently, the simulation time is only advanced by the occurrence of simulation events. In our approach, the simulation time will only be advanced when scheduling events happen at either the batch- or application-level. Lines 13 to 16 represent certain functions of the original Slurm simulator [SDI+18] that we extended to produce or consume scheduling events of the SimGrid-based simulator.

The communication between the two simulators employs a shared data structure called *all_apps*, which holds all information about jobs' execution (Line 1 in Algorithm 6.2). Scheduling events, such as starting a job on a specific set of resources, are produced by the Slurm-based simulator and stored in the *all_apps* data structure. Also, scheduling events, such as job completion, are produced by the SimGrid-based simulator and are stored in the *all_apps* data structure. Each simulator *consumes* the events *produced* by the other simulator.

For ALS, the present work designs and extends an accurate SimGrid-based simulator [MEC+20] to simulate applications' executions with various DLS techniques. This simulator *simultaneously* simulates the execution of several applications executing on the same HPC platform. The intention behind this difference is to let the simulator account for application interference. Earlier research efforts [EMC17b; MEC+20] focused on studying applications' performance under various scheduling techniques. In contrast, the current work relaxes the assumption of applications executing on separate sets of resources during their entire execution, thereby increasing the realism of the simulation.

Algorithm 6.2 shows a single *scheduling round* of our extended SimGrid-based simulator. A scheduling round refers to a scanning procedure where all simulated applications and their assigned resources are examined to identify the idle resources and to self-schedule the remaining work. Algorithm 6.2 illustrates the logic of the function *run_scheduling_round()* of Algorithm 6.1.

Algorithm 6.2 Application-level scheduling

```
run_scheduling_round(){
1  foreach app in all_apps do
2      unscheduled = check_unscheduled_tasks(app);
3      hosts = get_free_hosts(app);
4      foreach host in hosts do
5          if unscheduled >0 then
6              scheduling_method= schedudling_method(app);
7              tasks=chunk_size(app, scheduling_method);
8              schedule_tasks(host, tasks);
9              unscheduled = unscheduled - tasks;
10             continue; /*Go to Line 4*/
11         release_host(host,app);
} /*scheduling round in SimGrid*/
```

For native Slurm RJMS, the BLS-ALS communication can be implemented via remote procedure call (RPC) similar to the communication between the Slurm daemons (slurmctl and slurmd). The Slurm daemons periodically exchange messages to monitor resources' status. These small messages have minimal impact on the performance of the running application. The BLS-ALS communications are not periodic, and they are occasionally sent. For instance, BLS-ALS communication messages are sent when the originating entity is not executing any workload. The BLS-ALS communication messages in that sense will not degrade applications' performance.

6.3 Performance Evaluation and Discussion

Experimental design: In all experiments reported herein, a simulated platform with 256 compute hosts is used. A fully-connected network topology is used to connect all hosts. The network fabric is assumed InfiniBand with a link bandwidth and latency of 50 Gbps and 500 ns, respectively.

The effective system performance (ESP) [WOK+00b; WOK+00a] benchmark is used to evaluate the usefulness of the proposed approach. ESP describes batch workloads that can be used to assess batch systems' performance. The description includes guidelines regarding the total number of jobs, estimated job execution time, number of requested resources per job, and job arrival times [WOK+00b; WOK+00a; PNR+15; GH12]. Table 6.1 illustrates the characteristics of the ESP system benchmark, which consists of 230 jobs divided into 14 job categories. Jobs of different categories require various numbers of computing resources, from 3.12% to 100% of the available computing resources. For instance, one job in Category A requires eight computing resources (3.12% of the entire system), while one job in Category Z requires 256 computing resources (the entire system).

Table 6.1 Characteristics of the two implemented versions of the ESP system benchmark: ESP-PSIA and ESP-Mandelbrot.

Category ID	Requested Hosts	Total Jobs	ESP-PSIA #images	ESP-Mandelbrot #iterations
A	8	75	32 K	0.635 M
B	16	9	76.5 K	1.2 M
C	128	3	800 K	15 M
D	64	3	582 K	8.5 M
E	128	3	595 K	8.8 M
F	16	9	440 K	6.5 M
G	32	6	635 K	1 M
H	40	6	630 K	10 M
I	8	24	170 K	3.35 M
J	16	24	174.5 K	2.75 M
K	24	15	172.6 K	2.85 M
L	32	36	172.5 K	2.725 M
M	64	15	176.5 K	2.65 M
Z	256	2	375 K	5.25 M

Another essential factor in the ESP system benchmark is the job arrival time. The authors of the ESP system benchmark suggested a job arrival scheme in which Category Z jobs arrive in such a way that they divide the arrival timeline into three parts (see Figure 6.2) [WOK+00b; WOK+00a]. In this way, the jobs arrive during batch execution. This arrival pattern prevents the batch scheduler from knowing the entire workload before the execution, which would be unrealistic.

ESP jobs are synthetic and can be represented by various applications [WOK+00b; WOK+00a]. In the present work, we exemplify the ESP system benchmark with the PSIA [EFM+16; EMC17a] and Mandelbrot [Man80] applications (See Section 4.4). We generate and use two different workloads of the ESP system benchmark called *ESP-PSIA* and *ESP-Mandelbrot*. PSIA and Mandelbrot are chosen to represent two extremes of interest for testing our approach: a balanced execution (PSIA) and a high load imbalanced execution (Mandelbrot). Moreover, individual research efforts [MEC+20; MEC+18] proposed an accurate and verified representation of the computational workload of both applications in SimGrid.

PSIA [EFM+16; EMC17a] is a computationally-intensive application from computer vision that consists of a large loop that dominates the entire execution. Loop iterations in PSIA have different computational loads and require efficient loop scheduling to achieve a balanced execution of these iterations. Various dynamic scheduling techniques can achieve a balanced execution for PSIA. Consequently, there are few differences in computing resource finishing times that execute the PSIA application. Such times are important in this work as they represent idle resources that can be relinquished. The Mandelbrot set is a

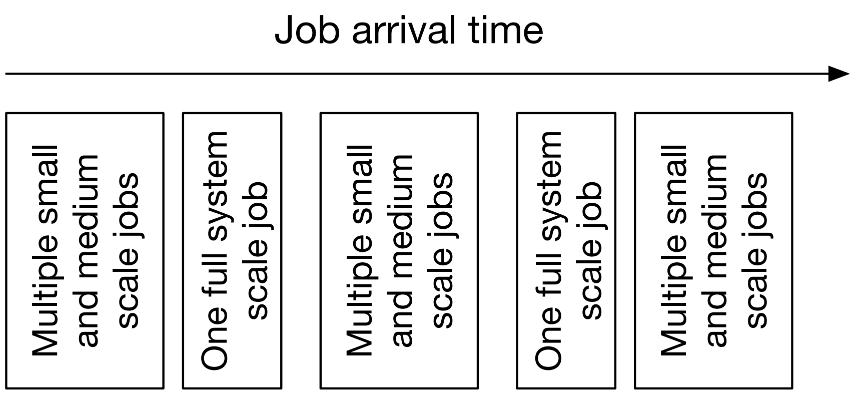

Job arrival time
Multiple small and medium scale jobs
One full system scale job
Multiple small and medium scale jobs
One full system scale job
Multiple small and medium scale jobs

Figure 6.2 ESP job arrival scheme (adapted from [WOK+00b; WOK+00a]). Full system scale jobs refer to jobs from Category Z. Multiple and small scale jobs refer to jobs from other categories.

well-known mathematical kernel. It contains a set of irregular and independent loops and has been used to evaluate DLS techniques in the literature [SYT07; CAB+01].

The last two columns of Table 6.1 show the characteristics of the two versions of the ESP system benchmark workload that contain PSIA and Mandelbrot jobs. Various input parameters control the execution of PSIA and Mandelbrot [EMC17a; Man80]. One parameter for each application is changed to let the applications meet the ESP's job execution category [PNR+15]. For PSIA, *#images* indicates the total number of generated spin-images. For Mandelbrot, *#iterations* indicates the maximum number of iterations per pixel. The two parameters are chosen because they had a linear relation to the application execution time. Therefore, it is more precise to estimate their initial values that meet the job execution category.

Figure 6.3 shows the load imbalance profile of the two versions of the ESP system benchmark: ESP-PSIA and ESP-Mandelbrot. The metric *max/mean* denotes the ratio between the finishing time of the latest computing resource and the average finishing time of all computing resources that execute a certain job. When the ratio *max/mean* of a certain job is very close to one, the job has a balanced load execution on its allocated resources.

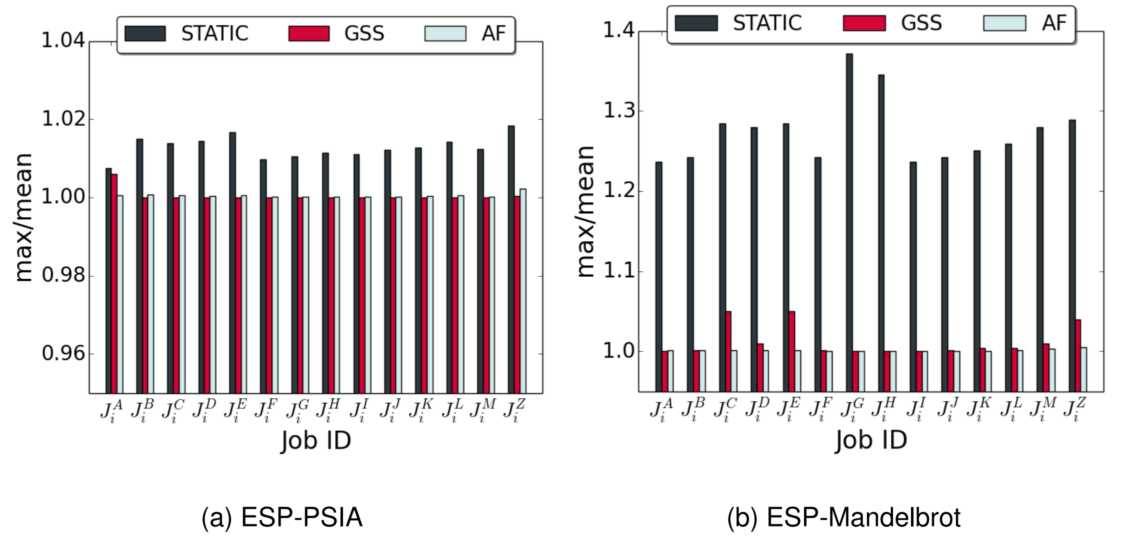


Figure 6.3 Load imbalance profile of the jobs within the ESP-PSIA and ESP-Mandelbrot workloads. The ratio max/mean indicates the degree of balanced execution for each job J_i^x, where x is a job category (see Table 6.1) and i ranges according to the size of each job category. Values that are close to 1 denote a balanced execution.

In Figure 6.3(a), for all job categories, the values of max/mean are close to one. This reflects the balanced load execution of the PSIA. For ESP-PSIA, AF achieved the most balanced execution compared to GSS and STATIC (see Figure 6.3(a)). AF also achieved a fully-balanced execution for ESP-Mandelbrot compared to STATIC and GSS (see Figure 6.3(b)). The results in Figure 6.3 indicate less idle resources when executing ESP-PSIA than when executing ESP-Mandelbrot. Therefore, the ESP-PSIA workload represents a challenging case for the proposed approach, i.e., computing resources have short idle times that can only briefly be exploited by other applications.

Experimental Evaluation and Discussion: System utilization (SU) is an important metric that indicates the efficiency of batch scheduling techniques. We calculate system utilization as shown in Eq. 6.1, where T_k is the time that ac computing resource k spent executing jobs, P is the total number the computing resources, and T_{batch} denotes the system makespan measured as the total execution time of *the entire batch*, i.e., $T_{batch} = T_j - T_i$, where T_i is the time when the first job starts execution and T_j is the time when the last job in the batch completes execution. System utilization ranges from 0% to 100%. Higher values of system utilization indicate better system performance.

$$SU = \frac{\sum_{k=0}^{P-1} T_k}{P * T_{batch}} * 100 \tag{6.1}$$

Figure 6.4 shows the system utilization over batch execution time for the ESP-PSIA *with* and *without* the proposed approach. When our resourceful scheduling approach is not enabled in the simulation, the makespan of the ESP-PSIA using STATIC, GSS, and AF is 13,000, 12,875, and 12,875 seconds, respectively (see Figure 6.4(a)). This corresponds to the increase in the system utilization in Figure 6.4(a); the GSS (blue) and AF (black) curves are slightly higher than the STATIC (red) curve.

Figure 6.4(b) shows that the system makespan improved with our resourceful scheduling approach. For instance, the system makespan for ESP-PSIA with STATIC is 12,965 instead of 13,000 seconds. For GSS and AF the improvement is not impressive. As discussed earlier in this section, ESP-PSIA is an extreme case of a highly balanced execution. This means that the differences in resource finishing times that execute the PSIA application are minimal. In this case, enabling RCA will have limited advantages. One can still notice that the gap in system utilization when using STATIC, GSS, and FAC with RCA (see Figure 6.4(b)) is slightly smaller than the gap in Figure 6.4(a)(without RCA).

