Job Scheduling for HPC Clusters: Constraint Programming vs. Backfilling Approaches

Alexander V. Goponenko
Kenneth Lamar
alexander.goponenko@ucf.edu
kenneth.lamar@ucf.edu
University of Central Florida
Orlando, Florida, USA

Benjamin A. Allan
James M. Brandt
baallan@sandia.gov
brandt@sandia.gov
Sandia National Laboratories
Albuquerque, New Mexico, USA

Damian Dechev
damian.dechev@ucf.edu
University of Central Florida
Orlando, Florida, USA

ABSTRACT

Scheduling problems are naturally formulated in terms of Constraint Programming (CP), yet the application of CP-based approaches for job scheduling on High-Performance Computing (HPC) clusters remains underexplored. This study aims to bridge this gap by analyzing the scheduling of diverse real workload traces using IBM ILOG CP Optimizer. The analysis considers not just the basic metrics Average Bounded Slowdown and Average Response Time, but also Area-Weighted Average Response Time and Level-2 Priority-Weighted Specific Time, which measure packing efficiency and fairness. For each workload trace and metric combination, schedules produced through optimizing the metric with CP Optimizer are compared against schedules generated by the variants of the list scheduling with backfilling that are most suitable for the metric. The CP-based scheduling improves scheduling quality in most of the examined cases. The analysis of the metrics that represent different scheduling goals uncovers several non-trivial insights. Presently, CP Optimizer still encounters scalability issues with large wait queues and non-linear objective functions. Nonetheless, it often demonstrates improvements in packing efficiency, which make CP techniques attractive, particularly in scenarios where high-maintenance clusters run a moderate number of large, rigid, well-characterized jobs.

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC (NTESS), a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration (DOE/NNSA) under contract DE-NA0003525. This written work is authored by an employee of NTESS. The employee, not NTESS, owns the right, title and interest in and to the written work and is responsible for its contents. Any subjective views or opinions that might be expressed in the written work do not necessarily represent the views of the U.S. Government. The publisher acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this written work or allow others to do so, for U.S. Government purposes. The DOE will provide public access to results of federally sponsored research in accordance with the DOE Public Access Plan.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

CCS CONCEPTS

• **Theory of computation** → *Scheduling algorithms*; • **Applied computing** → Operations research; Enterprise computing; • **Computing methodologies** → **Planning and scheduling**; Optimization algorithms.

KEYWORDS

High-performance computing, parallel job scheduling, schedule quality, constraint programming, CP Optimizer, packing efficiency, fairness, weighted average response time

ACM Reference Format:
Alexander V. Goponenko, Kenneth Lamar, Benjamin A. Allan, James M. Brandt, and Damian Dechev. 2024. Job Scheduling for HPC Clusters: Constraint Programming vs. Backfilling Approaches. In . ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

A High-Performance Computing (HPC) cluster is an assemblage of many high-end computer servers (called nodes) networked by a fast interconnect. The purpose of HPC clusters is to handle intensive computations that are not feasible on a single computer. A core component of the cluster management software is a scheduler, which decides the order of execution of queued jobs and the distribution of cluster resources between the running jobs. Since job throughput on HPC clusters can be very large and the scheduling problem is NP-hard [10], the schedulers traditionally resort to using approximate algorithms, usually based on list scheduling with backfilling [29, 30, 33] (Section 2.4). These algorithms are fast but produce sub-optimal schedules. Their efficiency is further reduced when applied to modern HPC clusters, which often feature several types of resources and heterogeneous nodes [13, 26]. At the same time, the increasing prevalence of high-intensity computing and its growing economic and environmental impact call for improving the utilization of HPC infrastructure and, thus, necessitate more efficient schedulers [11, 12]. The interest in more elaborate scheduling approaches also emerges from the relevant fields of grid and cloud computing [25, 51, 54].

Constraint Programming (CP) is a paradigm for solving combinatorial search problems [43]. It is widely used for scheduling in Operational Research but is rarely known in the HPC community. This technique lets one define a set of variables, declare several constraints that these variables must satisfy simultaneously, and use a CP solver as a black box to find values for the variables that satisfy the constraints, aiming to optimize some objective function.

The recent development of CP solvers has significantly improved their performance and simplified the process of defining scheduling problems in terms of CP [3]. Therefore, a CP approach could be a viable alternative to the list scheduling for HPC job scheduling. It could be employed in production schedulers or used in research to generate close-to-optimal schedules to compare them with the results of other schedulers. Recently, several attempts to employ CP for HPC job scheduling were reported [4, 6, 18, 19], but the evaluation of the results was inconclusive, mainly because the utilized scheduling quality metrics do not measure the accomplishments of such scheduling goals as packing efficiency and fairness (Section 3.3). At the same time, recent development suggests that Area-Weighted Response Time (*AWF*, also referred to as Area-Weighted Flow Time) is an accurate metric of packing efficiency in this scheduling problem [24, 45], while metrics called Priority-Weighted Specific Response Time ($P^{\alpha}SF$) combine contributions of the scheduler's fairness and packing efficiency [24]. Fairness and packing efficiency are important aspects of scheduling quality but are poorly covered by the other metrics (Section 3.1). Thus, *AWF* and $P^{\alpha}SF$ create opportunities for more comprehensive comparison of different scheduling strategies, such as the list scheduling algorithms and the CP approach. Furthermore, these metrics are easy to calculate but sensitive to minor schedule changes, which makes them good prototypes of objective functions for CP optimization. In this publication, we seize upon these emerging opportunities to impartially compare the traditional list-based schedulers with the existing readily available CP tools, with the hope of increasing awareness of these tools.

1.1 Contributions

We present a comprehensive evaluation of CP-based scheduling, utilizing a diverse set of real workload traces from various production systems. We separately examine the optimization of four scheduling quality metrics (Section 2.1): Average Bounded Slowdown (*BSLD*), Average Response Time (*AF*), Area-Weighted Average Response Time (*AWF*), and Level-2 Priority-Weighted Specific Time (P^2SF). This highly informative set of metrics covers several important aspects of scheduling: packing efficiency, fairness, and performance perceived by an individual user [24]. For each metric, we precisely define the scheduling goal in terms of the metric under consideration and compare the most applicable for the metric list scheduling algorithms with the CP-based optimization of an objective function directly based on the metric. This study surpasses the prior research, which considered limited test cases and assessed the scheduling quality using metrics that did not align with the objectives of the CP optimization, or with the design of the stock PBS scheduler being compared to the CP approach. Our state-of-the-art analysis provides readers with an accurate understanding of the potential benefits of applying Constraint Programming to HPC scheduling.

In addition to introducing first CP-based schedulers that address *AWF* and $P^{\alpha}SF$, and thus, packing efficiency and fairness, the outcomes of this research make two significant contributions to job scheduler development. Firstly, our study, by improving the targeted metrics over the previous best results, uncovers opportunities for improvement of the scheduling algorithms and offers other insights to guide further scheduler development. Since we use IBM ILOG CP Optimizer, a mature out-the-box product for scheduling-related problems [35], our methodology can serve as a reference example for future investigations. Secondly, our findings demonstrate the capabilities of modern CP optimization tools and help understand the feasibility of employing CP optimization for practical applications in HPC job scheduling.

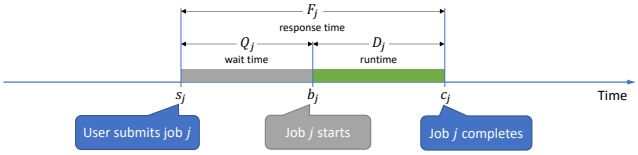


Figure 1: Diagram of job lifetime showing the notation used herein.

In Section 2 we describe this particular scheduling problem, its CP representation, the list-based scheduling approach that serves as an alternative in our study, the scheduling quality metrics used for the analysis, and key elements of our methodology. In Section 3 we summarize relevant works that evaluated HPC cluster scheduling algorithms, identify best known algorithms for each of the four metrics, discuss metrics excluded from our analysis, and review previous research of CP-based scheduling. Section 4 covers the remaining details of our study and is followed by a discussion of the results in Section 5 and a summary of the findings in Section 6.

2 BACKGROUND AND FUNDAMENTALS OF OUR METHODOLOGY

HPC systems simultaneously process many jobs from different users. Formally, job j arrives at submit time s_j (also called arrival time). When more jobs arrive than can be dispatched to the cluster, they end up in a wait queue. Scheduling algorithms determine the order of starting such jobs, aiming at best performance according to an objective determined by the cluster administration. After spending wait time Q_j (sometimes called queue time) in the queue, the job starts (at time b_j) and completes after additional runtime D_j at time c_j. The lifetime of a job is depicted in Figure 1. The total time elapsed between the submit time and the completion time is the job's response time F_j (also called flow time or turnaround time):

$$F_j = Q_j + D_j \,.$$

Users specify requirements for the jobs, including the number of nodes r_j required and the guaranteed runtime timelimit L_j (we do not consider job preemption). We assume that the number of nodes is the only resource requirement of the job. We also assume that the jobs are rigid (that is, r_j and D_j remains the same regardless of how the jobs are scheduled), all nodes in the cluster are identical, and the total number of nodes R in the cluster does not change. HPC clusters often do not satisfy these assumptions and may impose other conditions, but we ignore them for the sake of clarity of the following analysis.