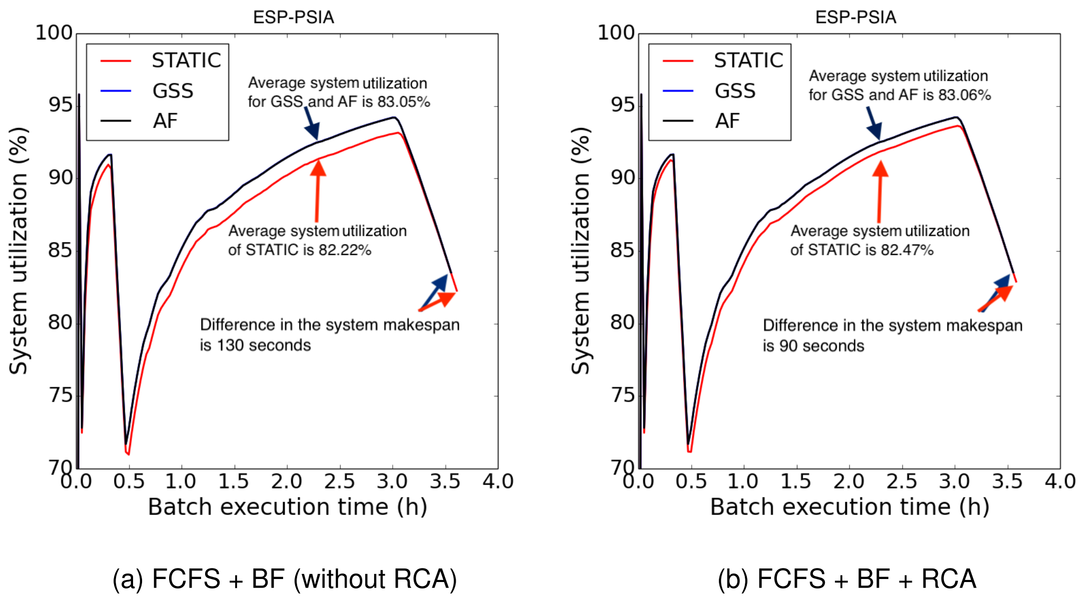


Figure 6.4 System utilization for the ESP-PSIA workload.

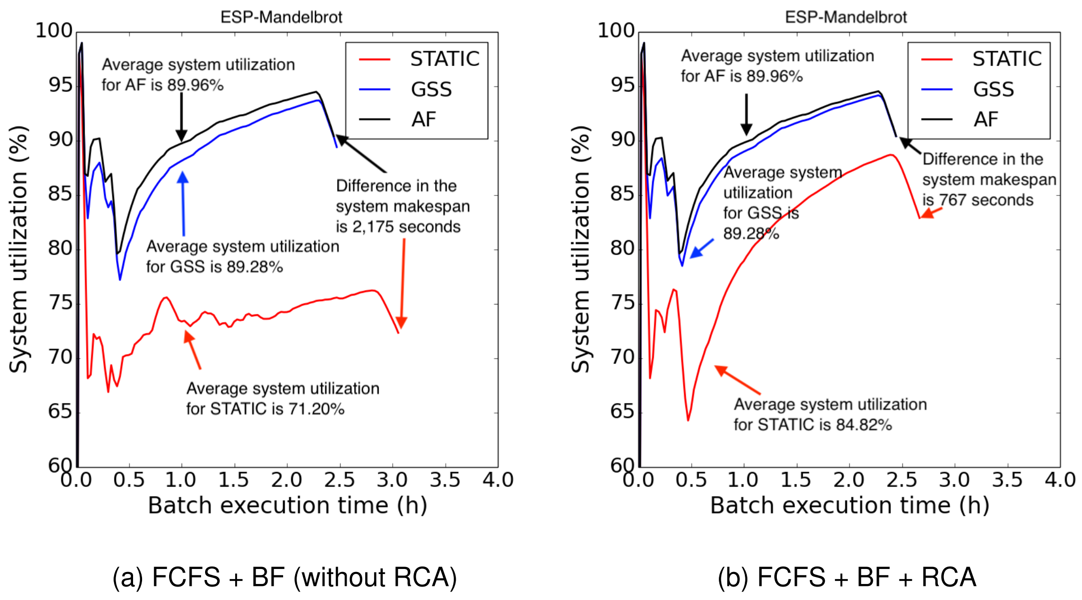


Figure 6.5 System utilization for the ESP-Mandelbrot workload.

For ESP-Mandelbrot, Figure 6.5 shows that RCA increased the average system utilization when the jobs used STATIC from 71.2% to 83.82%. When jobs are executed using GSS and AF, RCA only increase the average system utilization by 0.5% and 0.05%, respectively. This is because AF can achieve a highly balanced execution of all jobs (see Figure 6.3(b)). By enabling our resourceful scheduling approach, the system makespan of the ESP-Mandelbrot using STATIC is reduced from 11,020 to 8,840 seconds (the red curves in Figures 6.5(a) and 6.5(b)).

In general, when all jobs are highly load-balanced, our approach offers slight improvements in terms of increased system utilization. However, this slight improvements in system utilization are of high value for HPC operators as they translate into efficient power consumption [SLG+14]. Future work will explore the relation between RCA and power consumption efficiency.

Because of the new feature that we added to the Slurm simulator [SIJ+17], we can also visualize the execution trace of the workload at coarse- and fine-grain scales. The left side of Figure 6.6 shows the entire ESP-Mandelbrot execution trace in which STATIC is used at the ALS, FCFS+BF is used at the BLS, and the *proposed resourceful ordination approach is not enabled*. The right side of Figure 6.6 is a horizontal zoom into the timeline of the execution trace from 415 to 550 seconds. Zooming at such a fine-time resolution helps to understand the poor system utilization, i.e., certain jobs J8, J9, J10, and J11 are waiting for the latest computing resources of job J7 to become free.

Figure 6.7 shows the execution trace of the same scenario (STATIC at ALS and FCFS+BS at BLS) with the *proposed resourceful coordination approach enabled*. At the coarse-grain time scale (left side), the intensity of the green color (busy computing resources) is higher in Figure 6.7 than Figure 6.6. The total system makespan is shorter in Figure 6.7 than Figure 6.6 by 1,413 seconds. On the right side of Figure 6.7 (horizontal zoom from 415 to 550 seconds), due to the usage of the *proposed resourceful coordination approach*, jobs J8, J9, J10, and J11 started earlier than in Figure 6.6. This reduces the idle times of the computing resource and increases the overall system utilization.

Jobs J8, J9, J10, and J11 in Figure 6.7 are assigned to non-contiguous hosts compared to their resource allocation in Figure 6.6. In practice, such a non-contiguous resource allocation may cause performance degradation for communication-intensive applications. The applications PSIA and Mandelbrot used in the current work are computationally-intensive. Therefore, such a non-contiguous allocation bears no effect on their simulated performance.

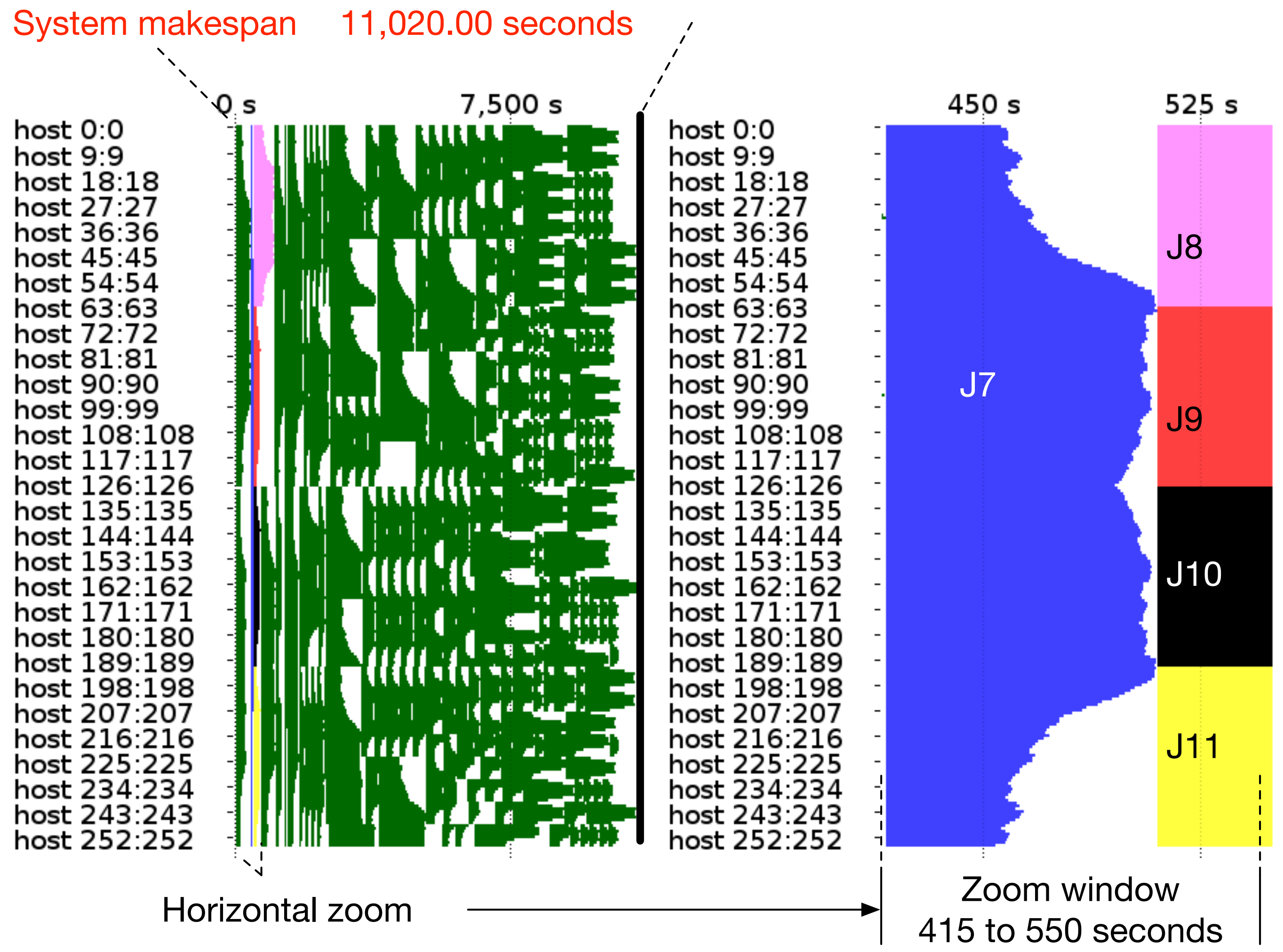


Figure 6.6 **Visualization (obtained using Vampir [KBD+08]) of the execution trace of the ESP-Mandelbrot workload.** STATIC is used at the ALS, while FCFS+BF is used at the BLS. White spaces indicate idle computing resources, while colors denote executing jobs. The short timeline on the right side of each sub-figure is a zoom into a certain time interval of the timeline on the left side. Poor system utilization with RCA at 71.20% due to idle resources, while J7 approaches completion.