Jobs cannot use more resources than available, which leads to the following constraint:

$$\forall\, t : \sum_{j \in \mathcal{J}} \text{pulse}_j(t) \le R, \tag{1}$$

where t represents a moment in time, $\mathcal{J}$ is the set of jobs in the schedule and

$$\text{pulse}_j(t) = \begin{cases} r_j, & b_j \le t < c_j \\ 0, & \text{otherwise} \end{cases}.$$

Another set of constraints signifies that no job can start before it was submitted:

$$\forall j : s_j \le b_j . \tag{2}$$

The goal of the scheduler is to identify values of b_j that satisfy these constraints aiming at best performance according to predefined goals. If the achievement of the goals is measurable with a metric, the scheduling problem becomes an optimization problem, the objective function of which corresponds to the metric.

2.1 Scheduling Quality Metrics

In this work, we investigate four metrics that represent different scheduling goals: Average Bounded Slowdown (*BSLD*), Average Response Time (*AF*), Area-Weighted Average Response Time (*AWF*), and Level-2 Priority-Weighted Specific Time (P^2SF).

Average Bounded Slowdown (*BSLD*) reflects the idea that it is reasonable for jobs with longer runtime to spend more time waiting in the queue:

$$BSLD(\mathcal{J}) = \frac{1}{|\mathcal{J}|} \sum_{j \in \mathcal{J}} \max\left(1, \frac{F_j}{\max\left(D_j, k\right)}\right), \tag{3}$$

where k is some lower bound introduced to reduce the disproportionately large effect of very small jobs [15]. *BSLD* becomes the regular Slowdown metric when $k = 0$. Typically, k is set to 10 s [21, 50], and we use this value in our analysis. Although the best value of k may be an open question, 10 s is comparable to user tolerable waiting time [39]. The workloads we study herein (Section 4) have an insignificant number of jobs that trigger this bound. Yet, even with a reasonable value for k, *BSLD* is often so strongly influenced by moderate delays of small jobs that it almost completely ignores the quality of scheduling of long and resource-demanding jobs, the very jobs HPC clusters are meant to accommodate. *BSLD* can be trivially improved by starting smaller jobs first [24]. Nevertheless, *BSLD* is widely used for analysis of HPC job scheduling.

Another, classical scheduling quality metric is Average Response Time (*AF*):

$$AF(\mathcal{J}) = \frac{1}{|\mathcal{J}|} \sum_{j \in \mathcal{J}} F_j . \tag{4}$$

For rigid jobs, *AF* can be considered equivalent to several linearly related metrics, such as Total Wait Time $\sum Q_j$. All such metrics improve when smaller jobs are given priority (although not as much as *BSLD*) as they do not recognize that computing a larger job creates more value.

Average Response Time *AWF* accounts for job sizes:

$$AWF(\mathcal{J}) = \frac{\sum r_j D_j F_j}{\sum r_j D_j}, \tag{5}$$

where the weight equals the job area $r_j D_j$. Such weights do not encourage any preference for jobs based on their size, making *AWF* a good unbiased measure of a schedule's job packing efficiency [24, 45]. Improving packing efficiency is regarded as reducing resource fragmentation and making the resources available for use by future jobs [53]. *AWF* measures precisely this property of a schedule because it provides no gain in splitting a large job into several smaller jobs processed in the same time slot but improves whenever some computation moves to earlier time [44].

Finally, Level-α Priority-Weighted Specific Response Time ($P^\alpha SF$) reflects both a schedule's job packing efficiency and fairness without a job size bias. $P^\alpha SF$ originates from splitting jobs into infinitely small "units of computation" and assuming the weight of a unit is a power function of the unit's specific response time [24]:

$$P^\alpha SF(\mathcal{J}) = \frac{\alpha+1}{\alpha+2} \frac{\sum r_j \left({F_j}^{\alpha+2} - {Q_j}^{\alpha+2}\right)}{\sum r_j \left({F_j}^{\alpha+1} - {Q_j}^{\alpha+1}\right)}. \tag{6}$$

The parameter α controls the determines the balance between fairness and packing efficiency contributions. A higher α increases the metric's sensitivity to job fairness. The most practical range is $1 \le \alpha \le 3$ [24]. In the forthcoming analysis, we use P^2SF, setting α to the midpoint of its practical range.

2.2 "On-line" Scheduling Mode

For any of these metrics, if the scheduler has complete information about every job, it can directly employ the metric as the objective function and optimize the entire schedule at once. However, the HPC cluster scheduler has no information about jobs until they are submitted. It must operate in "on-line" mode [42], meaning that at any time it must make decisions using only currently available information, but it can later reevaluate its scheduling decisions for the jobs that are not yet started. In practice, the scheduler usually runs at regular intervals. In our analysis, we adhere to the common practice in scheduling simulation, starting a scheduling round whenever there is a change in available information, that is, every time a job arrives or terminates [4, 19, 21, 24, 50]. This approach removes the dependence of the scheduling quality on the interval between the scheduling rounds. Furthermore, the scheduler typically only learns about the job runtime once the jobs have finished. Consequently, it cannot accurately calculate the metric value of a prospective schedule; the value of the objective function at a scheduling round (local objective function) may differ from the value of the metric of the resulting schedule (global objective function) even if the same formula is used.

2.3 Constraint Programming Approach

The "off-line" scheduling (when the scheduler has all information, including the jobs that are yet to arrive) can be implemented straightforwardly in a CP solver by defining b_j as the set of variables, (1) and (2) as the constraints, and one of (3)–(6) as the objective function. Implementation of a scheduling round for the "on-line" scheduling problem is also trivial: the scheduler just must use runtime estimates (referred to as E_j) instead of actual job runtimes D_j and take into account only the jobs that are currently running or waiting in the queue. The resource constraint (1) becomes

$$\forall t : \sum_{j \in \mathcal{Q} \bigcup \mathcal{R}} \text{pulse}_j(t) \le R, \tag{7}$$

where $\mathcal{Q}$ is the set of jobs in the queue, $\mathcal{R}$ is the set of running jobs, and, given that job runtime D_j is approximated by an estimate E_j,

$$\text{pulse}_j(t) = \begin{cases} r_j, & b_j \le t < b_j + E_j \\ 0, & \text{otherwise} \end{cases}.$$

Equation (2) transforms into

$$\forall\, j \in \mathcal{Q} : b_j \ge t_{\text{now}}\,, \tag{8}$$

where t_{now} is the time of the round. The scheduler solves, with constraints (7) and (8), the optimization problem

$$\underset{\{\, b_j \mid j \in \mathcal{Q} \,\}}{\arg\min} \; \mathit{Objective}(\mathcal{Q})\,, \tag{9}$$

(where *Objective* is either *BSLD*, *AF*, *AWF*, or P^2SF calculated using E_j in place of D_j) and dispatches to the cluster the jobs scheduled to start immediately ($b_j = t_{\text{now}}$). The other waiting jobs remain in the queue until the next scheduling round.

2.4 List Scheduling Approach

The practical implementations of HPC cluster job scheduling are usually based on a variant of list scheduling. Algorithm 1 outlines the majority of the variants of the scheduling round of such algorithms.

Algorithm 1: General list scheduling algorithm

```
1  Arrange waiting jobs in Queue according to some rule.
2  for j ∈ Queue do
3      if job j can start right away then
4          Start job j.
5      else
6          switch Option
7              case STRICT
8                  Stop this scheduling round.
9              case GREEDY
10                 Skip job j until the next scheduling round.
11             case BACKFILL
12                 b_j ← earliest time job j can run.
13                 Reserve resources r_j for interval (b_j, b_j + E_j).
```

During the round, the scheduler transforms the job wait queue into an ordered list (Line 1) and processes the jobs according to that order. The order of the jobs is one of the two pivotal factors that determines the behavior of the algorithm. The second factor is the option chosen when a job being considered cannot start immediately, for example because the job requires more resources than currently available (Line 6). The *strict* option (Line 7) terminates the scheduling round, effectively pausing the scheduler until the head job can start [20]. This algorithm can be optimal if it is given an optimal job order. However, in reality it often leads to poor packing efficiency. The *greedy* scheduling option (Line 9) skips the job and tries the next job [52]. It may improve resource utilization in practice (where the job queue order is often sub-optimal) over the *strict* option but can prevent attaining an optimal schedule with an optimal job order and cause job starvation (where a job with high resource requirements is indefinitely postponed while other jobs are scheduled). Finally, the *backfill* option (Line 11) reserves resources at the earliest time the job can run without violating already existing reservations. The reservation makes the resources unavailable (for the jobs later in the queue) for the time period the job is scheduled to run, but another job can start earlier if its resource requirements can still be satisfied. This technique is called "backfilling" [38]. If job runtimes are known during the scheduling, no job is delayed by a later job and the resource utilization may improve without violating job priorities.[1] In the real world, job runtime is often approximated and complying with job priorities is not guaranteed. This option is also the most computationally intensive due to the complexity of keeping track of reservations and finding the earliest time a job can run, but as discussed in Section 3.2, it, among the three variants, attains best results for all four metrics if appropriate job queue orders are used [24]. Therefore, we use the list algorithms with *backfill* option as a baseline to compare with the results of the CP approach.