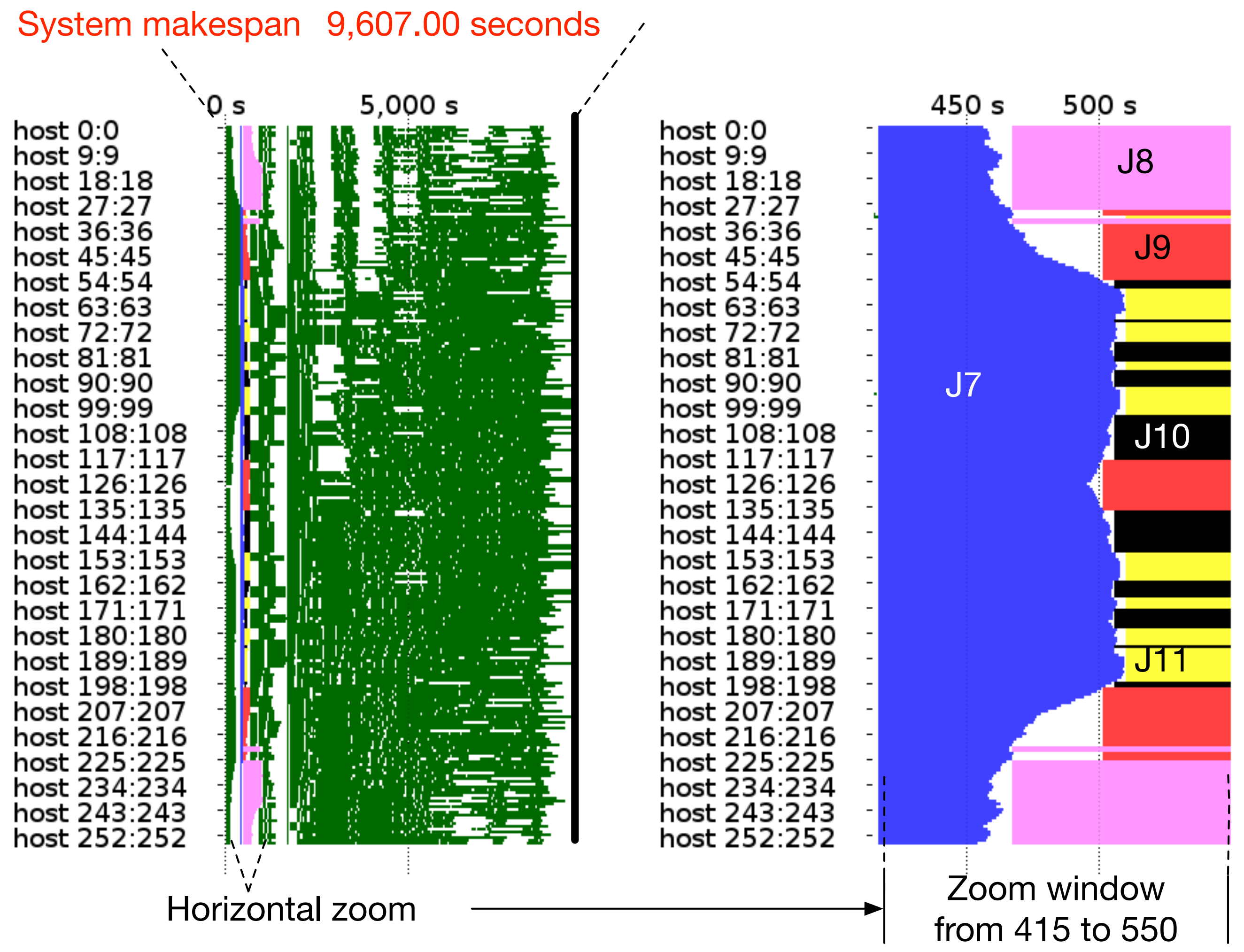


Figure 6.7 Visualization (obtained using Vampir [KBD+08]) of the execution trace of the ESP-Mandelbrot workload. STATIC is used at the ALS, while FCFS+BF is used at the BLS. White spaces indicate idle computing resources, while colors denote executing jobs. The short timeline on the right side of each sub-figure is a zoom into a certain time interval of the timeline on the left side. Improved system utilization with RCA at 84.82%, while J7 approaches completion. J8 and J9 start earlier and utilize the idle resources.

6.4 Summary

This chapter showed the resourceful coordination approach (RCA) that allows application schedulers to cooperate by involving the batch scheduler. The proposed approach is implemented in a two-level scheduling simulator using realistic and well-known simulators (a Slurm-based simulator [SIJ+17] and a SimGrid-based simulator [MEC+20]). The effective system performance (ESP) benchmark was used to assess the proposed approach. ESP jobs were instantiated with the parallel spin-image generation and the Mandelbrot set.

RCA increased the entire system utilization by 12.6% and decreased the system makespan by the same percentage when the applications had a severe load imbalance. System utilization was slightly improved by 0.05% when applications had balanced execution. These improvements are of high value for HPC operators as they translate to efficient power consumption [SLG+14]. The present work also shows that for long-executing HPC applications, exploiting computing resources' idle times (in the order of a few seconds) can significantly improve the entire system utilization. Prior to this work, it was commonly accepted that short computing resource idle times filled by Big Data workloads [MGG+17]. The current work highlighted the potential of exploring such idle times also for HPC workloads as well.

The proposed extensions to the Slurm-simulator [SIJ+17] enabled the visual analysis of the workload execution at coarse- and fine-grain temporal resolutions using Vampir [KBD+08]. With RCA, the visual analysis showed that idle resources were exploited efficiently and jobs were not assigned to contiguous computing resources. Such a non-contiguous resource allocation may cause performance degradation of communication-intensive applications, which were not in the scope of the present work but planned as future work.

7 The Multilevel Scheduling (MLS) Prototype

This chapter introduces the MLS prototype and highlights the prototype's implementation details. The MLS prototype connects the job scheduler of Slurm [YJG03] (at the batch level) with the LB4MPI scheduling library [MEC+20; MC20] (at the application level). Figure 7.1 shows how the MLS prototype connects the batch and application level scheduling. The wide scheduling portfolio offered in LB4MPI and implemented using the DCA allows applications to minimize their execution times and avoid computing resources' idle time. The connection between LB4MPI and the Slurm scheduler allows LB4MPI to report idle computing resources instantaneously. The RCA (implemented as a thread in the Slurm scheduler) allows Slurm to reassign idle computing resources of one job to execute pending jobs.

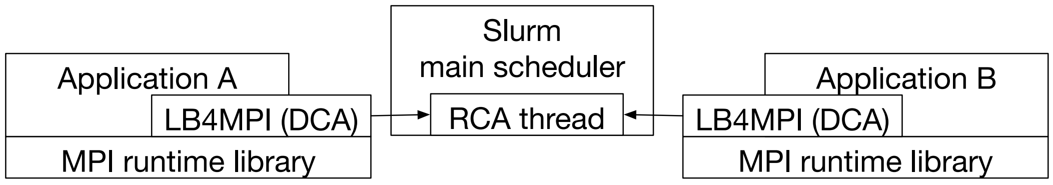


Figure 7.1 **The MLS prototype.** LB4MPI (at the application level) manages the allocated computing resources. When there is no more work to schedule on a particular computing resource, LB4MPI instantaneously reports such information to a specific thread in the Slurm, called RCA thread. The RCA thread marks that resource as a free and allows the main Slurm scheduler to reassign that resource to pending applications.

7.1 DCA in a Scheduling and Load Balancing Library

LB4MPI [1] [MEC+20; MC20] is a recent MPI-based library for loop scheduling and dynamic load balancing. LB4MPI extends the LB tool [CB05] by including certain bug fixes and additional DLS techniques. LB4MPI has been used to enhance the performance of various scientific applications [MC20]. We extend the LB4MPI in two directions: (1) We enable the support of DCA. All the DLS techniques originally supported in LB4MPI were implemented with a centralized chunk calculation approach (CCA), as shown in Figure 7.2. We re-implement them with DCA, as shown in Figure 7.3. (2) We add six additional DLS techniques and implement them with CCA and DCA. These DLS techniques are TAP [Luc92], TFSS [CAB+01], FISS [PD97], VISS [PD97], RND [CIB18], and PLS [SYT07].

LB4MPI has six API functions: `DLS_Parameters_Setup`, `DLS_StartLoop`, `DLS_-Terminated`, `DLS_StartChunk`, `DLS_EndChunk`, and `DLS_EndLoop`. For backward compatibility reasons, our extension of LB4MPI maintained the six original APIs and their signature. However, we added a new API: `Configure_Chunk_-Calculation_Mode` that selects between CCA and DCA. We changed each of the six APIs' functionality to include a condition that checks the selected approach (CCA or DCA). When the selected approach is CCA, the six APIs work as in the original LB4MPI. For instance, `DLS_StartChunk` calls either `DLS_StartChunk_-Centralized` or `DLS_StartChunk_Decentralized` based on the selected approach. `DLS_StartChunk_Centralized` is a function that wraps the original CCA of LB4MPI, while `DLS_StartChunk_Decentralized` provides the newly added functionality that supports DCA.

One can use LB4MPI as in Listing 7.1. One important observation regarding Listing 7.1 is that LB4MPI does not perform any data exchange related to the allocated chunks. LB4MPI assumes that each MPI process has access to the data associated with the loop iterations it executes. The simplest way to ensure the validity of this assumption is to replicate the data of all loop iterations across all MPI processes. Users can also centralize or distribute data of the loop iterations across all MPI processes. In this case, however, users need to provide a way to their applications to exchange the required data associated with the loop iterations.

When a worker calls `DLS_EndLoop`, this means that LB4MPI will not sched-

[1] https://github.com/unibas-dmi-hpc/DLS4LB.git

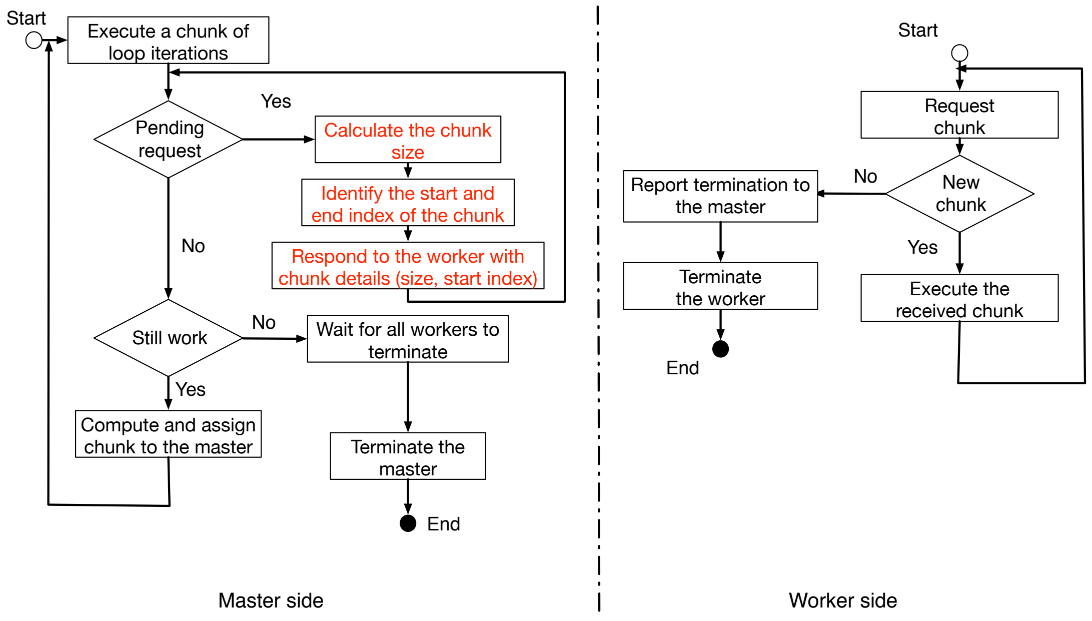

Start
Execute a chunk of loop iterations
Pending request
Yes
Calculate the chunk size
Identify the start and end index of the chunk
Respond to the worker with chunk details (size, start index)
No
Still work
No
Wait for all workers to terminate
Yes
Compute and assign chunk to the master
Terminate the master
End
Master side
Start
Request chunk
New chunk
No
Report termination to the master
Terminate the worker
End
Yes
Execute the received chunk
Worker side

Figure 7.2 The centralized chunk calculation approach (CCA) in LB4MPI. The master performs chunk calculation and assignment for all worker requests. The operations written in the red font represent the centralized operations performed by the master.

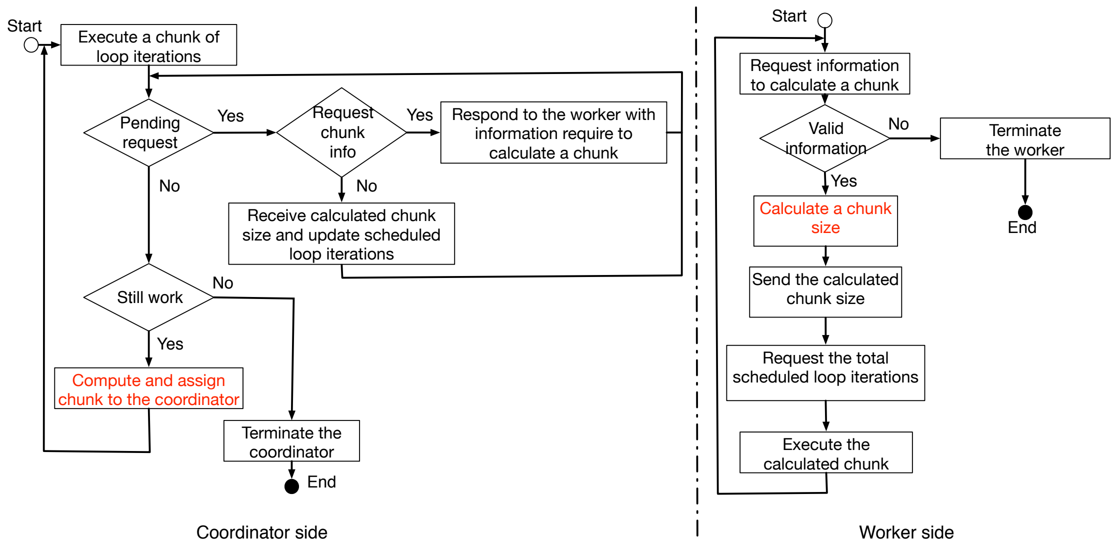

Start
Execute a chunk of loop iterations
Pending request
Yes
Request chunk info
Yes
Respond to the worker with information require to calculate a chunk
No
No
Receive calculated chunk size and update scheduled loop iterations
Still work
No
Yes
Compute and assign chunk to the coordinator
Terminate the coordinator
End
Coordinator side
Start
Request information to calculate a chunk
Valid information
No
Terminate the worker
End
Yes
Calculate a chunk size
Send the calculated chunk size
Request the total scheduled loop iterations
Execute the calculated chunk
Worker side

Figure 7.3 The distribution chunk calculation approach (DCA) in LB4MPI. Each worker calculates its chunk and the coordinator allow worker to synchronize for chunk assignment.

Listing 7.1: Usage of LB4MPI for loop scheduling and dynamic load balancing in scientific applications

```
#include<mpi.h>
#include<LB4MPI.h>
int main()
{
        /*...application code ...*/
        int mode = DECENTRALIZED; /* Or CENTRALIZED */
        Configure_Chunk_Calculation_Mode(mode);
        infoDLS iInfo;  /* a data structure  that holds
        the scheduling information */
        DLS_Parameters_Setup(&iInfo); /* Scheduling params
        include number of tasks, scheduling method,
        scheduling parameters mean, std, ... etc */
        DLS_StartLoop(iInfo, start_index,end_index, scheduling_method);
        while{!DLS_Terminated(info)}
        {
            int start;
            int chunk_size;
                DLS_StartChunk(iInfo, &start, &chunk_size);
                /*... application code to process loop from start to
                start + chunk_size ...*/
                DLS_EndChunk(iInfo);
        }
        DLS_EndLoop(iInfo);
        /*... rest of the application code that does not dominate
        the performance ...*/
}
```

ule any chunks to be executed on the computing resource of that worker. We consider `DLS_EndLoop` to be the communication point where LB4MPI sends a messages to Slurm that one resource becomes idle and can be reassigned to another jobs. Therefore, we extend `DLS_EndLoop` as shown in Listing 7.2. As one may notice, the extension does not depend on the chunk calculation mode, i.e., both CCA and DCA modes report idle resources.

7.1.1 Performance Assessment of DCA in LB4MPI

PSIA and Mandelbrot are used to evaluate the performance of the scheduling techniques in LB4MPI (implemented using both chunk calculation approaches: DCA and CCA). DCA and CCA were assessed in three different scenarios. These scenarios represent cases when a system slowdown affects the PEs and results in slowing down the chunk calculation function.

In the first scenario, no delay is injected during the chunk calculation. In the other two scenarios, a constant delay is injected in the chunk calculation. The injected delay was 10 and 100 microseconds for these two scenarios, respectively. The target experimental system is miniHPC (see Section 4.4). We used sixteen dual-socket nodes (Intel Xeon E5-2640 with 10 cores per socket).

In Figure 7.4(a), using CCA, the parallel loop execution time T_{loop}^{par} is 73.41

Listing 7.2: Connection point between LB4MPI and Slurm

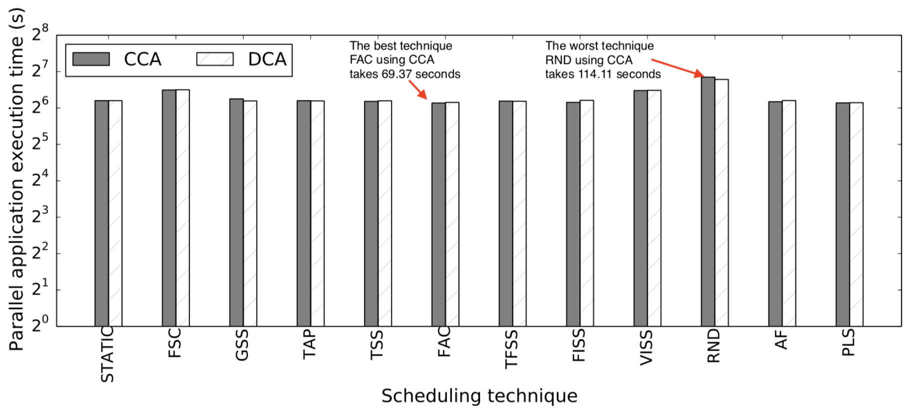
```
#include<LB4MPI.h>
#include <utmpx.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
void DLS_EndLoop(infoDLS *info)
{
    switch(Chunk_Calculation_Mode)
    {
        case DECENTRALIZED:
             DLS_EndChunk_Decentralized(info);
            break;
        default:
            DLS_EndChunk_Centralized(info);
            break;
    }
    char host_name [100];
    gethostname(host_name, 100);
    sprintf(message,"%s", host_name );
    int sock = 0, valread;
    struct sockaddr_in serv_addr;
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        log("\n␣Socket␣creation␣error␣\n");
        return;
    }
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
    int error=inet_pton(AF_INET, SLURM_IP, &serv_addr.sin_addr);
    if(error<=0)
    {
        log("\nInvalid␣address/␣Address␣not␣supported␣\n");
        exit(error);
    }
    error=connect(sock, &serv_addr, sizeof(serv_addr));
    if(error<0)
    {
        log("\nConnection␣Failed␣\n");
        exit(error);
    }
    send(sock, message, strlen(message), 0);
    close(sock);
}
```

seconds with STATIC, while the best T_{loop}^{par} is 69.37 with FAC2. With FAC2, the performance of PSIA is enhanced by 5.5%. Other techniques achieve comparable performance. For instance, T_{loop}^{par} is 69.53 seconds with PLS. In contrast, other techniques degrade the performance of PSIA. GSS and RND degrade the PSIA performance by 2.7% and 61.2% compared STATIC. For the DCA, one can make the same observations regarding the best and the worst techniques. The CCA and DCA versions of all techniques are comparable to each other, i.e., the difference in performance ranges from 2% to 3%.

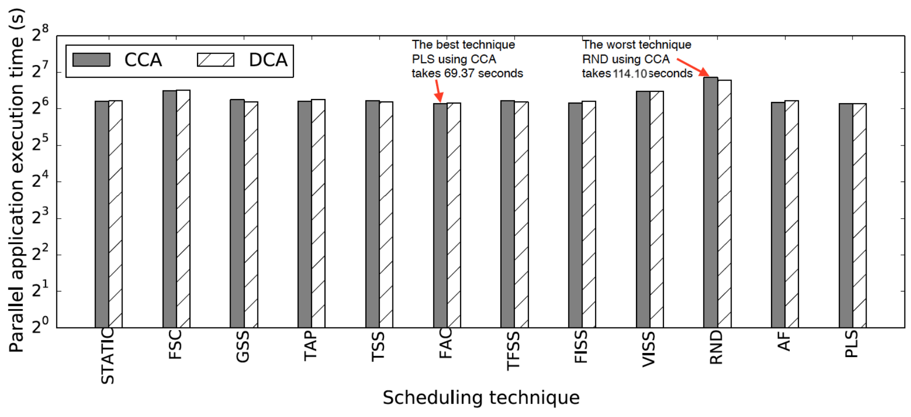
Figures 7.4(b) and 7.4(c) show the performance of both CCA and DCA with different techniques for PSIA when the injected delay is 10 and 100 microseconds, respectively. In Figure 7.4(b), one can notice that when the injected delay is 10 microseconds, the performance differences between CCA and DCA with all techniques range from 2% to 3%. Considering the variation in T_{loop}^{par} of the 20 repetitions of each experiment, one observes that both approaches still have a comparable performance.

For the largest injected delay, the DLS techniques implemented with CCA are more sensitive than the DLS techniques implemented with DCA (see Figure 7.4(c)). For Mandelbrot, one can notice the same behavior, i.e., when there is no injected delay or when the inject delay is 10 microseconds, the performance differences between CCA and DCA with all techniques are minor (see Figures 7.5(a) and 7.5(b)). In contrast, Figure 7.5(c) shows that the DCA version of all the DLS techniques is more capable of maintaining its performance than the CCA version.

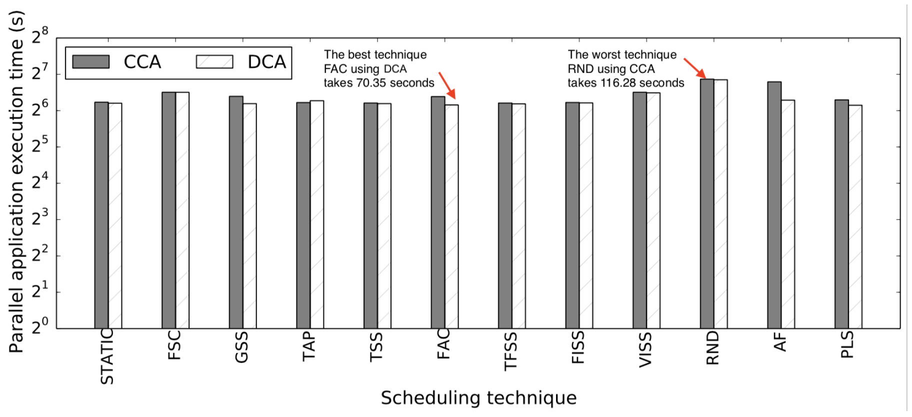
Another interesting observation is the poor AF performance with CCA (see Figure 7.5(c)). AF is an adaptive technique, and it accounts for all sources of load imbalance that affect applications during the execution. However, AF only considers mu_{pi} and σ_{pi}. Since we inject the delay in the chunk calculation function, AF cannot account for such a delay, and it works similarly to the case of no injected delay. Considering the Mandelbrot application's characteristics, the majority of the AF chunks are equal to 1 loop iterations. This fine chunk size leads to an increased number of chunks, i.e., the performance significantly decreased because the injected delay is proportional to the total number of chunks. For PSIA, the corresponding AF implementation (with CCA) does not have the same extreme poor performance (see Figure 7.4(c)) because the AF chunk sizes in the case of PSIA are larger than the chunk sizes in the case of Mandelbrot.



(a) Without an injected delay

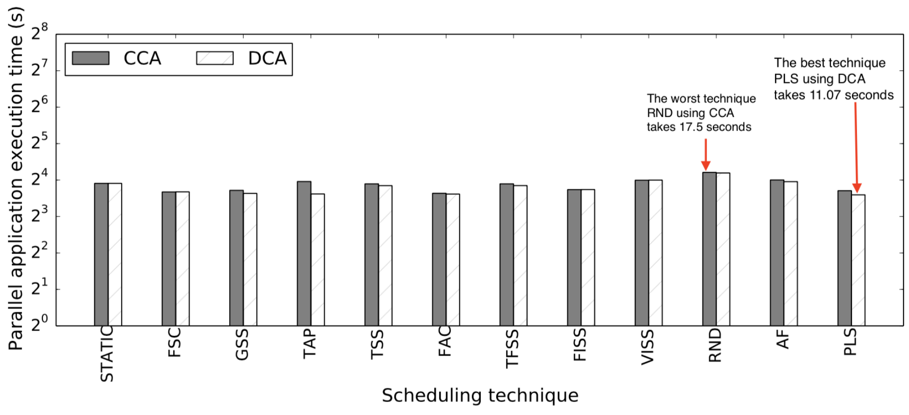


(b) With low injected delay (10 microseconds)

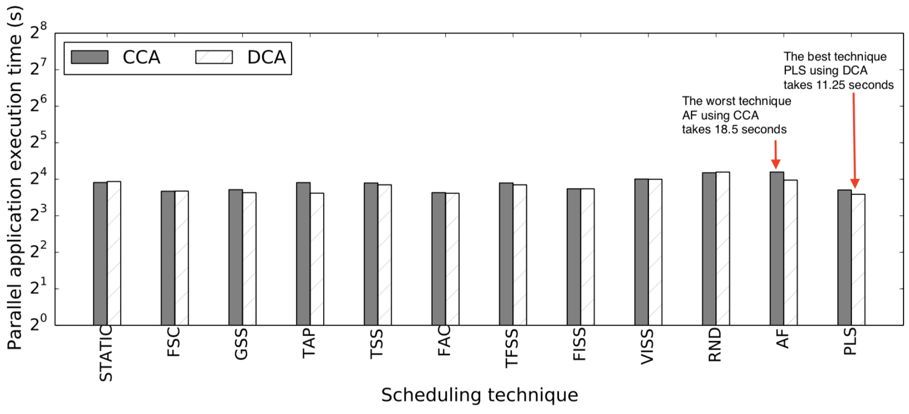


(c) With severe injected delay (100 microseconds)

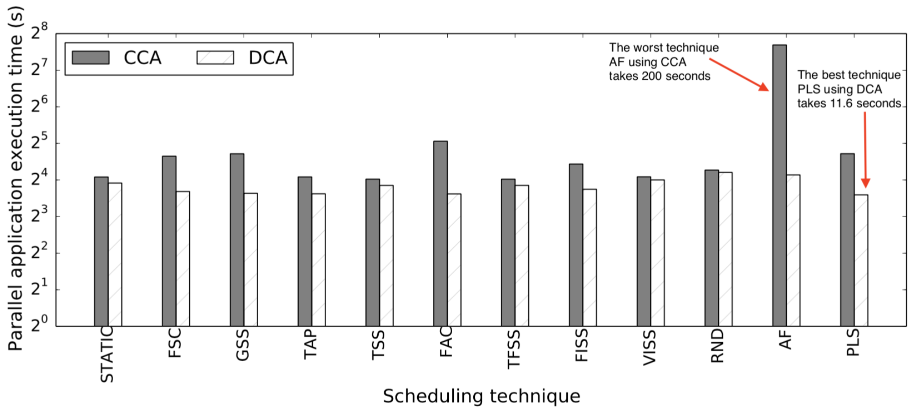
Figure 7.4 Parallel application execution time of PSIA in the three slowdown scenarios.