3 RELATED WORK

3.1 Alternative Metrics of Scheduling Quality

Many metrics of "off-line" scheduling, such as *Makespan*:

$$\mathit{Makespan}(\mathcal{J}) \;=\; \max_{j\in\mathcal{J}}\left(c_j\right) - \min_{j\in\mathcal{J}}\left(s_j\right)$$

and *Utilization*:

$$\mathit{Utilization}(\mathcal{J}) \;=\; \frac{\sum\limits_{j\in\mathcal{J}} r_j D_j}{\left(\max\limits_{j\in\mathcal{J}}\left(c_j\right) - \min\limits_{j\in\mathcal{J}}\left(s_j\right)\right)\times R}\,,$$

are not useful to evaluate the "on-line" scheduling of jobs [15]. *Makespan* and *Utilization* are determined by the completion time of the last job (that is, $\max c_j$) and rarely represent HPC cluster scheduling quality if job submit times are independent of scheduling decisions. Use of *Utilization* to evaluate packing efficiency of job schedulers in an HPC cluster is considered a pitfall in the methodology [17]. As an alternative, Boëzennec et al. [5] suggested evaluating the packing efficiency by measuring the variance of *Utilization* or by studying the density function of *Utilization*. However, these new metrics only measure how the utilization varies over time and do not reflect the scheduler's ability to pack the jobs to earlier times. Even though the variance of *Utilization* may correlate with packing efficiency for the existing algorithms, it can misrepresent the scheduling quality of future, less "common sense" approaches and is not suitable as an objective function for optimization. Leung et al. [37] suggested Loss of Capacity (*LOC*) as a metric of packing efficiency. This metric corresponds to the fraction of processor cycles that are idle when waiting jobs could have used them:

$$\mathit{LOC}(\mathcal{J}) \;=\; \frac{\int\limits_{\min\limits_{j\in\mathcal{J}} s_j}^{\max\limits_{j\in\mathcal{J}} c_j} \min\left(\sum\limits_{j\,:\,s_j<t<b_j} r_j\,,\; R \;-\; \sum\limits_{j\,:\,b_j<t<c_j} r_j\right) dt}{\left(\max\limits_{j\in\mathcal{J}}\left(c_j\right) - \min\limits_{j\in\mathcal{J}}\left(s_j\right)\right)\times R}\,.$$

[1] Also, the scheduler that knows job runtimes will attain an optimal schedule when given an optimal job order (although no job will be backfilled).

LOC is a less reliable metric of job packing efficiency than *AWF*. While the job wait queue is long, *LOC* equals *Utilization* and ignores local improvements. When the queue is short, *LOC* may encourage scheduling shorter jobs earlier. *LOC* is also more cumbersome to use as a local objective function.

Several approaches were suggested to measure scheduling fairness. Maximum wait time [8] and similar metrics have limited use as they depend on a small subset of the jobs and disregards all other scheduling decisions. L_p norm of the response time, that is, $\left(\sum F_j^p\right)^{1/p}$ [42], is similar to $P^\alpha SF$ but may encourage preferential treatment based on job size, while the weights of $P^\alpha SF$ are designed to prevent such bias. Metrics based on the concept of jobs' fair start times [37], in addition to other shortcomings [24], may be computationally intensive for use as objective functions. Due to the lack of suitable metrics, some publications introduce ad-hoc techniques to evaluate scheduling fairness, such as measuring performance separately for jobs of various categories ("long" and "short," etc.) [19, 31, 41, 47]. Results of such techniques are hard to interpret and compare because the criteria for job categories are not well defined and often arbitrary, which also precludes the development of objective functions based on such approaches.

3.2 Metrics-Focused List Scheduling Variants

BSLD and *AF*[2] are the most widely used metrics of HPC job scheduling [15, 18, 19, 21, 41, 50]. These metrics can be trivially improved by giving preference to smaller jobs. Multiple publications report improvement of these metrics by incorporating Shortest Job First (SJF) order (that is, the wait queue is arranged in ascending order based on estimated job runtime E_j) into list-based algorithms [14, 41, 46, 50]. SJF is an appropriate heuristic when all jobs require same number of nodes. When jobs require different numbers of nodes, additional improvement of *BSLD* and *AF* is expected if some degree of preference is given also to jobs that use less resources. Indeed, Smallest Area First (SAF) order, where the job area is $r_j E_j$, usually outperforms SJF [7, 24]. Quite surprisingly, even with SAF and SJF orders, the *greedy* scheduling option often results in inferior *BSLD* and *AF* values than the *backfill* option, although few opportunities for backfill may be expected when smaller jobs are already considered before larger jobs [24].

Ordering the job wait queue by SAF or SJF in list-based algorithms, while improving *BSLD* and *AF*, typically results in degradation of packing efficiency, as indicated by the values of *AWF* [24]. Bin-packing research suggests that packing efficiency improves by sorting the wait queue in descending order of the job resource requirement R_j [45]. However, Largest Area First (LAF) order and the *backfill* option often attain even better values of *AWF* (and, thus, the packing efficiency) [24], probably because this algorithm increases the chance that small jobs are available to fill the holes that cannot accommodate the larger jobs.

The use of LAF, SAF, or SJF orders in list scheduling negatively affects fairness and may cause starvation. While starvation can be reduced by modifying the job ordering [2, 7, 22, 47], the backfill option with the original FCFS (that is, earliest submit time first) order still remains the "gold standard" of fair scheduling and attains the lowest values of P^2SF [24].

[2] Including metrics that are equivalent to *AF* in the case of rigid jobs.

3.3 Constraint Programming Application in HPC Job Scheduling

Bartolini et al. [4] evaluated CP Optimizer for HPC cluster job scheduling on workload traces from the EURORA cluster at the CINECA center. They measured scheduling quality using *Utilization* and Average Wait Time (an equivalent of *AF* in case of rigid jobs) and noted that *Makespan* (with modifications to account for other objectives) is the local objective function that produced the best scheduling quality. Although indicating improvements over the standard algorithm of PBS [40], the conclusions of the evaluation are uncertain because the study considered only three traces with 1480 jobs in total. Additionally, the large observed improvement in *Utilization* suggests that the conditions of the simulation may be incomparable with the real-world scheduling, as *Utilization* in "on-line" scheduling usually measures the offered load with little influence of the scheduler [15]. Bridi et al. [6] reported a more elaborate comparison of a CP-based scheduler to the PBS scheduler on the same EURORA cluster. For workloads that did not result in long wait queues, the CP scheduler demonstrated improvement to a variant[3] of *AF* when the same metric was used as the local objective function. The results, although limited to only one metric and workload traces of 700 jobs or less, established use cases for CP-based job scheduling and encourage its further development.

Galleguillos et al. [19] used a workload trace of 372 321 jobs from the EURORA cluster to compare a "hybrid" CP-based scheduler to list-based algorithms similar to Algorithm 1 with the *strict* option. Instead of using CP Optimizer, they employed Google's open-source OR-Tools [23]. Although the local objective function was equivalent to *AF*, the resulting schedules were compared using Slowdown. Nevertheless, the CP-based scheduler attained better values of Slowdown than the list-based algorithm with SJF order. However, the analysis did not cover the best possible list-based algorithm. Even when targeting Slowdown and using SJF order, the *backfill* option of the list-based algorithm outperforms the *strict* option [24]. Therefore, the outcome may differ if the CP approach is compared to the list-based algorithms with the *backfill* option.

OR-Tools could be a viable alternative to IBM ILOG CP Optimizer. We use CP Optimizer because it is well maintained and documented, is often praised for its performance, and likely has a lower entry barrier. Furthermore, our preliminary evaluations (not shown), while not sufficient to allow a decisive conclusion, suggest that CP Optimizer would outperform OR-Tools in our experimental conditions. Da Col and Teppan [9] also identified that CP Optimizer was more suitable for scheduling problems compared to OR-Tools. Thus, we do not consider OR-Tools at this stage. We also do not consider meta-heuristic-based optimization techniques, although some of them may be viable alternatives to the CP approach. Some examples of such promising techniques include slack-based priority backfilling [48], Tabu search [32], genetic algorithms [1], simulated annealing [55], and random search [31]. Novel approaches to job scheduling in Cloud computing [25, 51] may also find applications in HPC job scheduling. Although comparing these approaches is outside of the scope of this work, our analysis may serve as an example for their future evaluations.