(a) Without an injected delay



(b) With low injected delay (10 microseconds)



(c) With severe injected delay (100 microseconds)

Figure 7.5 Parallel application execution time of Mandelbrot in the three slowdown scenarios.

7.2 RCA in a Production Batch Scheduler

Slurm is an open-source software, which is commonly used to manage HPC clusters in government laboratories, universities, and companies worldwide. Since Slurm was introduced in 2003 and until now, its usage constantly spreads in the HPC community. For instance, in 2013, Slurm was used on 50% of the ten most powerful supercomputers [Sch20]. Slurm has a unique design that relies on three components that interact to manage jobs' execution and system resources. Figure 7.6 shows the main three components of Slurm: slurmctld, slurmd, and slurmdbd.

The *slurmd* component is a multithreaded daemon (a process that executes in the background). Each computing node executes a single slurmd daemon. Every slurmd daemon notifies about the status of its compute node. The slurmd daemons are responsible for executing jobs on their resources and exchanging job and node status with the slurmctld daemon (main controller).

The *slurmdbd* component is another daemon that is responsible for storing the job accounting information. The job accounting information includes job arrival time, request resource, start time, execution time, etc. The user can query job accounting information with a command utility, called *sacct*. The slurmdbd either interfaces with a database server to store the job accounting information or directly writes job information to an ordinary file as plain text.

The *slurmctld* component, also known as the main controller, is the central component of Slurm. It has three responsibilities:

- Polling slurmd daemons to receive their periodic updates to ensures that nodes their intended configuration
- Gathering nodes into local groups, called partitions. The partitions are used to apply a common configuration to a group of compute nodes. For instance, jobs submitted to a specific partition should finish execution within a time limit.
- Scheduling jobs by accepting user jobs, assigning jobs to compute nodes, and adding pending jobs in a priority ordered queue. Jobs can be pending because there are insufficient compute nodes to start.

The slurmctld daemon has a unique multithreaded design that preferably requires the slurmctld daemon to execute on a dedicated node, called a head node. The slurmctld daemon carries out its responsibilities by spinning off several threads. Certain threads have the lifetime of the slurmctld daemon.

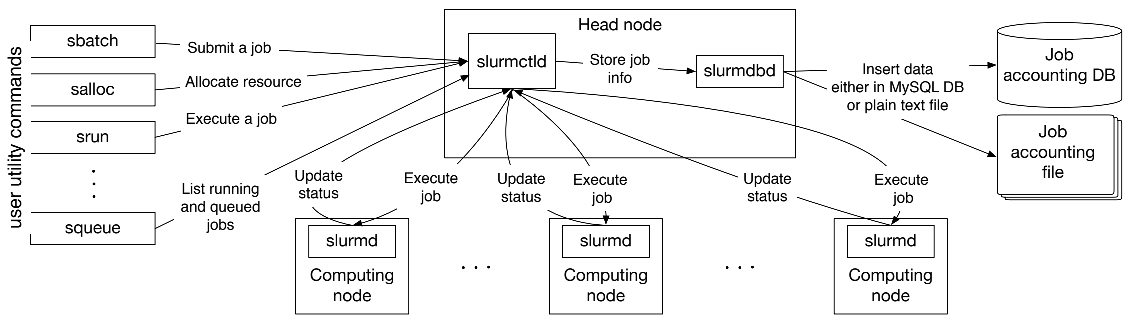


Figure 7.6 The main components of Slurm.

The job scheduling functionality (*slurmctld_background*) executes as a separate thread. The slurmctld daemon also starts short lifetime threads to initiate/retire jobs. Such a design of the slurmctld daemon supports high scalability, availability, and high computing throughput [YJG03].

Inspired by the multithreaded design of the slurmctld daemon, we add a new thread, called RCA thread. The newly added RCA thread handles communication messages from the ALS library (LB4MPI in our case). This thread accepts ALS communication messages on a specific port that the user can define as a Slurm configuration parameter (slurm.conf file). The message contains information regarding the node that the ALS scheduling library wants to share with other applications. The thread extracts the node's hostname and calls a function that makes the node available for reassignment by the main scheduler. Such a function is called `share_node_with_others` and described in Listing 7.3.

Slurm intensively uses certain global variables that we also reuse to develop `share_node_with_others`. For instance, we use two global variables, called `job_list` and `node_record_table_ptr`, defined in `slurmctld.h`. `job_list` represents a linked list that maintains a detailed record of each job submitted to Slurm. Each item in this linked list is of type `job_record` that is defined in `slurmctld.h`. The `node_bitmap` is among the several pieces of information that a `job_record` holds. A bitmap is an efficient representation that Slurm uses to represent node status, i.e., each bit in the map corresponds to a compute node. The corresponding bit of the `node_bitmap` of the `job_record` is set to true for all allocated computing nodes. Otherwise, it is false. `node_record_table_ptr` also represents a linked list that maintains a detailed record of each node in the system. Each item in this linked list is of type `node_record` that is defined in `node_conf.h`. The hostname is among the several pieces of information a

Listing 7.3: Marking a compute node as available for other applications

```
void share_node_with_others(char * target_node)
{
    struct job_record * job_ptr=NULL;

     // job_list is a global variable defined by Slurm
     // holds information about all jobs submitted to the system
    ListIterator job_iterator = list_iterator_create(job_list);

    //node_record_table_ptr is a global variable deined by slurm
    // holds information about all nodes in the systems
    struct node_record *node_ptr = node_record_table_ptr;

    // iterate over all jobs
    while ((job_ptr = (struct job_record *)list_next(job_iterator)))
    {
        if(job_ptr->job_state!=JOB_RUNNING)
        {
            // continue to check another job
            continue;
        }
        // converts the node bitmap to list of node hostnames
        char * nodes_list= bitmap2node_name(job_ptr->node_bitmap);

        // check whether the nodes_list has the target_node
        int res= has_node(target_node, nodes_list);

        if(res!=-1)
        {
            //Job that executes on the target node is identified
            break;
        }
    }
    // free the job_iterator
    list_iterator_destroy(job_iterator);

    if(job_ptr==NULL)
    {
        // this share request is not valid. ignore it
        return;
    }

    // make sure that the job has more than one node
    int nodes_count=bit_set_count(job_ptr->node_bitmap);
    if(nodes_count>1)
    {

        node_record=get_node_record(target_node,target_node );
        excise_node_from_job(job_ptr,node_ptr);
        info("job␣%d␣shared␣%s",job_ptr->job_id, node_ptr->name);
        queue_job_scheduler();
    }
}
```

`node_record` holds.

The incoming messages (from the ALS library) contain only the *hostname* of the node to be shared. The message has no information about the job itself (job id, allocated nodes, start time, execution constraints, etc.). The logic in `share_node_with_others` has two parts; the first part is to identify the job that is associated with the incoming message; the second part is to free and reassign the resource(s) identified in the incoming message.

In the first part, the code iterates overall jobs in the `job_list`, and uses the `node_bitmap` of each `job_record` in the `job_list` to check whether the hostname in the message belongs to the current job allocation. Once this check returns true, the current `job_record` represents the target job. We also identify the target node record by checking the hostname of each `job_record` in the `node_-record_table_ptr`.

In the second part, we use certain functions that are defined in the Slurm source code. The first function is `excise_node_from_job`. This function takes the two input parameters (`job_record` and `node_record`) that have been identified in the first part. The two parameters are used by `excise_node_from_job` to remove the given node from the allocation of the given job as follows. It marks the node's corresponding bit in the Slurm's global bitmaps (`idle_node_bitmap`, `avail_node_bitmap`). Marking nodes as available and idle does not mean that the job binaries have been killed on the node, i.e., the job executes and owns the allocation. However, the scheduling library at the application level will not schedule any tasks on the node. Thus, the node can be shared with other applications via the batch level scheduler.

As discussed before, Slurm has a multithreaded design in which the primary scheduling function runs on a separate thread. The scheduling thread is periodically awakened or whenever scheduling events occur, such as job arrival and completion. Therefore, `share_node_with_others` calls a Slurm function, called `queue_job_scheduler` that immediately causes the main scheduling thread to wake up.

7.3 Performance Evaluation and Discussion

Evaluating the MLS prototype in a production system is challenging because it means interrupting and disturbing the original RMJS of that production system. Furthermore, all Slurm daemons require root privileges to execute appropriately. Therefore, 16 compute nodes were segregated from miniHPC [Cio18] and dedicated to install and evaluate the prototype. The 16 nodes form an HPC cluster (MLS cluster) with one head node and 15 compute nodes. Table 7.1 describes the hardware and the software specification of the MLS cluster.

Table 7.1 Software and hardware components of the MLS cluster

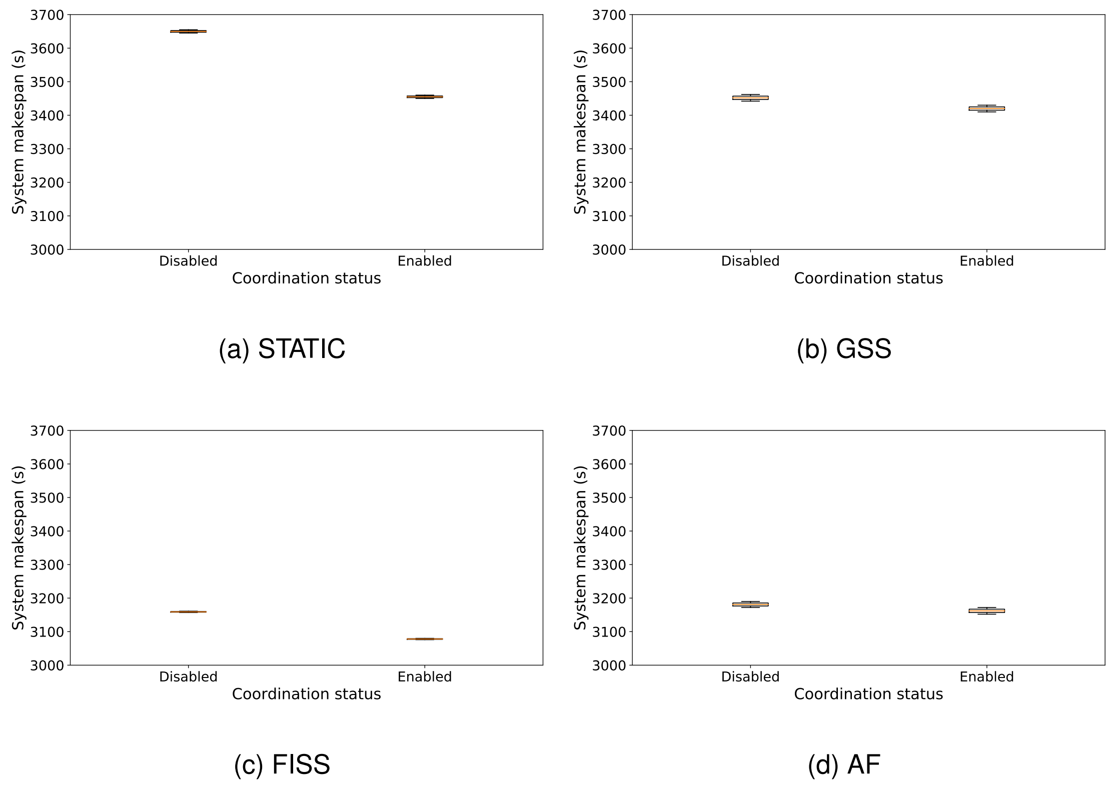
Parameter	Value
Operating system	CentOS Linux release 7.2.1511
Job scheduler	Slurm 20.02.1
Compiler	OpenMPI 2.0.2/GCC-6.3.0-2.27
Number of nodes	16
Processor	Intel Xeon E5-2640 v4
Hyper-threading	disabled
Operating frequency	2.4 GHz
RAM	64 GB per node
Topology	non-blocking fat tree
Interconnection	Intel Omni-Path
Bandwidth	100 Gbit/s
Latency	100 ns

In our evaluation, we use the ESP [WOK+00b; WOK+00a] benchmark. As discussed in Chapter 6, the ESP benchmark can be exemplified with any parallel application. For the MLS evaluation, we exemplify ESP with Mandelbrot [Man80]. We selected FCFS with BF at the batch scheduling level. We selected STATIC, GSS, FISS, and AF at the application level.

Another challenge is the expected variation in jobs' execution time. Such a variation does not exist when event-based simulators (like the proposed one in Chapter 6) are used. For the MLS prototype, the expected variation necessitates multiple times of repetitions per scheduling experiment. Each scheduling experiment has been repeated five times.

Figure 7.7(a) shows the performance of the MLS prototype in two scenarios: (1) the coordination between ALS and BLS is disabled, and (2) the coordination is enabled. In the first scenario, the average system makespan is 3630 seconds, while in the second scenario, the average system makespan is 3455 seconds. The difference between the two scenarios is 175 seconds. This result means that by enabling the coordination, the makespan is reduced by 4.82%.

Such improvement is justified by employing STATIC as the ALS technique used by all the ESP jobs. The ESP jobs are exemplified with Mandelbrot. As explained in Section 4.4, Mandelbrot with STATIC represents the maximum load imbalance execution.



(a) STATIC (b) GSS

(c) FISS (d) AF

Figure 7.7 System makespan of the ESP (Mandelbrot) with different application level scheduling techniques. Slurm is configured to use FCFS for BLS.

Figures 7.7(b), 7.7(c) and 7.7(d) represent other cases when GSS, FISS, and AF were used at the application level. With GSS, FISS, and AF, Mandelbrot has a balanced execution. This is reflected in the achieved improvement in these cases. For instance, employing GSS, FISS, and AF lead to 0.9%, 2.5%, and 0.6% improvement in the system makespan.

When the coordination between BLS and ALS is enabled, the selected ALS technique's influence becomes less significant on the system makespan. For instance, without enabled coordination, the average system makespan is 3630, 3452, 3159, 3181 seconds for STATIC, GSS, FISS, and AF, respectively. By enabling the coordination, the average system makespan is 3455, 3420, 3078, and 3161 seconds. In general, the results confirm the same patterns observed in the simulation (see Section 6.3). Therefore, we can conclude that sharing information about idle computing resources enables coordination between batch and application schedulers and has significant performance potential.

8 Conclusions and Future Work

This chapter describes and summarizes the main conclusions that came out of the work presented in this doctoral dissertation. This chapter also outlines the future extensions of this work.

8.1 Conclusions

Throughout this doctoral dissertation, it has been shown that idle times of computing resources towards the end of applications' execution negatively impact performance at the batch and application levels. Idle times degrade performance at the batch level because they decrease system utilization. They also degrade performance at the application level because they increase applications' execution time. Therefore, a coordination between schedulers at various levels of hardware parallelism exploits these idle times.

This conclusion is supported by analyzing simulation results of an exploratory study, which has been conducted on workload traces obtained from large-scale HPC systems in production. The exploratory study includes twelve combinations of three BLS and four ALS techniques and is enabled by the two-level scheduling simulator proposed in Chapter 3.

The two-level scheduling simulator connects two well-known simulators (GridSim and SimGrid), i.e., each simulator is responsible for a certain scheduling level. By collecting the simulation events of both simulators and storing these events in an OTF2 format, we are able to visualize using Vampir [KBD+08], for the first time, the system utilization from system to core level. This visualization allowed us to conclude that **coordination absence between BLS and ALS techniques hinders exploiting idle times of computing resources**.

Also, we have learned that the standard workload format (SWF), which was used to store workload traces for the past two decades, **is of limited usefulness for simulating BLS and ALS**. SWF does not preserve any information about how applications schedule their tasks on the allocated resources. To overcome this limitation, we have proposed the task variation factor that varies the tasks' length within a certain application randomly (see Chapter 3). However, the proposed task variation factor does not eliminate the need for storing additional information about how applications schedule their tasks on the allocated resources.

Dynamic loop scheduling (DLS) techniques are essential to improve applications' performance by mitigating all sources of load imbalance. We examined and assessed the performance of twelve well-known DLS techniques at the application level. We highlighted a shift in their development. Originally, DLS techniques are devised for shared-memory systems. In the middle of the 1990s, Beowulf clusters and the first MPI standard appeared. Since that time, DLS techniques have been implemented on distributed-memory systems by employing a master-worker execution model that centralizes chunk calculation and chunk assignment at the master side.

We have concluded that the **centralization of chunk calculation and chunk assignment** contributes to idle times of computing resources, i.e., a worker waits for the master to calculate and assign work to all other workers. Thus, the distribution of chunk calculation across all workers is essential for applications' performance. We proposed a distributed chunk calculation approach (DCA) and its hierarchical DCA (HDCA). For both approaches, we have presented the need for straightforward formulas that do not depend on any information about previously calculated chunks. We have shown the mathematical transformation required to change the formulas of the considered DLS techniques into straightforward formulas. We recommend to use these straightforward formulas and the DCA approach for implementing DLS techniques.

Both approaches (DCA and HDCA) are implemented on distributed-memory systems using the latest features (one-sided communications and MPI shared memory) of the latest MPI standard (MPI 3.1). **One primary lesson** learned is that the performance of these latest features significantly depends on the MPI runtime library. For instance, the *lock polling* strategy (employed by Intel MPI) significantly increases scheduling overhead (see Chapter 4). Also, when a large number of MPI processes simultaneously access a shared-memory region, the associated overhead of such an access is higher than the overhead associated with simultaneous access of OpenMP threads (see Chapter 5).

In practice, we have observed and learned that **none of the considered DLS techniques completely eliminates load imbalance**. The considered DLS techniques allow computing resources **to have nearly equal finishing times, but computing resources do not have the exact finishing times**, i.e., certain computing resources experience idle times. In this case, we have proposed a resourceful coordination approach (RCA) that allows batch systems to exploit computing resources once they become free. RCA leverages and combines the advantages of node sharing and dynamic resource and job management. It offers an efficient resource sharing (of idle resources only) and avoids shrinkage and expansion operations on the application side (see Chapter 6).

Employing DCA and RCA in the MLS prototype confirms and promotes our primary conclusion: enabling coordination between batch and application schedulers **via exchanging scheduling information** is crucial to fully exploit computing resources of modern HPC systems (see Chapter 7).

8.2 Future Work

Possible extensions can be explored based on the work presented in this thesis. These possible extensions cover various research directions. In the DLS direction, recent research efforts [KCY+] expanded specific OpenMP runtime libraries by adding more DLS techniques. One possible extension is to apply the proposed DCA in such libraries. We also discussed and implemented specific DLS techniques (TFSS, PLS, FISS, and VISS) in the LB4MPI library (see Chapters 4 and 7). These techniques have not yet been implemented in any of the OpenMP runtime. Adding these techniques to OpenMP runtime libraries, such as LLVM, is essential as it widens the scheduling portfolio for selecting the best performing scheduling.

Few DLS techniques, such as self-adapting scheduling (SAS) [RCA+06], were designed to support scheduling tasks with data dependencies. These techniques consider a centralized chunk calculation. Furthermore, they have never been implemented in any OpenMP runtime library nor any MPI scheduling library. One immediate extension to our work is to study the potential of applying the proposed DCA to such techniques and implement them in the LLVM OpenMP runtime library and in the LB4MPI library.

Another exciting research direction is how to eliminate the required synchronization for the work assignment in the DLS techniques. All DLS techniques (except Fractiling [BFH95]) assume a central work queue that necessitates synchronization for the work assignment between all workers.

In the direction of batch scheduling, the proposed RCA allows applications to share their idle computing resources with other applications through the batch system. One future extension is to develop batch techniques that use such idle resources to achieve specific performance targets. For instance, a batch technique that reuses these idle resources to execute only jobs with short execution time may increase system throughput.

Furthermore, the current doctoral dissertation focused on idle resource times towards the end of applications' execution. One future extension is to study exploiting idle resource times during applications' execution. Such idle times exist in communication-intensive applications, i.e., data transfer between compute nodes happens frequently, and in many cases, compute nodes often remain idle until the data transfer completes.

With the existing computer technology, parallelism remains the gateway for HPC. Future HPC systems will continue to offer massive parallelism at the core, node, and system levels. Various scheduling techniques are employed to schedule computations across all parallelism levels. In this doctoral dissertation, we addressed the following research problem: *the absence of coordination between schedulers at different scheduling levels* in HPC systems. We have shown how multilevel scheduling efficiently exploits multiple levels of hardware parallelism of modern HPC systems.

Bibliography

[ABS+11] Zafril Rizal M. Azmi, Kamalrulnizam Abu Bakar, Mohd Shahir Shamsir, Wan Nurulsafawati Manan, and Abdul Hanan Abdullah. Scheduling Grid Jobs Using Priority Rule Algorithms and Gap Filling Techniques. *Journal of Advanced Science and Technology*, 37:61–76, 2011.

[BSC+12] Mahadevan Balasubramanian, Nitin Sukhija, Florina M. Ciorba, Ioana Banicescu, and Srishti Srivastava. Towards the Scalability of Dynamic Loop Scheduling Techniques via Discrete Event Simulation. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshops*, 2012, pages 1343–1351.

[BCP+05] Ioana Banicescu, Ricolindo L. Cariño, Jaderick P. Pabico, and Mahadevan Balasubramaniam. Design and Implementation of a Novel Dynamic Load Balancing Library for Cluster Computing. *Journal of Parallel Computing*, 31(7):736–756, 2005.

[BFH95] Ioana Banicescu and Susan Flynn Hummel. Balancing Processor Loads and Exploiting Data Locality in N-body Simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, 1995, pages 43–43.

[BV02] Ioana Banicescu and Vijay Velusamy. Load Balancing Highly Irregular Computations With the Adaptive Factoring. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002, 12 pp.

[BVD03] Ioana Banicescu, Vijay Velusamy, and Johnny Devaprasad. On the Scalability of Dynamic Scheduling Scientific Applications With Adaptive Weighted Factoring. *Journal of Cluster Computing*, 6(3):215–226, 2003.

[Ban00] Banicescu, Ioana and Liu, Zhijun. Adaptive Factoring: A Dynamic Scheduling Method Tuned to the Rate of Weight Changes. In *Proceedings of the High performance computing Symposium*, 2000, pages 122–129.

[BFM+06] Marinho P. Barcellos, Giovani Facchini, Hisham H. Muhammad, Guilherme B. Bedin, and Paulo Luft. Bridging the gap between simulation and experimental evaluation in computer networks. In *Proceedings of the Annual Simulation Symposium*, 2006, 8–pp.

[BCC+04] Kevin Barker, Andrey Chernikov, Nikos Chrisochoides, and Keshav Pingali. A Load Balancing Framework for Adaptive and Asynchronous Applications. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):183–192, 2004.

[BW91] Katherine M. Baumgartner and Benjamin W. Wah. Computer scheduling algorithms: past, present and future. *Journal of Information Sciences*, 57:319–345, 1991.

[BG01] Gordon Bell and Jim Gray. High Performance Computing: Crays, Clusters, and Centers. What Next? *Communications of the ACM*, 2001.

[BLR+12] Wes E. Bethel, David Leinweber, Oliver Rübel, and Kesheng Wu. Federal Market Information Technology in the Post–FlashCrash Era: Roles for Supercomputing. *The Journal of Trading*, 7(2):9–25, 2012.

[BLP95] Prashanth B. Bhat, Young Won Lim, and Viktor K. Prasanna. Issues in using heterogeneous hpc systems for embedded real time signal processing applications. In *Proceedings of the International Workshop on Real-Time Computing Systems and Applications*, 1995, pages 134–141.

[BDH+15] Mark S. Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D. Underwood, and Robert C. Zak. Intel Omni-Path Architecture: Enabling scalable, high performance fabrics. In *Proceedings of the Annual Symposium on High-Performance Interconnects*, 2015, pages 1–9.

[BCC+97] Susan L. Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, et al. *ScaLAPACK Users' Guide*, 1997.

[BBHB+07] Guy E. Blelloch, Lenore Blum, Mor Harchol-Balter, and Robert Harper. Multiscale Scheduling: Integrating Competitive and Cooperative Scheduling in Theory and in Practice. http://lambda-the-ultimate.org/node/2337. [Online; accessed 08 August 2020]. 2007.

[Boa18] OpenMP Architecture Review Board. OpenMP Application Programming Interface. `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`. [Online; accessed 23 August 2020]. 2018.

[BMB+13] Joris Borgdorff, Mariusz Mamonski, Bartosz Bosak, Derek Groen, Mohamed Ben Belgacem, Krzysztof Kurowski, and Alfons G Hoekstra. Multiscale Computing With the Multiscale Modeling Library and Runtime Environment. *Journal of Procedia Computer Science*, 18:1097–1105, 2013.

[BWA16] Anthony Boulmier, John White, and Nabil Abdennadher. Towards a Cloud Based Decision Support System for Solar Map Generation. In *Proceedings of the International Conference on Cloud Computing Technology and Science*, 2016, pages 230–236.