[3] They used weighted wait time, where weights were determined by the queue to which the job was submitted [6].

Table 1: Characteristics of the workload traces used herein.

	# CPU	# Jobs	Utilization	Year	Duration
KTH-SP2	100	28 k	70 %	1996	11 months
CTC-SP2	338	77 k	85 %	1996	11 months
SDSC-SP2	128	54 k	83 %	2000	24 months
SDSC-BLUE	1 152	233 k	77 %	2003	32 months
CEA-CURIE	93 312	296 k	62 %	2012	8 months
BW201911	22 636	92 k	68 %	2019	1 month

4 TECHNICAL APPROACH

In our study, we utilize simulations rather than experiments on a real system. This mitigates the influence of noise and variability unavoidable in real-world HPC clusters. By employing simulations, we can consistently schedule the same traces of jobs multiple times, enabling us to isolate cause-and-effect relationships and draw more accurate conclusions regarding the outcomes of the scheduling algorithms. We evaluate the algorithms through the scheduling of a variety of traces of real parallel workloads from production systems listed in Table 1. We obtained the workload *BW201911* from December 2019 TORQUE logs of the NCSA Blue Waters supercomputer [49]. The rest of the traces are the recommended "cleaned" versions from the Parallel Workload Archive [16]. This set of traces spans three decades and different machine sizes. Some traces have been studied in related work [21, 24, 50]. This makes the results obtained in this paper representative of a large variety of practical use cases.

We refrain from directly comparing the results of CP optimization with those of real-world HPC job schedulers due to the numerous conditions that influence the real-world schedulers but are omitted from our models for clarity, such as scheduling overheads, node failures, job placement intricacies, complexity of modern HPC systems, and an extensive array of features demanded by both users and administrators. We compare the CP-based algorithm to list-based algorithms that, as discussed in Section 3.2, are most suited for the studied metric. All these list-based algorithms employ Algorithm 1 with *backfill* option (Line 11) but differ by job ordering within the queue (Line 1). The algorithm that uses the original, FCFS order is labeled *List-REG*. The algorithms that use SJF, SAF, and LAF orders are referred to as *List-SJF*, *List-SAF*, and *List-LAF*. One more algorithm, *List-SPF*, utilizes Smallest Area-Runtime Product First (SPF) order, where area-runtime product is $r_j D_j{}^2$. It is discussed in Section 5.2.

CP-based scheduling algorithms solve optimization problems expressed by (7)–(9), as described in Section 2.3. The local objective function is either *BSLD* (using $k = 10$ s), *AF*, *AWF*, or P^2SF. The corresponding algorithms are *CP-BSLD*, *CP-AF*, *CP-AWF*, and *CP-P^2SF*. We also include an algorithm *CP-BSLD+*, which, as described in Section 5.2, makes sure that its "local" value of *BSLD* at any scheduling round is not higher than the lowest value attainable by the alternative list-based algorithms. Similarly, in Section 5.4 we consider an algorithm *CP-P^2SF+*, which makes sure that its "local" value of P^2SF at any scheduling round is not higher than the value attainable by *List-REG*.

The simulation of the scheduling process was performed by an expanded version of Predictsim [21, 24], an event-based parallel job scheduler simulator derived from PYSS [34]. Conceptually, the simulator replays job arrival events based on the information from a workload trace it is given. Any job arrival or completion event triggers a scheduling round (up to one round per a moment of time). At the end of the scheduling round, the simulator generates a job start event for each job scheduled to start immediately, which in turn causes a job completion event delayed by the job's runtime. The source code and additional details on simulation are available at GitHub.[4] The CP-based schedulers were implemented with the help of IBM ILOG CPLEX Optimization Studio 20.1. Since CP optimizations are not guaranteed to converge, the optimization time was limited to 20 s per round. If no solution that satisfies the constraints was found, the optimization time was extended to 40 s. If no solution was found still, the scheduling round was performed by *List-REG* algorithm, merely to continue the simulation.[5] The experiments ran on a machine with two Intel Xeon processors E5-2683 v4 (16 cores each) and 128 GiB RAM.

To get additional insights, we analyze, for *KTH-SP2* only, the "off-line" scheduling using (1) and (2) as constraints and one of (3)–(6) as an objective function. The "off-line" schedules are labeled *CP-OFF-BSLD*, *CP-OFF-AF*, *CP-OFF-AWF*, and *CP-OFF-P^2SF*. The optimization time of the "off-line" scheduling roughly equals the cumulative time of scheduling the same trace with the corresponding"on-line" CP scheduler.

In order to evaluate the performance of the methods without interference from runtime prediction error, we assume for most of the following analysis that job runtimes are known during the scheduling ($E_j = D_j$). In Section 5.5 we evaluate how the methods perform when the runtime is estimated to be the user-requested timelimit, that is, $E_j = L_j$.

5 EXPERIMENTAL RESULTS

In this Section we consider the performance of CP-based scheduling separately for each of the studied metrics, discuss insights our investigation uncovers, and evaluate possibilities of practical application of CP-based scheduling.

5.1 Area-Weighted Average Response Time

A well-known heuristic for optimizing weighted average response time with list scheduling is Highest Density First (HDF) [28]. For HPC cluster job scheduling, job density is the ratio of job weight to job area. Therefore, *AWF* is special among weighted average response time metrics because for *AWF*, from HDF perspective, every queue order is equivalent, since every job density in this case equals 1. However, as Figure 2 demonstrates, *List-LAF* improves *AWF* over *List-REG* by as much as 17% (for *SDSC-SP2*). This phenomenon demonstrates that the HDF heuristic does not account for packing efficiency. *CP-AWF* improves *AWF* further, by as much as 10% over *List-LAF* (for *CTC-SP2*). Optimizing the *KTH-SP2* schedule with *AWF* objective function all at once in "off-line" mode (*CP-OFF-AWF*) leads to the similar value of *AWF* than the result of

[4] https://github.com/algo74/predictsim/DEBS2024
[5] The fraction of scheduling rounds, in which the scheduler fell back to *List-REG*, is considered negligible.

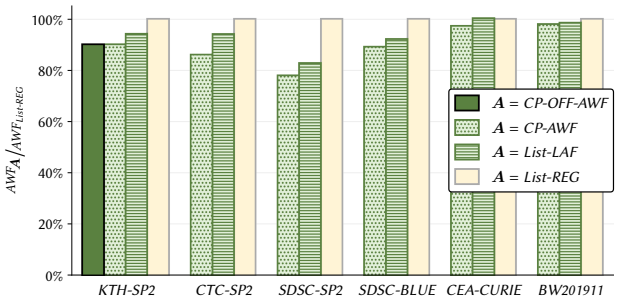


Figure 2: *AWF* of schedules of different workload traces produced by *CP-OFF-AWF*, *CP-AWF*, *List-LAF*, and *List-REG* assuming that job runtime is known to the scheduler. Values are shown as percentage of *AWF* for *List-REG*. Lower is better.

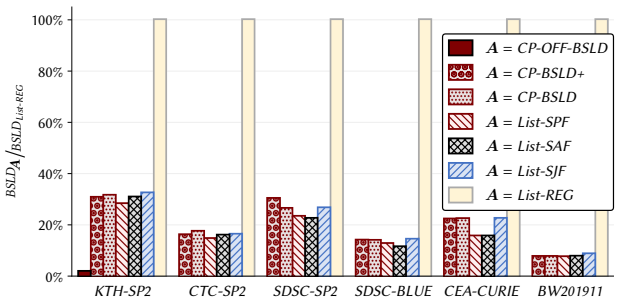


Figure 3: *BSLD* of schedules of different workload traces produced by *CP-OFF-BSLD*, *CP-BSLD+*, *CP-BSLD*, *List-SPF*, *List-SAF*, *List-SJF*, and *List-REG* assuming that job runtime is known to the scheduler. Values are shown as percentage of *BSLD* for *List-REG*. Lower is better.

the "on-line" optimization. The similar value of *AWF* may be due to a combination of a larger problem size in the "off-line" mode and a limited number of possibilities for improving the packing efficiency using the knowledge of the future jobs.

5.2 Average Bounded Slowdown

BSLD strongly improves when smaller jobs get priority during scheduling. Therefore, *List-SJF* significantly improves *BSLD* compared to *List-REG* (Figure 3). However, Shortest Job First (SJF) order does not take into account the resource requirements of the jobs. Smallest Area First (SAF) order often results in more jobs finishing earlier [7, 27]. Indeed, *List-SAF* outperforms *List-SJF* in terms of *BSLD* (Figure 3). Nevertheless, the equivalent of HDF for *BSLD*[6] is Smallest Area-Runtime Product First (SPF), where area-runtime product is $r_j D_j^2$. *List-SPF* does deliver lower than *List-SAF* values of *BSLD* on average.

Surprisingly, *CP-BSLD* often produces a worse *BSLD* than *List-SPF* and *List-SAF* are able to attain. At the same time, optimizing *KTH-SP2* schedule with *BSLD* objective function in "off-line" mode results in an order of magnitude reduction of *BSLD* (Figure 3). This could represent a theoretically possible scenario wherein the "on-line" scheduling algorithm, aimed at optimizing the global objective function (in our context, the metric value of the final schedule), achieves better results by employing a local objective function (the objective function being optimized at a single scheduling round) that differs from its global counterpart. Rather than replicating the global objective function by using the same formula limited to the data available at the scheduling round, the algorithm may benefit from adopting a different heuristic. In this case, minimizing *BSLD* using the information available during the scheduling rounds appears to be less effective for minimizing *BSLD* globally than the heuristics that constitute *List-SPF* and *List-SAF*. Possibly, an objective function that anticipates the arrival of smaller jobs in the future may lead to a further improvement in *BSLD*.

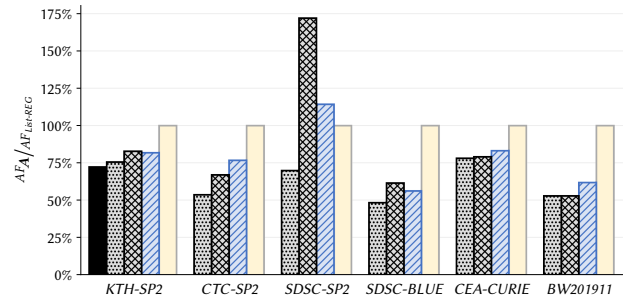
One could assume that the poor performance of *CP-BSLD* might be a result of an insufficient level of optimization, which causes many scheduling rounds having higher "local" values of *BSLD* than what *List-SPF* or *List-SAF* would produce. To check this possibility, we implemented *CP-BSLD+* scheduler that, at each scheduling round, uses *List-SPF*, *List-SAF*, and *List-SJF* to make three alternative scheduling plans, then further optimizes these plans using CP Optimizer, also makes a CP scheduling plan "from scratch", and finally selects for the round the plan that has the lowest *BSLD*. As seen in Figure 3, the schedules of all traces produced by *CP-BSLD+* still have higher *BSLD* than the schedules produced by *List-SPF*, even though *CP-BSLD+* is guaranteed to reach or improve at every scheduling round the value of *BSLD* that any algorithm evaluated can achieve with the same conditions. Therefore, *List-SPF* outperforms *CP-BSLD*, most likely, because *CP-BSLD* has a less effective local objective function and not because its optimization process is incomplete.

5.3 Average Response Time

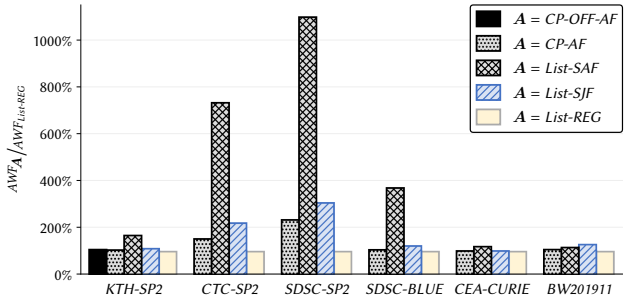
Similarly to *BSLD*, *AF* improves when smaller jobs get priority during scheduling, but not as dramatically. The HDF order for *AF* is equivalent to SAF. However, for some of the workloads, *List-SAF* resulted in higher values of *AF* than *List-SJF* (Figure 4a). For *SDSC-SP2*, *List-SAF* (and *List-SJF*) produced less optimal values of *AF* than even *List-REG*. The reason for this unreliable performance of *List-SAF* lies in its lower packing efficiency. Indeed, according to *AWF* (Figure 4b), *List-SAF* reduces the packing efficiency (compared to *List-REG*) more than *List-SJF*, while for *SDSC-SP2* the value of *AWF* of *List-SAF* is anomalously high. This anomaly is caused by the inability to handle the workload, as indicated by the continuous growth of the wait queue in Figure 5. It is worth noting that, while *AWF* clearly signal a problem, *Utilization*, which is often relied upon to detect such overload, inconspicuously decreases from 82.5% to 78.8%.

CP-AF better preserves the packing efficiency, prevents overload in the case of *SDSC-SP2*, and results in stable improvement of *AF* over *List-REG* for all workloads (Figure 4a). Optimizing *KTH-SP2* with the *AF* objective function in "off-line" mode results in an *AF* that is comparable with the value obtained with *CP-AF*. Thus, *AF* appears to be a good local objective function for the "on-line"

[6] *BSLD* may be assumed to be a variant of the weighted average response time if the clipping related to bound k in (3) has negligible effect.



(a) *AF*



(b) *AWF*

Figure 4: Values of *AF* and *AWF* of schedules of different workload traces produced by *CP-OFF-AF*, *CP-AF*, *List-SAF*, *List-SJF*, and *List-REG* assuming that job runtime is known to the scheduler. Values are shown as percentage of the value of the same metric for *List-REG*. Lower is better.

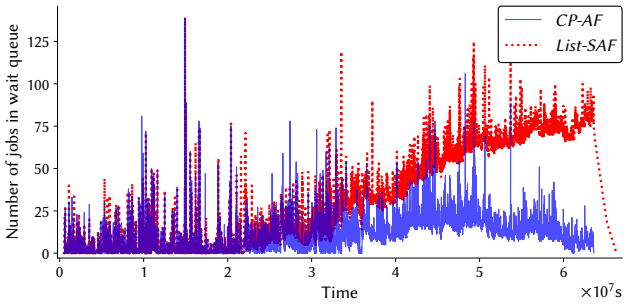


Figure 5: Number of jobs in wait queue as a function of simulation time when *SDSC-SP2* scheduled with *List-SAF* and *CP-AF*.

scheduling optimization when the global objective is also *AF* (in contrast with what we observed for *BSLD*).

5.4 Level-2 Priority-Weighted Specific Time

While the other studied metrics can be considered linear, P^2SF is neither linear nor polynomial. This is possibly a reason why CP Optimizer performs worse with P^2SF as an objective function than

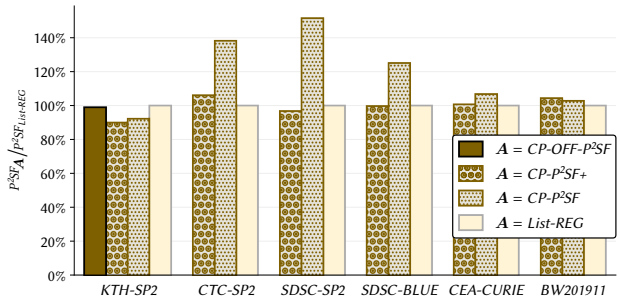


Figure 6: P^2SF of schedules of different workload traces produced by *CP-OFF-P²SF*, *CP-P²SF+*, *CP-P²SF*, and *List-REG* assuming that job runtime is known to the scheduler. Values are shown as percentage of P^2SF for *List-REG*. Lower is better.

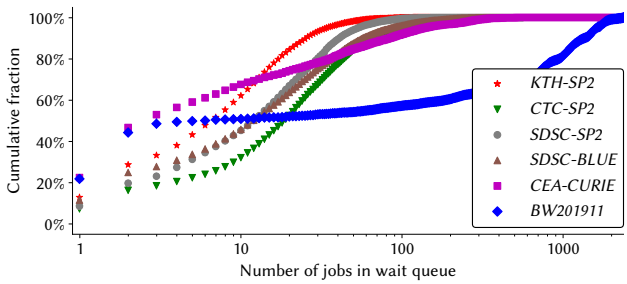


Figure 7: Cumulative distribution of queue size in scheduling rounds of *CP-P²SF*.

with other metrics. *CP-P²SF* manages to improve P^2SF (over *List-REG*) only for *KTH-SP2* but not for the other workloads (Figure 6). Most likely, the heuristics that CP Optimizer uses internally do not work well for non-linear objective functions, which forces CP Optimizer to fall back to searching through all possible combinations. As Figure 7 demonstrates, in the case of *KTH-SP2*, the size of the wait queue mostly stays below 30 jobs, but for the other workloads it often gets larger. The brute force approach is unlikely to succeed for such long queues. However, in case of *KTH-SP2*, the *CP-P²SF* improves P^2SF by 8% over *List-REG*.