[BM02] Rajkumar Buyya and Manzur Murshed. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Journal of Concurrency and Computation: Practice and Experience*, 14(13-15):1175–1220, 2002.

[Can08] Caniou, Yves and Gay, J. -S. Simbatch: An API for Simulating and Predicting the Performance of Parallel Resources Managed by Batch Systems. In *Proceedings of the European Conference on Parallel Processing*, 2008, pages 223–234.

[CB05] Ricolindo L. Cariño and Ioana Banicescu. A Load Balancing Tool for Distributed Parallel Loops. *Journal of Cluster Computing*, 8(4):313–321, 2005.

[CB08] Ricolindo L. Cariño and Ioana Banicescu. Dynamic Load Balancing With Adaptive Factoring Methods in Scientific Applications. *Journal of Supercomputing*, 44(1):41–63, 2008.

[CBR+04] Ricolindo L. Cariño, Ioana Banicescu, Thomas Rauber, and Gudula Rünger. Dynamic loop scheduling with processor groups. In *Proceedings of the 17th international conference on parallel and distributed computing systems.* 2004, pages 78–84.

[CGL+14] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Patforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, 2014.

[CBL08] Marc Casas, Rosa Badia, and Jesús Labarta. Automatic Analysis of Speedup of MPI Applications. In *Proceedings of the International Conference on Supercomputing*, 2008, pages 349–358.

[CFK+18] Bastien Chopard, Jean-Luc Falcone, Pierre Kunzli, Lourens Veen, and Alfons Hoekstra. Multiscale Modeling: Recent Progress and Open Questions. *Journal of Multiscale and Multidisciplinary Modeling, Experiments and Design*, 1(1):57–68, 2018.

[CW10] Martin J. Chorley and David W. Walker. Performance Analysis of a Hybrid MPI/OpenMP Application on Multi-core clusters. *Journal of Computational Science*, 1(3):168–174, 2010.

[CAB+01] Anthony T. Chronopoulos, Razvan Andonie, Manuel Benche, and Daniel Grosu. A Class of Loop Self-scheduling for Heterogeneous Clusters. In *Proceedings of International Conference on Cluster Computing*, 2001, pages 282–291.

[CPY+05] Anthony T. Chronopoulos, Satish Penmatsa, Ning Yu, and Du Yu. Scalable Loop Self-scheduling Schemes for Heterogeneous Clusters. *Journal of Computational Science and Engineering*, 1(2-4):110–117, 2005.

[CIB18] Florina M. Ciorba, Christian Iwainsky, and Patrick Buder. OpenMP Loop Scheduling Revisited: Making a Case for More Schedules. In *Proceedings of the 2018 international workshop on openmp*, 2018, pages 21–36.

[Cio18] Ciorba, Florina M. The miniHPC Cluster. `https://hpc.dmi.unibas.ch/HPC/miniHPC.html`. [Online; accessed 08 August 2020]. 2018.

[CGJ83] Edward G. Coffman Jr, Michael R. Garey, and David S. Johnson. Dynamic Bin Packing. *SIAM Journal on Computing*, 12(2):227–258, 1983.

[CMHG+16] Isaías Comprés, Ao Mo-Hellenbrand, Michael Gerndt, and Hans-Joachim Bungartz. Infrastructure and API Extensions for Elastic Execution of MPI Applications. In *Proceedings of the European MPI Users' Group Meeting*, 2016, pages 82–97.

[DGGL+18] Marco D'Amico, Marta Garcia-Gasulla, Víctor López, Ana Jokanovic, Raül Sirvent, and Julita Corbalan. DROM: Enabling Efficient and Effortless Malleability for Resource Managers. In *Proceedings of the International Conference on Parallel Processing Companion*, 2018, pages 1–10.

[DJC19] Marco D'Amico, Ana Jokanovic, and Julita Corbalan. Holistic slowdown driven scheduling and resource management for malleable jobs. In *Proceedings of the International Conference on Parallel Processing*, 2019, page 31.

[DHJ07] Luiz DeRose, Bill Homer, and Dean Johnson. Detecting Application Load Imbalance on High End Massively Parallel Systems. In *Proceedings of European Conference on Parallel Processing*, 2007, pages 150–159.

[Don03] Jack Dongarra. High Performance Computing Trends and Self Adapting Numerical Software. In *Proceedings of the International Symposium on High Performance Computing*, 2003, pages 1–9.

[Don04] Jack Dongarra. The Boole Lecture. Trends in High Performance Computing. *Computer Journal*, 47(4):399–403, 2004.

[Don20] Jack Dongarra. Report on The Fujitsu Fugaku System. `https://www.icl.utk.edu/files/publications/2020/icl-utk-1379-2020.pdf`. [Online; accessed 02 September 2020]. 2020.

[DSS+05] Jack Dongarra, Thomas Sterling, Horst Simon, and Erich Strohmaier. High-performance Computing: Clusters, Constellations, MPPs, and Future Directions. *Computing in science & engineering*, 7(2):51–59, 2005.

[EC19a] Ahmed Eleliemy and Florina M. Ciorba. Dynamic Loop Scheduling Using MPI Passive-Target Remote Memory Access. In *Proceedings of the Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2019, pages 75–82.

[EC19b] Ahmed Eleliemy and Florina M. Ciorba. Hierarchical Dynamic Loop Self-Scheduling on Distributed-Memory Systems Using an MPI+MPI Approach. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshops*, 2019, pages 689–697.

[EC20] Ahmed Eleliemy and Florina M. Ciorba. A Distributed Chunk Calculation Approach for Self-scheduling of Parallel Applications on Distributed-memory Systems. *Journal of Computational Science*, 2020, revised and resubmitted.

[EC21] Ahmed Eleliemy and Florina M Ciorba. A Resourceful Coordination Approach for Multilevel Scheduling. In *Proceedings of the International Conference on High Performance Computing & Simulation*, 2021.

[EFM+16] Ahmed Eleliemy, Mahmoud Fayze, Rashid Mehmood, Iyad. Katib, and Naif Aljohani. Loadbalancing on Parallel Heterogeneous Architectures: Spin-image Algorithm on CPU and MIC. In *Proceedings of the 9th EUROSIM Congress on Modelling and Simulation*, 2016, pages 623–628.

[EMC16] Ahmed Eleliemy, Ali Mohammed, and Florina M. Ciorba. Simulating Batch and Application Level Scheduling Using GridSim and SimGrid. Extended Abstract at the International Conference for High Performance Computing, Networking, Storage, and Analysis. 2016.

[EMC17a] Ahmed Eleliemy, Ali Mohammed, and Florina M. Ciorba. Efficient Generation of Parallel Spin-images Using Dynamic Loop Scheduling. In *Proceedings of the 8th International Workshop on Multicore and Multithreaded Architectures and Algorithms in conjunction with the 19th IEEE International Conference for High Performance Computing and Communications*, 2017, page 8.

[EMC17b] Ahmed Eleliemy, Ali Mohammed, and Florina M. Ciorba. Exploring the Relation between Two Levels of Scheduling Using a Novel Simulation Approach. In *Proceedings of the International Symposium on Parallel and Distributed Computing*, 2017, pages 26–33.

[EWG+11] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Wolf. Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support

Libraries. In *Proceedings of the International Conference on Parallel Computing*, 2011, pages 481–490.

[FTY+90] Zhixi Fang, Peiyi Tang, Pen-Chung Yew, and Chuan-Qi Zhu. Dynamic Processor Self-scheduling for General Parallel Nested Loops. *IEEE Transactions on Computers*, 39(7):919–929, 1990.

[Fei05] Dror G. Feitelson. Parallel Workloads Archive. http://www.cs.huji.ac.il/labs/parallel/workload/. [Online; accessed 08 August 2020]. 2005.

[Fei20] Dror G. Feitelson. Standard Workload Format. `http://www.cs.huji.ac.il/labs/parallel/workload/swf.html`. [Online; accessed 08 August 2020]. 2020.

[FJ97] Dror G. Feitelson and Morris A Jettee. Improved Utilization and Responsiveness with Gang Scheduling. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, 1997, pages 238–261.

[FR95] Dror G. Feitelson and Larry Rudolph. Parallel Job Scheduling: Issues and Approaches. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, 1995, pages 1–18.

[FR96] Dror G. Feitelson and Larry Rudolph. Toward Convergence in Job Schedulers for Parallel Supercomputers. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, 1996, pages 1–26.

[FRS+97] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C Sevcik, and Parkson Wong. Theory and Practice in Parallel Job Sscheduling. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, 1997, pages 1–34.

[FTK14] Dror G. Feitelson, Dan Tsafrir, and David Krakov. Experience with using the Parallel Workloads Archive. *Journal of parallel and distributed computing*, 74(10):2967–2982, 2014.

[FW98] Dror G. Feitelson and Ahuva Mu'alem Weil. Utilization and Predictability in Scheduling the IBM SP2 With Backfilling. In *Proceedings of the Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, 1998, pages 542–546.

[FBP15] Edson Flórez, Carlos J. Barrios, and Johnatan E. Pecero. Methods for Job Scheduling on Computational Grids: Review and Comparison. In *Proceedings of latin american high performance computing conference*, 2015, pages 19–33.

[FHSU+96] Susan Flynn Hummel, Jeanette Schmidt, R. N. Uma, and Joel Wein. Load-sharing in Heterogeneous Systems via Weighted Factoring. In *Proceedings of the 8th annual ACM symposium on Parallel algorithms and architectures*, 1996, pages 318–328.

[FHSF92] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *Journal of Communications of the ACM*, 35(8):90–101, 1992.

[For20] MPI Forum. Message-Passing Interface. `https://www.mpi-forum.org`. [Online; accessed 23 August 2020]. 2020.

[GCL09] Marta Garcia, Julita Corbalan, and Jesus Labarta. LeWI: A Runtime Balancing Algorithm for Nested Parallelism. In *Proceedings of the International Conference on Parallel Processing*, 2009, pages 526–533.

[GGR+15] Eric Gaussier, David Glesser, Valentin Reis, and Denis Trystram. Improving Backfilling by Using Machine Learning to Predict Running Times. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pages 1–10.

[GH12] Yiannis Georgiou and Matthieu Hautreux. Evaluating Scalability and Efficiency of the Resource and Job Management System on Large HPC Clusters. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, 2012, pages 134–156.

[GAB+96] Martyn F. Guest, Edoardo Apra, David E. Bernholdt, Herbert A. Früchtl, Robert J. Harrison, Ricky A. Kendall, RA Kutteh, X. Long, John B. Nicholas, Jeffrey A. Nichols, et al. High-performance Computing in Chemistry: NW Chem. *Future Generation Computer Systems*, 12(4):273–289, 1996.

[HGC14] Jeff R. Hammond, Sayan Ghosh, and Barbara M. Chapman. Implementing OpenSHMEM Using MPI-3 One-sided Communication. In *Proceedings on the Workshop on OpenSHMEM and Related Technologies*, 2014, pages 44–58.

[Hen95] Robert L. Henderson. Job Scheduling Under the Portable Batch System. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, 1995, pages 279–294.

[HDB+13] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. MPI + MPI: a new hybrid approach to parallel programming with MPI plus shared memory. *Computing*, 95(12):1121–1136, 2013.

[HDT+15] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. Remote Memory Access Programming in MPI-3. *ACM Transactions on Parallel Computing*, 2(2):9, 2015.

[HCB17] Franziska Hoffeins, Florina M. Ciorba, and Ioana Banicescu. Examining the Reproducibility of Using Dynamic Loop Scheduling Techniques in Scientific Applications. In *International Parallel and Distributed Processing Symposium Workshops*, 2017, pages 1579–1587.

[HKK+03] Matthias Hovestadt, Odej Kao, Axel Keller, and Achim Streit. Scheduling in HPC Resource Management Systems: Queuing vs. Planning. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, 2003, pages 1–20.

[How98] Howell, Fred and McNab, Ross. A Discrete Event Simulation Library for Java. In *Proceedings of the International Conference on Web-based Modeling and Simulation*, 1998, page 6.

[IBM16] IBM LSF. Queue-Level User-based Fairshare. [Online; accessed 13 August 2020]. 2016.

[Joh97] Andrew E. Johnson. Spin-Images: A Representation for 3-D Surface Matching. PhD thesis. Robotics Institute, Carnegie Mellon University, 1997.

[JTS09] Raka Jovanovic, Milan Tuba, and Dana Simian. A New Visualization Algorithm for the Mandelbrot Set. In *Proceedings of the 10th WSEAS International Conference on Mathematics and Computers in Biology and Chemistry*, 2009, pages 162–166.

[KR01] Axel Keller and Alexander Reinefeld. Anatomy of A Resource Management System for HPC Clusters. *Annual review of scalable computing*, 3(1):1–31, 2001.

[KT11] Volodymyr Kindratenko and Pedro Trancoso. Trends in High-performance Computing. *Computing in science & engineering*, 13(3):92–95, 2011.

[Kis02] Laszlo B Kish. End of Moore's Law: Thermal (Noise) Death of Integration in Micro and Nano Electronics. *Physics Letters A*, 305(3-4):144–149, 2002.