To further investigate the causes of the CP approach's underperformance in optimizing P^2SF, we implemented, similarly to *CP-BSLD+* (Section 5.2), *CP-P²SF+* scheduler that, at each scheduling round, makes and refines through optimization with CP Optimizer a scheduling plan using *List-REG*, generates another scheduling plan "from scratch" with CP Optimizer, and selects for the round the plan that has yields the lowest P^2SF value. As seen in Figure 6, *CP-P²SF+* attains lower P^2SF values than *CP-P²SF* for all traces except *BW201911*. *CP-P²SF+* also improves P^2SF over *List-REG* by approx. 10% for *KTH-SP2*, 3% for *SDSC-SP2*, and less than 0.5% for *SDSC-BLUE*. However, P^2SF of the schedules produced by *CP-P²SF+* for the other traces (*CTC-SP2*, *CEA-CURIE*, and *BW201911*) is higher than of the schedules produced by *List-REG*. Therefore, the inferior performance of *CP-P²SF* to a great extent stems from the challenges that CP Optimizer faces in dealing with the non-linear objective

functions. Additionally, several workloads present scenarios, analogous to those described for *BSLD* in Section 5.2, where minimizing P^2SF using the information available during the scheduling rounds is less effective for minimizing P^2SF globally than the heuristics that constitute *List-REG*. *List-REG* finds for each jobs an earliest opportunity the job can run without delaying the jobs with higher priority and ensures this opportunity through resource reservation. This approach is efficient regardless of any knowledge about future jobs. On the other hand, *CP-*P^2SF (as well as *CP-*P^2SF*+*) may deviate from this strategy in pursuit of a temporary enhancement of P^2SF, although such deviation may turn out unwarranted when considering jobs arriving in the future.

5.5 Assessing Real-World Applications

Up to this point, our analysis has assumed that every job runtime is known at the time of scheduling (before job completion time), that is, our schedulers have used $E_j = D_j$ in Algorithm 1 and in (7)–(9). In real-world HPC scheduling, a job's runtime becomes known only after the job terminates. Thus, at the time of scheduling, the scheduler must make decisions based on estimated runtimes of the unfinished jobs. The error in job runtime estimation may unpredictably impact the scheduling efficiency. To evaluate the performance of the algorithms in conditions that closer represent practical applications, we performed scheduling of the studied workloads approximating job runtimes by user-provided timelimits L_j (that is, $E_j = L_j$). In this evaluation, the CP-based schedulers demonstrate the same or better robustness to such errors than the list-based approach. The left part of Table 2 contains the values of the metrics of the schedules produced using actual job runtimes D_j (the values are relative to the values produced by the related *List-REG* that also used D_j), while the right part shows results obtained with user-provided timelimits L_j (correspondingly, the values in the right part are relative to the values produced by *List-REG* using L_j). The correlation between the left and the right parts of the table is very good for *AF*, *AWF*, and P^2SF (meaning that the performance degradation due to runtime estimation inaccuracy is similar between CP and list-based schedulers), while for *BSLD* the performance of *CP-BSLD* is better than the performance of *List-SPF*, *List-SAF*, and *List-SJF* when all algorithms use L_j (although *CP-BSLD* is not the best when using D_j).

Therefore, the CP approach may provide a noticeable improvement of HPC job scheduling, especially in scenarios where the scheduling objective emphasizes packing efficiency and is well represented by metrics that resemble *AF* or *AWF*. Because one of the goals of this study is to evaluate the potential for improvement of job scheduling algorithms, our analysis employs simulations that deviate from real-world scenarios by not accounting for the time and resources utilized by the scheduling algorithms. Proper assessment of the real-world benefits of resource-intensive approaches, such as CP scheduling, must factor in the cost of running the scheduler. Nevertheless, the results presented herein demonstrate that the CP approach for HPC job scheduling warrants a thorough evaluation and may prove cost-effective, particularly for high-maintenance clusters predominantly used to execute large, rigid, and well-characterized jobs. The CP approaches are anticipated to become more advantageous in systems with multiple resources, as the packing efficiency of list-based algorithms is projected to decline under such circumstances. However, investigating the multiple resources problem is beyond the scope of our current analysis.

While discussing the CP optimization scalability, it is worth noting that the list algorithm with *backfill* option we utilize in our analysis also does not scale well with the number of jobs in the queue. Due to this limitation, real-world schedulers often employ *EASY* backfilling [38], which reserves resources for only the first job delayed due to resources shortage, then switches to *greedy* mode. To contrast our investigation's results with this common real-world approach, Table 2 lists the metric values of the schedules produced by *EASY* in our simulations (a detailed evaluation of *EASY* with list-based algorithms is presented in [24]). As *EASY* typically prioritizes smaller jobs by starting them earlier than *List-REG* does, it often improves the values of *BSLD* and *AF* over *List-REG*. However, *EASY* is not designed to optimize any of the considered metrics and, therefore, is outperformed for each metric by more suitable algorithms when the algorithms are allowed to use actual runtime D_i. For instance, *List-REG* outperforms *EASY* in optimizing P^2SF. When the schedulers have no access to D_i and must use the timelimit L_j, the performance becomes more uncertain, and the difference between *EASY* and the other algorithms becomes less pronounced. Occasionally, *EASY* can achieve better metric values by accident.

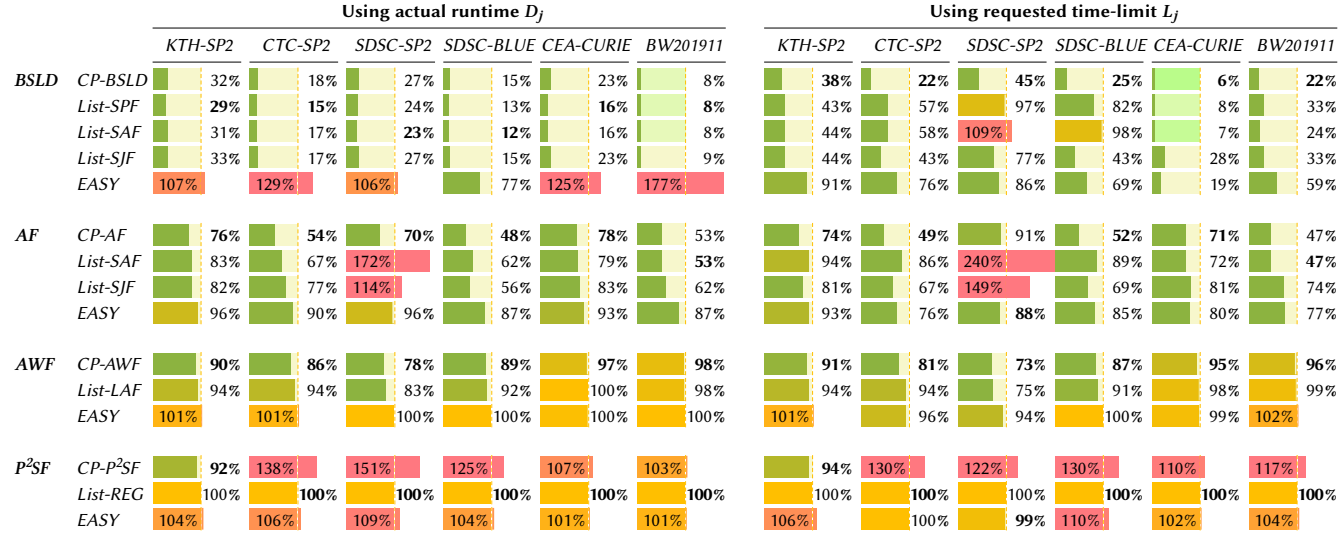
A hybrid of *EASY* and *List-REG*, which improves *EASY*'s fairness but maintains scalability, involves reserving resources for a limited number of jobs and then switching to the *greedy* mode once this limit is reached [8, 29]. Such an algorithm is implemented, for example, in Slurm. Similarly, the CP approach can be part of a hybrid scheduling strategy that schedules a limited number of jobs using CP and switches to an alternative approach for the rest of the job queue.

6 CONCLUSIONS

We compared schedules generated by IBM ILOG CP Optimizer using objective functions based on each of the four metrics (*BSLD*, *AF*, *AWF*, and P^2SF) with schedules produced by the most suitable for each respective metric variants of the list scheduling algorithm. We demonstrate significant improvements in *AF* and *AWF* with the CP-based algorithms, regardless of whether the scheduler utilizes job runtimes D_j or user-requested timelimits L_j. For instance, in the case of *CTC-SP2*, *CP-AWF* improves over the *AWF* value obtained by *List-LAF* by as much as 10% when both utilize D_j and by up to 13% when both utilize L_j. In the case of optimizing *AF*, the enhancement is particularly noteworthy, given the instability of *List-SAF*'s performance (which is considered the best list-based algorithm for *AF* according to the HDF heuristic). This improvement can be attributed to high packing efficiency of CP Optimizer's internal heuristics, whereas the packing efficiency of algorithms like *List-SJF* and *List-SAF* is substandard.

For P^2SF and *BSLD*, the CP approach demonstrated limited improvements for two underlying reasons. In case of *BSLD*, the heuristics SPF (the analogue of HDF for *BSLD*) and SAF are, apparently, so powerful in improving the metric that deviations from it towards optimizing "local" *BSLD* value often lead to a worse outcome globally because the local minimum does not account for the jobs arriving in the future. Results of optimizing P^2SF also suggest that

Table 2: *BSLD*, *AF*, *AWF*, and P^2SF of scheduling algorithms using job runtime or user-requested timelimit. Values are shown as percentage of the value of the corresponding *List-REG* schedule. Lower is better.



in certain situations attempts to minimize "local" P^2SF may be less efficient than using *List-REG* strategy. Additionally, P^2SF, being a non-linear function, presents challenges for CP Optimizer when dealing with large problem instances.

Our investigation highlights the significance of metrics such as *AWF* and P^2SF in evaluating scheduling quality. *AWF* offers insights into packing efficiency, showcasing CP Optimizer's capabilities in this aspect. P^2SF assesses both packing efficiency and fairness, revealing improvements attained in the case of *KTH-SP2*.

CP Optimizer can offer HPC scheduling superior packing efficiency compared to list-based algorithms, particularly evident in the results for *AWF* and *AF*. If the schedulers use L_j instead of D_j, the CP approach also consistently outperforms the list-based algorithms in the case of *BSLD*. This also could be attributed to CP Optimizer's ability to achieve higher packing efficiency, leading to earlier completion of all jobs. In contrast, the most appropriate for *BSLD* list-based algorithms focus only on completing smaller jobs earlier, thus depending more on the accuracy of job runtime estimates. Therefore, CP-based scheduling offer advantages for high-maintenance clusters primarily running a moderate number of large, rigid, and well-characterized jobs if high packing efficiency is a priority. However, practical implementation requires further research and, possibly, adjustments to CP Optimizer to better accommodate HPC scheduling needs. CP-based approaches could prove even more beneficial for scheduling in multi-resource environments, where the packing efficiency of list-based algorithms is expected to decline [13, 36]. However, such cases fall beyond the scope of this investigation. A comprehensive evaluation of scheduling costs and time, essential for accurately assessing the benefits of the CP approach, along with an analysis of alternative scheduling approaches mentioned in Section 3.3, are also subjects of future research. This research may build upon the methodology presented in this work.

ACKNOWLEDGMENTS

The authors thank Benjamin Schwaller and Omar Aaziz for their valuable discussions. The works at the University of Central Florida were supported by contracts with Sandia National Laboratories. The authors acknowledge the University of Central Florida Advanced Research Computing Center for providing computational resources.

REFERENCES

[1] Anurag Agarwal, Selcuk Colak, and Selcuk Erenguc. 2011. A Neurogenetic Approach for the Resource-Constrained Project Scheduling Problem. *Computers & Operations Research* 38, 1 (Jan. 2011), 44–50. https://doi.org/10.1016/j.cor.2010.01.007

[2] Kento Aida, Hironori Kasahara, and Seinosuke Narita. 1998. Job Scheduling Scheme for Pure Space Sharing among Rigid Jobs. In *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science)*, Dror G. Feitelson and Larry Rudolph (Eds.). Springer, Berlin, Heidelberg, 98–121. https://doi.org/10.1007/BFb0053983

[3] Philippe Baptiste, Philippe Laborie, Claude Le Pape, and Wim Nuijten. 2006. Constraint-Based Scheduling and Planning. In *Foundations of Artificial Intelligence*, Francesca Rossi, Peter van Beek, and Toby Walsh (Eds.). Handbook of Constraint Programming, Vol. 2. Elsevier, 761–799. https://doi.org/10.1016/S1574-6526(06)80026-X

[4] Andrea Bartolini, Andrea Borghesi, Thomas Bridi, Michele Lombardi, and Michela Milano. 2014. Proactive Workload Dispatching on the EURORA Supercomputer. In *Principles and Practice of Constraint Programming (Lecture Notes in Computer Science)*, Barry O'Sullivan (Ed.). Springer International Publishing, Cham, 765–780. https://doi.org/10.1007/978-3-319-10428-7_55

[5] Robin Boëzennec, Fanny Dufossé, and Guillaume Pallez. 2023. Optimization Metrics for the Evaluation of Batch Schedulers in HPC. In *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science)*, Dalibor Klusáček, Julita Corbalán, and Gonzalo P. Rodrigo (Eds.). Springer Nature Switzerland, Cham, 97–115. https://doi.org/10.1007/978-3-031-43943-8_5

[6] Thomas Bridi, Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. 2016. A Constraint Programming Scheduler for Heterogeneous High-Performance Computing Machines. *IEEE Transactions on Parallel and Distributed Systems* 27, 10 (Oct. 2016), 2781–2794. https://doi.org/10.1109/TPDS.2016.2516997

[7] D. Carastan-Santos, R. Y. De Camargo, D. Trystram, and S. Zrigui. 2019. One Can Only Gain by Replacing EASY Backfilling: A Simple Scheduling Policies Case Study. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 1–10. https://doi.org/10.1109/CCGRID.2019.00010

[8] Su-Hui Chiang, Andrea Arpaci-Dusseau, and Mary K. Vernon. 2002. The Impact of More Accurate Requested Runtimes on Production Job Scheduling Performance. In *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science)*, Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn (Eds.). Springer, Berlin, Heidelberg, 103–127. https://doi.org/10.1007/3-540-36180-4_7

[9] Giacomo Da Col and Erich Teppan. 2019. Google vs IBM: A Constraint Solving Challenge on the Job-Shop Scheduling Problem. In *Electronic Proceedings in Theoretical Computer Science*, Vol. 306. Open Publishing Association, Las Cruces, NM USA, 259–265. https://doi.org/10.4204/EPTCS.306.30

[10] Jianzhong Du and Joseph Y.-T. Leung. 1989. Complexity of Scheduling Parallel Task Systems. *SIAM Journal on Discrete Mathematics* 2, 4 (Nov. 1989), 473–487. https://doi.org/10.1137/0402042

[11] Nathan Ensmenger. 2018. The Environmental History of Computing. *Technology and Culture* 59, 4 (2018), S7–S33. https://doi.org/10.1353/tech.2018.0148

[12] Maja Etinski, Julita Corbalan, Jesus Labarta, and Mateo Valero. 2011. Linear Programming Based Parallel Job Scheduling for Power Constrained Systems. In *2011 International Conference on High Performance Computing Simulation*. 72–80. https://doi.org/10.1109/HPCSim.2011.5999809

[13] Yuping Fan, Zhiling Lan, Paul Rich, William E. Allcock, Michael E. Papka, Brian Austin, and David Paul. 2019. Scheduling Beyond CPUs for HPC. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 97–108. https://doi.org/10.1145/3307681.3325401

[14] Dror G. Feitelson and Larry Rudolph. 1996. Toward Convergence in Job Schedulers for Parallel Supercomputers. In *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science)*, Dror G. Feitelson and Larry Rudolph (Eds.). Springer, Berlin, Heidelberg, 1–26. https://doi.org/10.1007/BFb0022284

[15] Dror G. Feitelson and Larry Rudolph. 1998. Metrics and Benchmarking for Parallel Job Scheduling. In *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science)*, Dror G. Feitelson and Larry Rudolph (Eds.). Springer, Berlin, Heidelberg, 1–24. https://doi.org/10.1007/BFb0053978

[16] Dror G. Feitelson, Dan Tsafrir, and David Krakov. 2014. Experience with Using the Parallel Workloads Archive. *J. Parallel and Distrib. Comput.* 74, 10 (Oct. 2014), 2967–2982. https://doi.org/10.1016/j.jpdc.2014.06.013

[17] Eitan Frachtenberg and Dror G. Feitelson. 2005. Pitfalls in Parallel Job Scheduling Evaluation. In *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science)*, Dror Feitelson, Eitan Frachtenberg, Larry Rudolph, and Uwe Schwiegelshohn (Eds.). Springer, Berlin, Heidelberg, 257–282. https://doi.org/10.1007/11605300_13

[18] Cristian Galleguillos, Zeynep Kiziltan, Alina Sîrbu, and Ozalp Babaoglu. 2019. Constraint Programming-Based Job Dispatching for Modern HPC Applications. In *Principles and Practice of Constraint Programming (Lecture Notes in Computer Science)*, Thomas Schiex and Simon de Givry (Eds.). Springer International Publishing, Cham, 438–455. https://doi.org/10.1007/978-3-030-30048-7_26

[19] Cristian Galleguillos, Alina Sîrbu, Zeynep Kiziltan, Ozalp Babaoglu, Andrea Borghesi, and Thomas Bridi. 2018. Data-Driven Job Dispatching in HPC Systems. In *Machine Learning, Optimization, and Big Data (Lecture Notes in Computer Science)*, Giuseppe Nicosia, Panos Pardalos, Giovanni Giuffrida, and Renato Umeton (Eds.). Springer International Publishing, Cham, 449–461. https://doi.org/10.1007/978-3-319-72926-8_37

[20] M. R. Garey and R. L. Graham. 1975. Bounds for Multiprocessor Scheduling with Resource Constraints. *SIAM J. Comput.* 4, 2 (June 1975), 187–200. https://doi.org/10.1137/0204015

[21] Eric Gaussier, David Glesser, Valentin Reis, and Denis Trystram. 2015. Improving Backfilling by Using Machine Learning to Predict Running Times. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Austin, TX, USA, 1–10. https://doi.org/10.1145/2807591.2807646

[22] Eric Gaussier, Jérôme Lelong, Valentin Reis, and Denis Trystram. 2018. Online Tuning of EASY-Backfilling Using Queue Reordering Policies. *IEEE Transactions on Parallel and Distributed Systems* 29, 10 (Oct. 2018), 2304–2316. https://doi.org/10.1109/TPDS.2018.2820699

[23] Google. 2023. OR-Tools. https://developers.google.com/optimization.