[KMR07] Dalibor Klusáček, Luděk Matyska, and Hana Rudová. Alea–Grid Scheduling Simulation Environment. In *Proceedings of the International Conference on Parallel Processing and Applied Mathematics*, 2007, pages 1029–1038.

[KR10] Dalibor Klusáček and Hana Rudová. Alea 2: Job Scheduling Simulator. In *Proceedings of the International Conference on Simulation Tools and Techniques*, 2010, page 61.

[KR11] Dalibor Klusáček and Hana Rudová. Efficient Grid Scheduling Through the Incremental Schedule-based Approach. *Journal Computational Intelligence*, 27(1):4–22, 2011.

[KSS19] Dalibor Klusáček, Mehmet Soysal, and Frédéric Suter. Alea - Complex Job Scheduling Simulator. In *Proceedings of the International Conference on Parallel Processing and Applied Mathematics*, 2019, pages 217–229.

[KBD+08] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E. Nagel. The Vampir Performance Analysis Tool-set. In *Proceedings of the International Workshop on Parallel Tools for High Performance Computing*, 2008, pages 139–155.

[KCY+] Jonas H. Müller Korndörfer, Florina M. Ciorba, Akan Yilmaz, Christian Iwainsky, Johannes Doerfert, Hal Finkel, Vivek Kale, and Michael Klemm. A Runtime Approach for Dynamic Load Balancing of OpenMP Parallel Loops in LLVM. Poster at International Conference on High Performance Computing, Networking, Storage and Analysis.

[KW85] Clyde P. Kruskal and Alan Weiss. Allocating Independent Subtasks on Parallel Processors. *IEEE Transactions on Software Engineering*, SE-11(10):1001–1016, 1985.

[LMM+19] Ignacio Laguna, Ryan Marshall, Kathryn Mohror, Martin Ruefenacht, Anthony Skjellum, and Nawrin Sultana. A Large-scale Study of MPI Usage in Open-source HPC Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, page 31.

[LTS+93] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik. Locality and loop scheduling on numa multiprocessors. In *Proceedings of the international conference on parallel processing*, 1993, pages 140–147.

[LF03] Uri Lublin and Dror G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing*, 63(11):18, 2003.

[Luc92] Steven Lucco. A Dynamic Scheduling Method for Irregular Parallel Programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1992, pages 200–211.

[Luc11] Alejandro Lucero. Simulation of Batch Scheduling Using Real Production-ready Software Tools. In *Proceedings of the 5th IBERGRID*, 2011.

[Man80] Benoit B. Mandelbrot. Fractal Aspects of the Iteration of z $\rightarrow$ Λz (1-z) for Complex Λ and z. *Journal of Annals of the New York Academy of Sciences*, 357(1):249–259, 1980.

[MGG+17] Michael Mercier, David Glesser, Yiannis Georgiou, and Olivier Richard. Big Data and HPC collocation: Using HPC Idle Resources for Big Data Analytics. In *Proceedings of International Conference on Big Data*, 2017, pages 347–352.

[MC20] Ali Mohammed and Florina M. Ciorba. SimAS: A Simulation-assisted Approach for the Scheduling Algorithm Selection Under Perturbations. *Concurrency and Computation: Practice and Experience*, 32(15):e5648, 2020.

[MEC18] Ali Mohammed, Ahmed Eleliemy, and Florina M. Ciorba. Performance Reproduction and Prediction of Selected Dynamic Loop Scheduling Experiments. In *Proceedings of the international confer-*

ence on high performance computing & simulation, 2018, pages 398–405.

[MEC+18] Ali Mohammed, Ahmed Eleliemy, Florina M. Ciorba, Franziska Kasielke, and Ioana Banicescu. Experimental verification and Analysis of Dynamic Loop Scheduling in Scientific Applications. In *Proceedings of the international symposium on parallel and distributed computing*, 2018, pages 141–148.

[MEC+20] Ali Mohammed, Ahmed Eleliemy, Florina M. Ciorba, Franziska Kasielke, and Ioana Banicescu. An Approach for Realistically Simulating the Performance of Scientific Applications on high Performance Computing Systems. *Journal of Future Generation Computer Systems*, 111:617–633, 2020.

[Moo+65] Gordon E. Moore et al. Cramming More Components onto Integrated Circuits. 1965.

[Nag93] Wolfgang E. Nagel. A Distributed Scheduler System for Multiprocessor Computers with Shared memory: Investigations into the Scheduling of Parallel Programs. PhD thesis. RWTH Aachen, 1993, pages 1, 174.

[Ope20] OpenMP Architecture Review Board. OpenMP Application Programming Interface. `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`. [Online; accessed 10 September 2020]. 2020.

[PGW+17] Arnab K. Paul, Arpit Goyal, Feiyi Wang, Sarp Oral, Ali R. Butt, Michael J. Brim, and Sangeetha B. Srinivasa. I/o Load Balancing for Big Data HPC Applications. In *Proceedings of the international conference on big data (big data)*, 2017, pages 233–242.

[PPC86] Tang Peiyi and Yew Pen-Chung. Processor Self-scheduling for Multiple-nested Parallel Loops. In *Proceedings of the International Conference on Parallel Processing*, 1986, pages 528–535.

[Pfi01] Gregory F. Pfister. An introduction to the Infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[PD97] Teebu Philip and Chita R Das. Evaluation of Loop Scheduling Algorithms on Distributed Memory Systems. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, 1997, pages 76–94.

[PK87] Constantine D. Polychronopoulos and David J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, 100(12):1425–1439, 1987.

[Pra16] Suraj Prabhakaran. Dynamic Resource Management and Job Scheduling for High Performance Computing. PhD thesis. Technische Universität Darmstadt, 2016.

[PIR+14] Suraj Prabhakaran, Mohsin Iqbal, Sebastian Rinke, Christian Windisch, and Felix Wolf. A Batch System With Fair Scheduling for Evolving Applications. In *Proceedings of the International Conference on Parallel Processing*, 2014, pages 351–360.

[PNR+15] Suraj Prabhakaran, Marcel Neumann, Sebastian Rinke, Felix Wolf, Abhishek Gupta, and Laxmikant V Kale. A Batch System With Efficient Adaptive Scheduling for Malleable and Evolving Applications. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2015, pages 429–438.

[RBA+18] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, et al. Scalable System Scheduling for HPC and Big Data. *Journal of Parallel and Distributed Computing*, 111:76–92, 2018.

[RCA+06] I. Riakotakis, F. M. Ciorba, T. Andronikos, and G. Papakonstantinou. Self-adapting Scheduling for Tasks with Dependencies in Stochastic Environments. In *Proceedings of the International Conference on Cluster Computing*, 2006, pages 1–8.

[Rod17] Gonzalo P. Rodrigo. HPC Scheduling in a Brave New World. PhD thesis. Umeå Universitet, 2017.

[ST07] K. Y. Sanbonmatsu and C. S. Tung. High Performance Computing in Biology: Multimillion Atom Simulations of Nanoscale Systems. *Journal of Structural Biology*, 157(3):470–480, 2007.

[SV09] H. A. Sanjay and Sathish S Vadhiyar. A Strategy for Scheduling Tightly Coupled Parallel Applications on Clusters. *Concurrency and Computation: Practice and Experience*, 21(18):2491–2517, 2009.

[SLG+14] Osman Sarood, Akhil Langer, Abhishek Gupta, and Laxmikant Kale. Maximizing Throughput of Overprovisioned HPC Data Centers Under a Strict Power Budget. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pages 807–818.

[Sch97] Robert R. Schaller. Moore's Law: Past, Present and Future. *IEEE Spectrum*, 34(6):52–59, 1997.

[Sch20] SchedMD. SchedMD: Slurm Development and Support. `https://slurm.schedmd.com/pdfs/schedmd_slurm_data.pdf`. [Online; accessed 26 August 2020]. 2020.

[SWZ+16] Hongzhang Shan, Samuel Williams, Yili Zheng, Weiqun Zhang, Bei Wang, Stephane Ethier, and Zhengji Zhao. Experiences of Applying One-sided Communication to Nearest-neighbor Communication. In *Proceedings of the Workshop on PGAS Applications*, 2016, pages 17–24.

[SYT07] Wen-Chung Shih, Chao-Tung Yang, and Shian-Shyong Tseng. A Performance-based Parallel Loop Scheduling on Grid Environments. *Journal of Supercomputing*, 41(3):247–267, 2007.

[SDI+18] Nikolay A. Simakov, Robert L. DeLeon, Martins D. Innus, Matthew D. Jones, Joseph P. White, Steven M. Gallo, Abani K. Patra, and Thomas R. Furlani. Slurm Simulator: Improving Slurm Scheduler Performance on Large HPC Systems by Utilization of multiple Controllers and Node Sharing. In, *Proceedings of the Practice and Experience on Advanced Research Computing*, pages 1–8, 2018.

[SIJ+17] Nikolay A. Simakov, Martins D. Innus, Matthew D. Jones, Robert L. DeLeon, Joseph P. White, Steven M. Gallo, Abani K. Patra, and Thomas R. Furlani. A Slurm Simulator: Implementation and Parametric Analysis. In *Proceedings of the International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, 2017, pages 197–217.

[SB01] Lorna Smith and Mark Bull. Development of mixed mode MPI/OpenMP applications. *Scientific Programming*, 9(2-3):83–98, 2001.

[SBC+11] Srishti Srivastava, Ioana Banicescu, Florina M. Ciorba, and Wolfgang E. Nagel. Enhancing the Functionality of a GridSim-based Scheduler for Effective Use with Large-Scale Scientific Applications. In *Proceedings of the International Symposium on Parallel and Distributed Computing*, 2011, pages 86–93.

[STL+15] Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, and Jean-François Méhaut. Faithful Performance Prediction of a Dynamic Task-based Runtime system for Heterogeneous Multi-core Architectures. *Concurrency and Computation: Practice and Experience*, 27(16):4075–4090, 2015.

[SVP+10] Tony Stöcker, Kaveh Vahedipour, Daniel Pflugfelder, and N Jon Shah. High-performance Computing MRI Simulations. *Magnetic resonance in medicine*, 64(1):186–193, 2010.

[SBS+13] Nitin Sukhija, Ioana Banicescu, Srishti Srivastava, and Florina M. Ciorba. Evaluating the Flexibility of Dynamic Loop Scheduling on Heterogeneous Systems in the Presence of Fluctuating Load Using SimGrid. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshops*, 2013, pages 1429–1438.

[TES+19] A. Totounferoush, N. Ebrahimi Pour, J. Schröder, S. Roller, and M. Mehl. A New Load Balancing Approach for Coupled Multi-Physics Simulations. In *Proceedings of the international parallel and distributed processing symposium workshops (ipdpsw)*, 2019, pages 676–682.

[TN93] Ten H. Tzen and Lionel M. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, 1993.

[Ull75] J. D. Ullman. NP-complete Scheduling Problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

[VM02] Jeffrey S. Vetter and Frank Mueller. Communication Characteristics of Large-scale Scientific Applications for Contemporary Cluster Architectures. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002, 10–pp.

[WLD+10] K. Wang, G. Lavoué, F. Denis, A. Baskurt, and X. He. A Benchmark for 3D Mesh Watermarking. In *Proceedings of the 9th IEEE International Conference on Shape Modeling and Applications*, 2010, pages 231–235.

[WOK+00a] Adrian T. Wong, Leonid Oliker, William Kramer, Teresa L. Kaltz, and David H. Bailey. ESP: A System Utilization Benchmark. In *Proceedings of the International Conference on Supercomputing*, 2000, pages 15–19.

[WOK+00b] Adrian T. Wong, Leonid Oliker, William Kramer, Teresa L. Kaltz, and David H. Bailey. System Utilization Benchmark on the Cray T3E and IBM SP. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, 2000, pages 56–67.

[WYL+12] Chao-Chin Wu, Chao-Tung Yang, Kuan-Chou Lai, and Po-Hsun Chiu. Designing Parallel Loop Self-scheduling Schemes Using the Hybrid MPI and OpenMP Programming Model for Multi-core Grid Systems. *Journal of Supercomputing*, 59(1):42–60, 2012.

[Xha10] Xhafa, Fatos and Abraham, Ajith. Computational Models and Heuristic Methods for Grid Scheduling Problems. *Future Generation Computer Systems*, 26(4):608–621, 2010.

[YJG03] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple Linux Utility for Resource Management. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, 2003, pages 44–60.

[ZBG16] Xin Zhao, Pavan Balaji, and William Gropp. Scalability Challenges in Current MPI One-Sided Implementations. In *Proceedings of International Symposium on Parallel and Distributed Computing*, 2016, pages 38–47.

[ZG16] Huan Zhou and José Gracia. Asynchronous Progress Design for a MPI-based PGAS One-sided Communication System. In *Proceedings of the International Conference on Parallel and Distributed Systems*, 2016, pages 999–1006.

Index