[24] Alexander V. Goponenko, Kenneth Lamar, Christina Peterson, Benjamin A. Allan, Jim M. Brandt, and Damian Dechev. 2022. Metrics for Packing Efficiency and Fairness of HPC Cluster Batch Job Scheduling. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 241–252. https://doi.org/10.1109/SBAC-PAD55451.2022.00035

[25] Essam H. Houssein, Ahmed G. Gad, Yaser M. Wazery, and Ponnuthurai Nagaratnam Suganthan. 2021. Task Scheduling in Cloud Computing Based on Metaheuristics: Review, Taxonomy, Open Challenges, and Future Trends. *Swarm and Evolutionary Computation* 62 (April 2021), 100841. https://doi.org/10.1016/j.swevo.2021.100841

[26] Matthias Hovestadt, Odej Kao, Axel Keller, and Achim Streit. 2003. Scheduling in HPC Resource Management Systems: Queuing vs. Planning. In *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science)*, Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn (Eds.). Springer, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/10968987_1

[27] Syed Munir Hussain Shah, Kalim Qureshi, and Haroon Rasheed. 2010. Optimal Job Packing, a Backfill Scheduling Optimization for a Cluster of Workstations. *The Journal of Supercomputing* 54, 3 (Dec. 2010), 381–399. https://doi.org/10.1007/s11227-009-0332-3

[28] Sungjin Im, Benjamin Moseley, and Kirk Pruhs. 2014. Online Scheduling with General Cost Functions. *SIAM J. Comput.* 43, 1 (Jan. 2014), 126–143. https://doi.org/10.1137/120902288

[29] David Jackson, Quinn Snell, and Mark Clement. 2001. Core Algorithms of the Maui Scheduler. In *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science)*, Dror G. Feitelson and Larry Rudolph (Eds.). Springer, Berlin, Heidelberg, 87–102. https://doi.org/10.1007/3-540-45540-X_6

[30] Morris A. Jette and Tim Wickberg. 2023. Architecture of the Slurm Workload Manager. In *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science)*, Dalibor Klusáček, Julita Corbalán, and Gonzalo P. Rodrigo (Eds.). Springer Nature Switzerland, Cham, 3–23. https://doi.org/10.1007/978-3-031-43943-8_1

[31] Dalibor Klusáček and Václav Chlumský. 2017. Planning and Metaheuristic Optimization in Production Job Scheduler. In *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science)*, Narayan Desai and Walfredo Cirne (Eds.). Springer International Publishing, Chicago, IL, USA, 198–216. https://doi.org/10.1007/978-3-319-61756-5_11

[32] Dalibor Klusáček, Hana Rudová, Ranieri Baraglia, Marco Pasquali, and Gabriele Capannini. 2008. Comparison Of Multi-Criteria Scheduling Techniques. In *Grid Computing: Achievements and Prospects*, Sergei Gorlatch, Paraskevi Fragopoulou, and Thierry Priol (Eds.). Springer US, Boston, MA, 173–184. https://doi.org/10.1007/978-0-387-09457-1_15

[33] Jochen Krallmann, Uwe Schwiegelshohn, and Ramin Yahyapour. 1999. On the Design and Evaluation of Job Scheduling Algorithms. In *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science)*, Dror G. Feitelson and Larry Rudolph (Eds.). Springer, Berlin, Heidelberg, 17–42. https://doi.org/10.1007/3-540-47954-6_2

[34] Parallel Systems Lab. 2008. The Python Scheduler Simulator. https://code.google.com/archive/p/pyss/.

[35] Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. 2018. IBM ILOG CP Optimizer for Scheduling. *Constraints* 23, 2 (April 2018), 210–250. https://doi.org/10.1007/s10601-018-9281-x

[36] W. Leinberger, G. Karypis, and V. Kumar. 1999. Job Scheduling in the Presence of Multiple Resource Requirements. In *SC '99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*. 47–47. https://doi.org/10.1109/SC.1999.10067

[37] Vitus J. Leung, Gerald Sabin, and P. Sadayappan. 2010. Parallel Job Scheduling Policies to Improve Fairness: A Case Study. In *2010 39th International Conference on Parallel Processing Workshops*. IEEE, San Diego, CA, USA, 346–353. https://doi.org/10.1109/ICPPW.2010.48

[38] David A. Lifka. 1995. The ANL/IBM SP Scheduling System. In *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science)*, Dror G. Feitelson and Larry Rudolph (Eds.). Springer, Berlin, Heidelberg, 295–303. https://doi.org/10.1007/3-540-60153-8_35

[39] Fiona Fui-Hoon Nah. 2004. A Study on Tolerable Waiting Time: How Long Are Web Users Willing to Wait? *Behaviour & Information Technology* 23, 3 (May 2004), 153–163. https://doi.org/10.1080/01449290410001669914

[40] PBS Works. 2024. OpenPBS Open Source Project. https://www.openpbs.org/.

[41] D. Perkovic and P.J. Keleher. 2000. Randomization, Speculation, and Adaptation in Batch Schedulers. In *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE, Dallas, TX, USA, 7–7. https://doi.org/10.1109/SC.2000.10041

[42] Kirk Pruhs, Jiri Sgall, and Eric Torng. 2004. Online Scheduling. In *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman & Hall/CRC, Boca Raton, FL, USA.

[43] Francesca Rossi, Peter van Beek, and Toby Walsh. 2006. *Handbook of Constraint Programming*. Elsevier.

[44] Uwe Schwiegeishohn and Ramin Yahyapour. 1998. Improving First-Come-First-Serve Job Scheduling by Gang Scheduling. In *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science)*, Dror G. Feitelson and Larry Rudolph (Eds.). Springer, Berlin, Heidelberg, 180–198. https://doi.org/10.1007/BFb0053987

[45] Uwe Schwiegelshohn. 2011. A System-Centric Metric for the Evaluation of Online Job Schedules. *Journal of Scheduling* 14, 6 (Dec. 2011), 571–581. https://doi.org/10.1007/s10951-010-0206-9

[46] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. 2002. Characterization of Backfilling Strategies for Parallel Job Scheduling. In *Proceedings. International Conference on Parallel Processing Workshop*. IEEE, Vancouver, BC, Canada, 514–519. https://doi.org/10.1109/ICPPW.2002.1039773

[47] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and Ponnuswamy Sadayappan. 2002. Selective Reservation Strategies for Backfill Job Scheduling. In *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science)*, Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn (Eds.). Springer, Berlin, Heidelberg, 55–71. https://doi.org/10.1007/3-540-36180-4_4

[48] D. Talby and D.G. Feitelson. 1999. Supporting Priorities and Improving Utilization of the IBM SP Scheduler Using Slack-Based Backfilling. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*. IEEE, San Juan, PR, USA, 513–517. https://doi.org/10.1109/IPPS.1999.760525

[49] Blue Waters Team. 2022. Blue Waters User Portal. https://bluewaters.ncsa.illinois.edu/data-sets.

[50] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. 2007. Backfilling Using System-Generated Predictions Rather than User Runtime Estimates. *IEEE Transactions on Parallel and Distributed Systems* 18, 6 (June 2007), 789–803. https://doi.org/10.1109/TPDS.2007.70606

[51] Chun-Wei Tsai and Joel J. P. C. Rodrigues. 2014. Metaheuristic Scheduling for Cloud: A Survey. *IEEE Systems Journal* 8, 1 (March 2014), 279–291. https://doi.org/10.1109/JSYST.2013.2256731

[52] John Turek, Joel L. Wolf, and Philip S. Yu. 1992. Approximate Algorithms Scheduling Parallelizable Tasks. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '92)*. Association for Computing Machinery, New York, NY, USA, 323–332. https://doi.org/10.1145/140901.141909

[53] A. Verma, M. Korupolu, and J. Wilkes. 2014. Evaluating Job Packing in Warehouse-Scale Computing. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Madrid, Spain, 48–56. https://doi.org/10.1109/CLUSTER.2014.6968735

[54] Fatos Xhafa, Ajith Abraham, and Janusz Kacprzyk (Eds.). 2008. *Metaheuristics for Scheduling in Distributed Computing Environments*. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-69277-5

[55] Xingwu Zheng, Zhou Zhou, Xu Yang, Zhiling Lan, and Jia Wang. 2016. Exploring Plan-Based Scheduling for Large-Scale Computing Systems. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Taipei, Taiwan, 259–268. https://doi.org/10.1109/CLUSTER.2016.43

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009